UNIVERSIDADE FEDERAL FLUMINENSE

ALEXANDRE DA COSTA SENA

Um Modelo Alternativo para Execução Eficiente de Aplicações Paralelas MPI nas Grades Computacionais

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Doutor. Área de concentração: Processamento Paralelo e Distribuído.

Orientadores: Cristina Boeres, PhD
Vinod Rebello, PhD
Programa de Pós-Graduação em Computação - IC/UFF

Niterói, 25 de novembro de 2008

Para Lele, amor da minha vida.

Agradecimentos

Agradeço primeiramente a Deus, pois sem Sua presença na minha vida nada faria sentido nem valeria a pena.

Agradeço também aos meus orientadores Cristina e Vinod pela paciência e dedicação durante todo o trabalho realizado, pelo incentivo na hora do desânimo e pela liderança na hora da dúvida.

Em especial, quero agradecer a minha esposa Aline que esteve ao meu lado durante toda esta jornada, não só por sua paciência e compreensão nos momentos em que não pude dar a devida atenção, mas também pela sua ajuda em vários momentos de dificuldade e desânimo.

Aos meus pais, que desde a minha infância me ensinaram a importância de estudar. A minha querida mãe que está sempre ao meu lado nas alegrias e tristezas da vida e ao meu pai pelo incentivo.

Aos meus familiares e amigos pela compreensão nas horas em que estive ausente. Ao amigo Jacques, por seu apoio nos momentos de dificuldade.

Resumo

Apesar do crescimento exponencial da capacidade de processamento e armazenamento dos computadores nas últimas décadas, existe uma demanda cada vez maior por poder computacional. Com o advento das grades computacionais (*computational grids*), é possível o acesso a um número maior de recursos a um baixo custo. Porém, diferentemente dos *clusters* de computadores que normalmente são compostos de um número limitado de recursos homogêneos e dedicados, as grades computacionais são compostas por recursos dinâmicos, heterogêneos e compartilhados que podem estar dispersos geograficamente e sujeitos a políticas de acesso distintas. Essas características fazem com que a execução eficiente de programas paralelos nesses ambientes seja uma tarefa complexa.

Mesmo com todas as diferenças entre os ambientes tradicionais de computação de alto desempenho e as grades, a maioria dos programas continua sendo executada em ambientes de grade utilizando o modelo padrão de execução, que aloca um processo por processador durante toda duração da aplicação. Com o objetivo de explorar melhor as principais características das grades computacionais como, por exemplo, heterogeneidade, compartilhamento e dinamismo, esse trabalho propõe a adoção de um modelo alternativo de execução onde a principal característica é a execução de uma tarefa por processo. Mais importante, através desse modelo alternativo, este trabalho de tese mostra a viabilidade de se executar eficientemente aplicações fortemente acopladas nas grades.

Ainda nesta tese, o modelo proposto é validado através de experimentos em ambientes grades reais utilizando comparações de desempenho entre programas que executam sob o modelo tradicional e sob o modelo alternativo. Além disso, outros experimentos destacam as vantagens do novo modelo para implementação de mecanismos de escalonamento de tarefas e tolerância a falhas. Os resultados mostram a eficácia do modelo proposto para ambientes de grade conseguindo não só ser melhor do que o modelo tradicional mas também obter resultados próximos de um limite inferior em vários experimentos, inclusive para aplicações fortemente acopladas.

Como as estimativas iniciais sobre a aplicação e o ambiente de execução podem ser imprecisas, esse trabalho também avalia a sensibilidade da estratégia proposta as imprecisões do modelo da aplicação e da arquitetura. Os resultados mostram que é possível conseguir bom desempenho mesmo que as estimativas iniciais sobre as tarefas da aplicação e sobre as condições dos recursos não sejam precisas.

Abstract

Although the processing and storage capacity of computers has grown exponentially over the last decades, the demand for more computational power by the scientists, engineers and economists continues. Computational grids allow users to access large amount of resources at a low cost. However, different from clusters that are usually composed of a limited number of homogeneous dedicated resources, grids are composed of heterogeneous non-dedicated resources, which may not only be geographically dispersed but also subject to different access policies. These characteristics make the efficient execution of parallel applications in these environments a complex task.

In spite of all the differences between tradicional high performance computing environments and the grid, most applications are still being executed in grid environments under the standard execution model that allocates one process per processor for the entire duration of the application's execution. With the aim of exploring grid characteristics such as heterogeneity resource, sharing and dynamism, better, this work proposes an alternative execution model with the principal characteristic being the execution of one task per process. Through this alternative model, this thesis shows the feasibility of executing tightly coupled parallel applications in the grid.

The proposed model is validated through experiments in actual grid environments by comparing the performance between programs executing under the traditional and the alternative models. Further experiments highlight the benefits of the new model for task scheduling and fault tolerant mechanisms. The results show the effectiveness of the proposed model for grid environments achieving not only better performance than the traditional one but also achieving results close to the lower bound in some experiments, including experiments for tightly coupled applications.

Assuming that the initial estimates about the application and the execution environment may not be accurate, this work also evaluates the sensitivity of the proposed strategy to inaccuracies in the application and architectural models. The results show the efficiency of the alternative execution model, even when the initial estimates about the application's tasks and the resource conditions are not accurate.

Palavras-chave

1. Grades Computacionais
2. Modelagem de Execução de Aplicações
3. Aplicação MPI

Sumário

1 Introdução **1**

1.1 Contexto 4

1.2 Objetivos 5

1.3 Contribuições 6

1.4 Organização 7

2 Motivação e Trabalhos Relacionados **8**

2.1 Grades Computacionais 8

2.2 Aplicações MPI em Ambientes Heterogêneos 9

2.3 Representação de uma Aplicação Paralela 13

2.3.1 Gerenciamento e Geração de Programas representados por Grafos . . 15

2.4 Modelos Arquiteturais 17

2.5 Heurísticas de Escalonamento 19

2.6 Resumo 21

3 Modelo Alternativo para Execução de Aplicações MPI em Ambientes de Grades **23**

3.1 Modelo de Execução Alternativo 23

3.1.1 Modelo de Execução Tradicional - 1PProc 24

3.1.1.1 Execução de mais de um processo por processador 25

3.1.2 Modelo de Execução Alternativo - 1PTask 27

3.1.2.1 Escalonamento de Tarefas e Tolerância a Falhas no modelo 1PTask 29

3.1.2.2 Gerenciamento de Aplicações Paralelas através do Modelo 1PTask 30

3.2 Modelo da Aplicação 31
3.2.1 Escolha do Modelo de Aplicação 32
3.2.2 Modelo Adotado: GAD 34
3.3 Modelo Arquitetural 34
3.3.1 Modelo de Computação 35
3.3.1.1 Índice de Retardo Computacional 35
3.3.2 Modelo de Comunicação 37
3.4 Resumo 38

4 Validação do Modelo Alternativo para Aplicações bag-of-tasks 40

4.1 O Projeto *EasyGrid* 40
4.1.1 O *middleware EasyGrid* AMS 42
4.1.2 O Escalonamento de Tarefas do EasyGrid AMS 45
4.2 Avaliação do Modelo GAD para Aplicações *bag-of-tasks* 47
4.2.1 Metodologia para Estimar o Peso das Tarefas da Aplicação *Sched* . . 48
4.3 Avaliação e Análise do Modelo de Computação 50
4.3.1 Grau de concorrência 54
4.3.2 Desempenho e Escalabilidade 58
4.3.3 Desempenho do *EasyGrid AMS* na presença de outras aplicações . . 60
4.4 Avaliação do Modelo 1PTask para Aplicações *bag-of-tasks* 63
4.4.1 Aplicações Utilizadas 64
4.4.2 Ambiente de execução 65
4.4.3 Análise da Execução em Recursos Heterogêneos 66
4.4.4 Dinamismo da Grade x Políticas Distintas 69
4.4.5 Falha de Processos nas Grades 71
4.5 Resumo 74

5 Validação do Modelo Alternativo para Aplicações Paralelas 75

5.1 Avaliação do Modelo de Comunicação 75

Sobrecargas de Comunicação no *EasyGrid AMS* 77

5.2 Avaliação e Análise de Aplicações Paralelas representadas através de GADs 81

5.2.1 Aplicação Paralela N-corpos 82

5.2.1.1 O Algoritmo *ring* 83

5.2.1.2 Ambientes Homogêneos com Comunicação Não Nula 84

5.2.1.3 Ambientes Heterogêneos 86

5.2.1.4 Gridificando o Algoritmo *ring* 87

5.3 Avaliação do Modelo 1PTask para Aplicações Fortemente Acopladas 93

5.3.1 Ambiente Homogêneo 94

5.3.2 Ambiente Heterogêneo 95

5.3.3 Escolhendo o valor de W e os Recursos Apropriados 96

5.3.4 Distribuição Heterogênea 100

5.3.5 Grau de Heterogeneidade 102

5.3.6 Ambiente Compartilhado 103

5.3.7 Ambiente Grade 105

5.4 Resumo 108

6 Sensibilidade da Estratégia Proposta em Relação à Precisão dos Modelos da Aplicação e da Arquitetura 109

6.1 Escalonamento Híbrido para Aplicações com Tarefas Independentes 110

LPT 111

BESTFIT 111

MULTIFIT 112

6.1.1 Análise Experimental - Simulação 114

6.1.2 Prioridades para o Escalonamento Híbrido 115

Efeito Sobre o Escalonamento Dinâmico 116

6.1.3 Impacto da Ausência de Informação sobre o Ambiente no Escalonamento de Tarefas de Aplicações *bag-of-tasks* 119

Ambiente Dinâmico 121

6.1.4 Impacto de Estimativas Imprecisas no Escalonamento de Aplicações *bag-of-tasks* 122

6.1.4.1 Sem a execução do escalonador dinâmico 122

6.1.4.2 Execução Considerando a Estratégia Híbrida 124

6.2 Escalonamento Híbrido para uma Aplicação Paralela com Dependências entre Tarefas 128

6.2.1 Impacto da Ausência de Informação sobre o Ambiente no Escalonamento de Tarefas de Aplicações Paralelas com Dependência 128

6.2.1.1 Ambiente Dinâmico 129

6.2.2 Impacto de Estimativas Imprecisas no Escalonamento de Aplicações Paralelas com Dependências 131

6.3 Resumo 135

7 Conclusões e Trabalhos Futuros 137

7.1 Trabalhos Futuros 139

Apêndice A – Configurações dos Recursos Utilizados nos Experimentos 141

Referências 143

Capítulo 1

Introdução

Apesar do crescente aumento da capacidade de processamento e armazenamento dos computadores nas últimas décadas [77], assim como da transmissão de dados entre eles, existe uma demanda cada vez maior por poder computacional por parte dos cientistas, engenheiros, economistas, entre outros, tanto em áreas de pesquisa como de negócios. Essa demanda se deve a uma variedade de problemas complexos que vêm sendo estudados e também ao grande aumento na quantidade de dados a serem analisados, que surgem como consequência dos avanços tecnológicos e das pesquisas de maneira geral. Por exemplo, entre esses problemas [60, 76] é possível destacar: o projeto Genoma, que tem o objetivo de desvendar o código genético de vários organismos e com isso prevenir e obter tratamentos para doenças; simulações climáticas que estudam o efeito estufa com o objetivo de alcançar conhecimento para impedir o aquecimento do planeta Terra; a descoberta de novos planetas ou satélites que já possuíram ou onde seres vivos podem ser encontrados; e o maior acelerador de partículas do mundo *Large Hadron Collider* [156, 103] que está sendo construído no CERN (*Organisation Européenne pour la Recherche Nucléaire*) com previsão de produzir múltiplos *petabytes* de dados por ano [62].

Até bem pouco tempo, os supercomputadores eram uma das mais poderosas opções para computação de alto desempenho. Porém, devido ao alto custo e consequentemente, limitado acesso, poucos cientistas conseguiam utilizar esse tipo de plataforma. Com o surgimento da computação em *cluster* (*cluster computing* [25]), que foi baseado em *commodity componentes* ou COTS (*Components of the shelf*), os pesquisadores passaram a ter acesso a uma plataforma paralela de pequeno e médio porte de custo acessível. Nesses ambientes compostos de recursos homogêneos e dedicados, o grande desafio para o pesquisador era paralelizar a sua aplicação sequencial ou desenvolver uma versão paralela que executasse eficientemente. Porém, caso o usuário necessitasse de uma quantidade maior de poder computacional, seja para executar instâncias maiores ou para diminuir o tempo de execução

das instâncias existentes, a única opção era aumentar o seu *cluster* ou adquirir um novo *cluster* mais rápido, o que muitas vezes não era possível devido a limitações financeiras ou de espaço físico.

Devido ao compartilhamento de recursos entre instituições através de grades computacionais (*computational grids*) [65], é possível oferecer aos usuários uma ordem a mais de poder computacional a um baixo custo. No entanto, diferentemente dos ambientes tradicionais de computação de alto desempenho, as grades computacionais são ambientes dinâmicos compostos por recursos heterogêneos, não dedicados e sujeitos a regras de acesso distintas. Em razão dessas características, vários desafios devem ser superados para que as aplicações possam ser executadas eficientemente nas grades computacionais. Por serem compostas por recursos heterogêneos e por sua natureza dinâmica, onde recursos podem se juntar ou deixar a grade a qualquer momento, a distribuição e mais especificamente, o escalonamento de tarefas é mais difícil do que em ambientes tradicionalmente homogêneos e estáticos. Além disso, os recursos disponíveis podem ser compartilhados com outras aplicações e usuários, o que faz com que a capacidade de processamento de um recurso, e da grade como um todo, modifique durante a execução de uma aplicação, podendo assim, prejudicar o seu desempenho. Uma outra dificuldade é que os recursos de uma grade podem estar distribuídos geograficamente, conectados através de diferentes redes com capacidades distintas, o que pode afetar também o desempenho da aplicação. Esses recursos pertencentes a diferentes instituições podem não estar disponíveis ou ter ainda políticas de acesso distintas, o que pode causar atraso na execução de tarefas dependendo das prioridades de acesso oferecidas. Além disso, não se deve descartar uma maior possibilidade de falhas, tanto dos recursos como de componentes da aplicação, durante a execução desta.

Pesquisas atuais desenvolvidas para grades computacionais, consideram duas classes de aplicações principais: aplicações compostas apenas de tarefas independentes e aplicações paralelas comnposta de tarefas com relações de precedência. Muitos pesquisadores mostraram que as grades computacionais se adequam bem para resolver programas formados apenas por tarefas independentes [2, 8, 41], onde o paralelismo é explícito e imediato. Tais aplicações são usualmente chamadas de *bag-of-tasks*, onde, de forma geral, tarefas podem ser executadas em qualquer recurso, desde que seus arquivos de entrada/saída estejam disponíveis. Por outro lado, executar eficientemente em grades programas paralelos que se caracterizam por possuírem comunicação entre as suas tarefas, ainda é um desafio. Além de um conhecimento maior sobre a aplicação, são necessárias também informações sobre a topologia da rede e mais precisamente sobre a capacidade dos canais de comunicação que conectam os recursos. Atualmente, alguns poucos trabalhos vêm sendo desenvolvidos com aplicações do tipo *Workflow* [28, 39], onde o foco é muito mais facilitar o uso das grades do

que minimizar o tempo de execução de cada aplicação. Aplicações *Workflow* apresentam algum grau de dependência entre as suas tarefas (sub-aplicações) e se caracterizam por possuírem tarefas (aplicações) relativamente longas e comunicações realizadas através de entrada e saída de arquivos.

Ainda com relação as aplicações paralelas, segundo Feitelson e Rudolph [55] elas podem ser classificadas em quatro categorias. Enquanto aplicações **rigidas** (*rigid*) somente executam em um número fixo pré-programado de recursos, aplicações **moldáveis** (*mouldable*) podem ser executadas em uma quantidade variável de processadores (embora a alocação de processos se mantenha fixa durante toda a execução da aplicação). Diferentemente, as aplicações **evolutivas** (*evolving*) e **maleáveis** (*malleable*) podem alterar o número de processos durante a execução. Uma aplicação **evolutiva** é capaz de tomar decisões por si mesma, enquanto aplicações maleáveis tipicamente necessitam o suporte de um sistema gerenciador de recursos específico. Em [106] foi estendida a definição de maleabilidade também para aplicações que ajustam a granularidade de seus processos durante a execução.

O foco desta tese de doutorado é identificar uma metodologia que possibilite a execução eficiente de aplicações paralelas, especialmente as aplicações fortemente acopladas, nas grades computacionais. Adicionalmente, escolhemos aplicações paralelas desenvolvidas utilizando a biblioteca MPI, que se tornou o padrão *de facto* de troca de mensagens entre os cientistas [49, 61]. Como grande parte dos programas que utilizam a biblioteca MPI foi desenvolvido para executar em ambientes estáticos, homogêneos e dedicados (por exemplo, *clusters*), normalmente, um número fixo de processos é criado estaticamente, um processo por recurso, que executam até o final da aplicação [61]. Entretanto, apesar das grades computacionais oferecerem um ambiente completamente distinto dos ambientes tradicionais de alto desempenho, de uma maneira geral a execução das aplicações nas grades continua sendo feita através deste modelo de execução tradicional, o que limita a eficiência das aplicações paralelas nestes ambientes.

Para que uma aplicação paralela MPI execute eficientemente e com robustez em uma grade computacional, este trabalho de tese propõe um modelo alternativo de execução. Neste caso, ao invés do modelo tradicional MPI dependente da arquitetura (onde o número de processos MPI é igual ao número de processadores disponíveis), o número de processos MPI é proporcional ao grau de paralelismo da aplicação. Através desse novo modelo é possível tornar uma aplicação fortemente acoplada moldável em evolutiva, de maneira que ela possa se ajustar as mudanças que ocorram no ambiente de execução e com isso melhorar seu desempenho. A descrição deste novo modelo, assim como os benefícios desta nova abordagem, especialmente para os mecanismos de escalonamento de tarefas e tolerância a

falhas, são apresentados ao longo deste trabalho.

1.1 Contexto

Atualmente dois tipos de ambientes de grades disponíveis para pesquisadores podem ser destacados. Um tipo identificado disponibiliza os recursos da grade dedicados para cada usuário/aplicação, como por exemplo Grid'5000 [20, 80] da França e o EGEE (*Enabling Grids for E-sciencE*) [51] que conecta mais de 50 países. Nestes ambientes, o próximo programa (ou conjunto de tarefas de uma aplicação *bag-of-tasks*) a ser executado informa a quantidade de recursos e o tempo necessários para executar e um sistema gerenciador disponibiliza os recursos. Durante o período alocado para a aplicação os recursos ficam dedicados a sua execução, não importando se estão sendo efetivamente utilizados ou não. Por outro lado, existem as grades formadas pela cooperação entre organizações. Neste caso, a grade é formada por recursos compartilhados que são disponibilizados pelas organizações que fazem parte da grade. Neste tipo de ambiente, qualquer usuário autorizado pode executar sua aplicação em qualquer recurso, mesmo que outro usuário já esteja utilizando os recursos. No Brasil, várias universidades e laboratórios científicos disponibilizam recursos desta forma e com isso conseguem o acesso a um ambiente com maior poder computacional.

Em qualquer um desses ambientes, para que uma aplicação execute eficientemente é fundamental um bom escalonamento de suas tarefas, assim como tolerância a falhas para que, em caso de problemas no acesso aos recursos, não seja necessário reexecutar a aplicação desde o início. Entretanto, de uma maneira geral, os pesquisadores que necessitam utilizar as grades não estão aptos para implementar tais mecanismos nas suas aplicações. Com o objetivo de facilitar o acesso dos cientistas as grades, a execução das aplicações é feita através de sistemas gerenciadores de recursos ou de aplicações pertencentes à camada de *middleware*. Três abordagens distintas podem ser destacadas: um sistema gerenciador de recursos (RMS) que se preocupa com o sistema como um todo, e não com o desempenho de cada aplicação; um sistema gerenciador de usuários (UMS) que adota uma visão direcionada ao usuário, gerenciando apenas aplicações de um único usuário por vez; um sistema gerenciador da aplicação (AMS) que tem o objetivo de maximizar o desempenho de uma única aplicação de maneira que ela possa executar o mais rápido possível.

Se o objetivo for maximizar o desempenho da aplicação, então um sistema gerenciador da aplicação (AMS) é a melhor escolha. Entretanto, a eficiência da aplicação é totalmente dependente dos mecanismos de escalonamento de tarefas e tolerância a falhas disponíveis.

Dois tipos de implementação para o problema de escalonamento de tarefas podem ser

destacados: estático e dinâmico. Enquanto que o escalonamento estático é realizado antes da execução da aplicação, baseado apenas em estimativas sobre a aplicação e o ambiente, o escalonamento dinâmico age durante a execução, a partir das mudanças ocorridas no ambiente. Visto que as grades computacionais são ambientes dinâmicos e compartilhados é necessário o uso de um escalonador dinâmico para que a aplicação se adapte às mudanças ocorridas no ambiente. Porém, enquanto que as heurísticas estáticas podem ser complexas e analisar toda a aplicação já que não concorrem com a sua execução, as heurísticas dinâmicas tem que ser leves o suficiente para não prejudicar o desempenho da aplicação do usuário, já que estas executam juntamente com aplicação.

1.2 Objetivos

O objetivo central deste trabalho é viabilizar um bom desempenho das aplicações paralelas nas grades computacionais, aplicações estas que na sua maioria são desenvolvidas por cientistas não especialistas em ciência da computação. Tendo em vista que as aplicações nas grades são executadas através de sistemas gerenciadores, o desafio é fornecer as informações necessárias para que estes sistemas gerenciadores consigam uma execução eficaz das aplicações nesses ambientes complexos. Para atingir este objetivo, este trabalho propõe a adoção de uma estratégia inovadora baseada nos seguintes modelos: modelo de execução, modelo da aplicação e modelo arquitetural. O modelo de execução define não somente como as tarefas das aplicações são executadas nos ambientes de grades, mas também o que é um processo e uma tarefa para uma aplicação paralela. Por exemplo, uma aplicação pode ser composta de várias tarefas, mas ser implementada de maneira que várias destas tarefas estejam dentro de um mesmo processo. Por outro lado, uma outra opção seria a implementação de um processo para cada tarefa da aplicação. O modelo da aplicação representa as características importantes de uma aplicação que devem ser informadas ao sistema gerenciador para que este consiga uma execução correta e eficiente. Já, o modelo arquitetural informa as características sobre os recursos do ambiente a ser utilizado.

Para maximizar o desempenho das aplicações paralelas nos ambientes de grade a estratégia acima é definida para um sistema gerenciador da aplicação, juntamente como os mecanismos de escalonamento de tarefas e tolerância a falhas. Dessa maneira, é necessário que o sistema gerenciador implemente o modelo de execução proposto, assim como os mecanismos de escalonamento de tarefas e tolerância a falhas e com isso consiga gerenciar a execução da aplicação nos ambientes de grade de maneira robusta e eficaz. A partir desta estratégia proposta é possível transformar uma aplicação paralela moldável em uma aplicação evolutiva, capaz de auto-ajustar sua execução e, com isso, se adaptar as mudanças

que ocorrem na grade, melhorando seu desempenho.

1.3 Contribuições

A principal contribuição deste trabalho é propor e validar um modelo alternativo de execução para as grades computacionais que permite que os programas sejam executados de forma eficiente nas grades adaptando sua execução às características do ambiente disponível. Baseado nesse modelo alternativo, esta tese propõe a representação das características de uma aplicação através do modelo de Grafo Acíclico Direcionado (GAD) e um modelo arquitetural para representar as características da grade.

A validação do modelo proposto é feita através de experimentos em ambientes de grade reais, comparando o desempenho de aplicações que executam através do modelo tradicional e através do modelo alternativo. Para isso, é utilizado o sistema gerenciador *EasyGrid AMS* que atinge grande eficiência ao considerar o modelo proposto nesta tese. Apesar do *middleware EasyGrid AMS* ser baseado no modelo proposto nesta tese, este trabalho não discute como ele foi implementado, mas apenas apresenta uma descrição geral sobre seu funcionamento.

É importante destacar que neste trabalho são investigadas tanto as aplicações compostas por tarefas independentes (amplamente utilizadas nas grades) como também as aplicações paralelas compostas por tarefas dependentes (pouco utilizadas nos ambientes de grade). Além disso, os benefícios do novo modelo de execução para estratégias de escalonamento dinâmico e tolerância a falhas também são mostrados. Os resultados obtidos mostram a eficácia do modelo alternativo para ambientes de grade reais, não somente por obter desempenho superior ao modelo tradicional, mas também por permitir atingir resultados próximos de um limite inferior.

Uma segunda contribuição muito importante desta tese, consequência do modelo alternativo proposto, é mostrar a viabilidade de se executar aplicações fortemente acopladas nas grades computacionais, transformando uma aplicação tradicionalmente moldável em evolutiva. Para isso, inicialmente é analisado o algoritmo *ring* para o problema N-corpos desenvolvido para ambientes homogêneos. A aplicação N-corpos foi escolhida por ser altamente eficiente em ambientes homogêneos (*clusters*) e por sua estrutura paralela, ainda que fortemente acoplada, ser similar a uma grande quantidade de aplicações de simulação. Em seguida, é proposto uma adaptação desse algoritmo onde, apesar de pequenas modificações no código, a principal mudança é conceitual, ou seja, a utilização do modelo alternativo proposto que executa uma tarefa por processo. Além disso, uma análise desse novo algoritmo é apresentada junto com uma estratégia para alcançar uma execução eficiente nos

ambientes heterogêneos, dinâmicos e compartilhados como as grades computacionais. Resultados obtidos em ambientes de grade reais utilizando a aplicação N-corpos mostram que a aplicação executando sob o modelo de execução proposto foi até 4,3 vezes mais rápida do que quando executando utilizando o modelo de execução tradicional.

Caso as estimativas iniciais sobre a aplicação e o ambiente de execução não sejam precisas, esse trabalho também investiga, através de experimentos, a sensibilidade da estratégia proposta a precisão dos modelos da aplicação e da arquitetura. Os resultados obtidos mostram o bom desempenho da estratégia proposta, escalonamento híbrido baseado no modelo de execução alternativo, destacando a capacidade do escalonador dinâmico de adaptar a execução da aplicação apesar das imprecisões fornecidas pelos modelos.

1.4 Organização

Este trabalho está dividido da seguinte forma. Primeiramente é apresentada a motivação para a realização deste trabalho, junto com alguns trabalhos relacionados (Capítulo 2). No Capítulo 3 é apresentado o modelo alternativo proposto nesta tese, assim como os modelos da aplicação e arquitetural, fundamentais para a representação da aplicação e do ambiente de execução. A validação do modelo alternativo é dividida em dois capítulos. O Capítulo 4 avalia o modelo de execução proposto para as aplicações compostas de tarefas que não se comunicam entre si. Em seguida, o Capítulo 5 apresenta a viabilidade de se executar aplicações fortemente acopladas nas grades. A sensibilidade da estratégia proposta a erros nas estimativas sobre a aplicação e o ambiente de execução é analisada no Capítulo 6. Por último, no Capítulo 7, são apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Motivação e Trabalhos Relacionados

Este capítulo apresenta a principal motivação deste trabalho, que é a execução eficiente de programas MPI nas grades computacionais. Para isso, inicialmente é mostrado como as aplicações paralelas vem sendo executadas nas grades computacionais e, em seguida, mais especificamente, como vem sendo realizada à execução das aplicações MPI. Além disso, este capítulo também apresenta ferramentas e modelos existentes para execução e representação de aplicações paralelas, assim como os modelos arquiteturais relevantes para representação dos ambientes de execução.

2.1 Grades Computacionais

O ambiente de Grade (*Grid Computing*) [64, 65] já é uma realidade, porém nem todos os cientistas podem utilizá-lo devido às complexidades inerentes a este tipo de ambiente. Logo, o desafio é fazer com que os usuários e programadores possam acessá-lo de uma maneira simples e eficiente.

Enquanto que nos ambientes de alto desempenho tradicionais os usuários utilizam os recursos quase que exclusivamente, as grades são baseadas no compartilhamento de recursos em larga escala [122]. No entanto, esse compartilhamento permite que os usuários acessem uma maior quantidade de recursos. Além disso, a própria execução de uma aplicação em uma grade envolve uma série de serviços e camadas que podem introduzir uma sobrecarga considerável [53]. Estas características tornam a grade um ambiente complexo para execução de programas. A heterogeneidade acarreta uma maior dificuldade na alocação de tarefas a recursos, quando comparado a um ambiente com recursos homogêneos. Os recursos de uma grade, por serem dinâmicos e compartilhados, podem oferecer capacidade de processamento variada durante a execução da aplicação. Além disso, por possuírem redes de capacidade distintas que conectam instituições situadas até em diferentes conti-

nentes, a troca de dados e a submissão de tarefas podem sofrer atraso considerável. O controle de uma grade é distribuído, de modo que tarefas de usuários locais podem ter maior prioridade do que tarefas da aplicação paralela remota, dependendo da política de acesso utilizada.

Em razão das dificuldades de executar uma aplicação em uma grade computacional e da necessidade de lidar com outros mecanismos importantes, tais como, tolerância a falhas e escalonamento dinâmico, o acesso às grades computacionais tem sido feito através de *middleware* de gerenciamento, que servem de interface entre o cientista e a grade, simplificando a execução da aplicação. Entre as funções que estes sistemas gerenciadores podem desempenhar podemos citar a distribuição de arquivos [12], o escalonamento de tarefas [14, 67, 119], tolerância a falhas [45] e avaliação de desempenho [28]. Conforme discutido em [18, 97, 116, 120], os sistemas gerenciadores podem adotar uma das três filosofias a seguir: Sistemas gerenciadores de recursos, *RMS (Resource Management Systems)*, Sistemas gerenciadores de usuários, *UMS (User Management Systems)* e Sistemas gerenciadores de aplicações, *AMS (Application Management Systems)*.

Um *RMS* (por exemplo, CondorG [67] e GridFlow [28]) está mais preocupado com o desempenho do sistema (*system-centric*) e não especificamente com uma aplicação. Ele é normalmente centralizado, objetivando gerenciar várias aplicações ao mesmo tempo. Por outro lado, um *middleware* do tipo *UMS* (por exemplo, Nimrod/G [26] e MyGrid [41]) adota uma visão direcionada ao usuário (*user-centric*), gerenciando apenas as aplicações de um único usuário por vez segundo os seus requisitos. Um sistema gerenciador de aplicação *AMS* (GrADS [12], GridWay [89] e EasyGrid [19, 119]) possui a principal característica de transformar as aplicações em versões *system aware*, tornando-as ciente dos recursos necessários, podendo assim, se auto ajustar às mudanças que ocorrem na grade computacional, sem a interferência do usuário. Outra característica marcante é que, como existe um *AMS* responsável por cada aplicação, este pode ser embutido na aplicação (como por exemplo, o *EasyGrid*), ao invés de instalados nos recursos da grade, permitindo, assim, maior portabilidade.

Uma explicação mais detalhada sobre os tipos de sistemas gerenciadores, assim como uma taxonomia que descreve as características mais importantes desses sistemas pode ser vista em [116].

2.2 Aplicações MPI em Ambientes Heterogêneos

De uma maneira geral, dois modelos de programação em paralelo podem ser adotados: memória compartilhada e troca de mensagens [49]. Uma opção é a utilização do

modelo de memória compartilhada que assume que todas as estruturas de dados estão alocadas em um espaço comum que pode ser acessada por qualquer processador. Uma abordagem para facilitar a estruturação do programa no modelo de programação de memória compartilhada é o OpenMP [49, 123].

Devido a motivações tecnológicas, experiência e simplicidade, a maioria dos projetistas de programas paralelos adotou o paradigma de troca de mensagens, onde várias tarefas podem ser definidas, cada uma associada aos seus dados locais, e as quais interagem entre si através da troca de mensagens. Além disso, visto que de uma maneira geral os programadores vêem os computadores paralelos como um conjunto fixo de recursos dedicados, homogêneos, sem falhas, interconectados através de uma rede rápida, o modelo padrão adotado é a alocação de um único processo por recurso durante toda a execução da aplicação. A maioria dos programas implementados utilizando a biblioteca MPI (*Message Passing Interface (MPI)* [111]), por exemplo, adota esse modelo [61, 81, 94, 124, 129].

Este modelo de execução, apesar de simples, é bastante eficiente quando o ambiente disponível para os usuários e programadores é um único *cluster* de computadores. Por outro lado, de uma maneira geral, paralelizar um programa é um problema bastante complexo até mesmo quando esse modelo simples de execução é adotado.

Implementações da biblioteca MPI para grades estão atualmente limitadas às versões compatíveis com o Globus [63], como por exemplo, MPICH-G2 [113], LAM/MPI [101] e MPICH-GF [155]. Embora essas versões facilitem a execução de programas MPI existentes nas grades computacionais sem modificações no código (recompilar é necessário), elas não têm capacidade de lidar com um ambiente dinâmico, heterogêneo e compartilhado eficientemente. Logo, fazer com que programas paralelos MPI possam ser executados eficientemente em uma grade computacional é um desafio.

Além das características das grades que tornam a sua utilização bem mais complexa, programas paralelos desenvolvidos em MPI apresentam algumas limitações. Por exemplo, na implementação da versão LAM/MPI [101], existe um número máximo de processos que podem ser criados simultaneamente em um mesmo recurso. Essa quantidade de processos é muito maior do que a utilizada pela maioria dos programas MPI existentes, que executam apenas um processo por máquina. Porém, esses programas foram projetados para serem executados em um número fixo de recursos homogêneos e estáveis. Desse modo, cada processo (tarefa), normalmente, executa a mesma quantidade de trabalho em cada recurso [49, 59, 61].

Para ambientes como as grades, a utilização de apenas um processo por recurso pode não ser a melhor opção. Devido a sua natureza dinâmica, onde recursos podem se tornar

disponíveis ou não a qualquer momento, apenas um processo por recurso limita a execução da aplicação, caso algum novo recurso entre na grade. Além disso, visto que as grades estão mais sujeitas a falhas do que os ambientes tradicionais, mecanismos sofisticados e custosos, como por exemplo *checkpointing*, podem ser necessários. Por último, como a capacidade de processamento dos recursos de uma grade varia durante a execução da aplicação devido à heterogeneidade e compartilhamento de seus recursos, é necessário algum tipo de balanceamento de carga (escalonamento) dinâmico para um melhor aproveitamento do ambiente disponível para execução. Ao utilizar apenas um processo por recurso, esse balanceamento normalmente é programado no código do usuário, aumentando sua complexidade. Além disso, caso uma máquina fique sobrecarregada, para mudar o processo para uma outra máquina mais rápida é necessário uma migração de código (processo), que é um mecanismo custoso.

Com o intuito de explorar o grau de paralelismo inerente da solução do problema e aproveitar melhor as características das grades computacionais, uma abordagem alternativa é o desenvolvimento de programas MPI independentemente do número de recursos disponíveis. Sob a supervisão de um gerenciador de processos da aplicação, esta abordagem alternativa oferece uma maior flexibilidade na alocação de tarefas nos recursos que entram na grade ou recursos que se tornam ociosos ao longo da execução da aplicação. Ainda, em caso de falha, apenas processos atingidos terão que ser re-executados. Nesse contexto, o balanceamento de carga (escalonamento dinâmico) pode ser feito através da distribuição dos processos entre os recursos pelo gerenciador. Esse modelo de execução alternativo será apresentado no Capítulo 3. Algumas características desse novo modelo já foram implementadas em outras ferramentas, que serão descritas a seguir, e serviram como motivação para este trabalho.

Mais recentemente, uma extensão da biblioteca MPICH-G2 para grades, chamada MGF [79], implementa *daemons* de comunicação que permitem a conexão entre diferentes tipos de máquinas. Desse modo, um único programa MPI pode executar transparentemente em diferentes tipos de máquina (*cluster*, supercomputadores, entre outros), mesmo que os nós dessas máquinas façam parte de uma rede privada onde apenas um nó possua um endereço IP público. Porém, o modelo de execução continua sendo o tradicional um processo por recurso e nenhuma solução para a heterogeneidade e compartilhamento dos recursos foi acrescentada.

Uma abordagem interessante para permitir que os programadores desenvolvam programas paralelos capazes de executar eficientemente em redes heterogêneas de computadores (porém, ainda não está disponível para ambientes grades) é disponibilizada pelo HeteroMPI (*Heterogeneous MPI*) [102]. Esta extensão da biblioteca MPI fornece uma lin-

guagem de definição de modelo para que o programador especifique um modelo de desempenho dos recursos, ou seja, o número de processos da aplicação, o volume de computação a ser realizado por cada um dos processos, o volume de dados a ser transmitido entre os processos e a ordem de execução da computação e da comunicação para cada um dos processos. Através desse modelo é possível especificar as principais características da aplicação paralela que influenciam no seu desempenho. Além disso, o HeteroMPI disponibiliza novas operações (para estimar a velocidade dos recursos, conseguir uma previsão do tempo total de execução da aplicação, entre outras) para permitir que o programador desenvolva programas paralelos capazes de executar eficientemente em ambientes heterogêneos. A principal desvantagem dessa abordagem é deixar a cargo do programador o tratamento da heterogeneidade da aplicação e do ambiente, tarefa essa, que por ser muito complexa, limita o número de programadores aptos a usar essa ferramenta.

Também para ambientes heterogêneos, Open MPI [70, 75] é uma nova implementação MPI (baseada nos códigos das implementações LAM/MPI [101], LA-MPI [74] e FT-MPI [54]) totalmente compatível com as especificações MPI-1.2 [59] e MPI-2 [58]. O principal objetivo é dar suporte para heterogeneidade nos processadores, na rede, no ambiente de execução dentro de uma única implementação MPI. A execução de programas desenvolvidos em Open MPI requer a utilização de um ambiente de execução, chamado de openRTE [31], que forneça suporte (tais como, comunicação entre processos, descoberta de recursos, alocação e criação de processos) para as aplicações de alto desempenho nos ambientes heterogêneos. Embora essa implementação suporte a heterogeneidade dos ambientes computacionais atuais, como por exemplo, o uso efetivo de múltiplos canais de comunicação entre um par de processadores, é necessário que o programador implemente mecanismos para melhorar o desempenho da aplicação em ambientes dinâmicos, compartilhados e sujeitos a falhas (por exemplo, balanceamento de carga e tolerância a falhas), o que para uma grande parte dos programadores de aplicações paralelas é uma tarefa difícil.

Diferentemente das abordagens anteriores que se baseiam na execução de um processo por processador, Charm++ [92] é uma linguagem de programação baseada no C++, na qual programas são decompostos em uma grande quantidade de objetos que cooperam entre si através da troca de mensagens. Estes objetos (a quantidade de objetos é normalmente muito maior que a quantidade de processadores) são mapeados para os processadores disponíveis através de um sistema de execução que transparentemente suporta estratégias de balanceamento de carga e tolerância a falhas utilizando mecanismos de *checkpointing* e migração de objetos. No entanto, sua utilização para ambientes grades ainda não se encontra disponível para os usuários de uma maneira geral. Além disso, Charm++ é um ambiente centralizado, de maneira que não é possível definir políticas distintas para os diferentes

sites de uma grade, ou para seus variados recursos. Devido à popularidade do MPI, os pesquisadores desenvolveram uma implementação, chamada de Adaptive MPI (AMPI) [88], baseada no Charm++, onde os processos MPI são implementados como sendo objetos migráveis (ou threads), que são então gerenciados pelo sistema de execução do Charm++. Converter um programa MPI em AMPI pode requerer algumas modificações.

Alguns benefícios da criação dinâmica de processos na execução de aplicações MPI em ambientes heterogêneos já foram investigados. Em [32], um mecanismo simples de escalonamento é utilizado em aplicações desenvolvidas em MPI-2 com criação dinâmica de processos. Experimentos utilizando programas para calcular a série de fibonacci, números primos e o problema das N-Rainhas alcançaram bom desempenho em termos de tempo de execução e balanceamento de carga. Uma evolução desse mecanismo, apresentada em [126], mostrou um algoritmo de escalonamento *on-line* que utiliza a criação dinâmica de processos para conseguir um bom balanceamento de carga para programas implementados em MPI-2. Para aumentar o desempenho de aplicações que utilizam a estratégia de divisão e conquista foi utilizada a técnica de *work stealing* que permite que processadores ociosos roubem tarefas de outros processadores mais sobrecarregados. Devido a características do MPI-2, o uso da técnica de *work stealing* não é trivial. Logo, o trabalho apresentou uma variação desta técnica chamada de *work stealing* hierárquico, assim como seus benefícios através da execução da aplicação N-Rainhas em ambientes reais de execução.

2.3 Representação de uma Aplicação Paralela

O principal objetivo desse trabalho é a execução eficiente de programas MPI nas grades computacionais. De acordo com a discussão nas duas seções anteriores, devido à complexidade de se utilizar uma grade, os programas são executados neste ambiente através de sistemas gerenciadores (*middleware*). Além disso, foi falado também que o modelo de execução do MPI, onde é criado um processo (longo) por recurso que executa durante toda duração da aplicação, não atende às características dos ambientes grades.

O objetivo desta seção é dar uma visão geral sobre diferentes representações de um programa paralelo, mostrando suas vantagens e desvantagens e, baseado nesse estudo, propor um modelo eficaz para representar programas MPI a serem executados nas grades. Um bom modelo da aplicação deve ser capaz de representar as informações cruciais conhecidas sobre um programa paralelo (como por exemplo, custo das tarefas e relação de precedência) e ao mesmo tempo ignorar detalhes irrelevantes que dificultam a sua representação.

Normalmente, ferramentas para paralelizar aplicações [105, 145] e algoritmos para escalonamento e mapeamento [99, 100, 125] utilizam um modelo teórico de grafos para

representar um programa paralelo. A utilização desses modelos se deve a sua capacidade de representar um programa paralelo e também, devido às propriedades estudadas em teoria dos grafos (por exemplo, o caminho crítico e particionamento de um grafo). Nesses modelos, um programa paralelo é composto por duas partes: computação e comunicação. Os vértices do grafo representam a computação, enquanto as arestas representam as comunicações.

Em [139] é proposta uma classificação dos modelos teóricos de grafos de acordo com os seguintes aspectos: computação paralela, arquitetura paralela e ferramentas disponíveis para os modelos. A computação paralela define as características do programa paralelo que são possíveis de representar, como por exemplo, a capacidade do modelo de representar ciclos (*loops*) explicitamente. A segunda característica define quais os tipos de arquiteturas paralelas suportadas pelo modelo (SIMD e MIMD, por exemplo). Por último, é feita uma análise classificatória das ferramentas existentes para cada modelo, como por exemplo, algoritmos de análise de dependências, de mapeamento e de escalonamento de tarefas. A seguir, uma breve descrição de cada um dos modelos apresentados em [139] é apresentada.

Uma forma comum de representar um programa paralelo é através de um Grafo Acíclico Direcionado (GAD) [84, 140], como pode ser visto na Figura 2.1(a). Neste modelo, os nós ou vértices do grafo representam as tarefas da aplicação e as arestas representam a relação de precedência entre as tarefas, assim como também as trocas de mensagens que ocorrem entre elas. O ponto fraco deste modelo é a incapacidade de representar laços de repetição explicitamente, onde, por exemplo, um laço inteiro é colocado em um único nó. Se o número de repetições do laço for conhecido previamente, em tempo de compilação, então uma alternativa seria a representação de cada iteração do ciclo através de um sub-grafo. Porém, o tamanho do sub-grafo aumenta de acordo com o número de repetições do laço e ainda, na maioria das vezes o número de iterações só é conhecido durante a execução do programa.

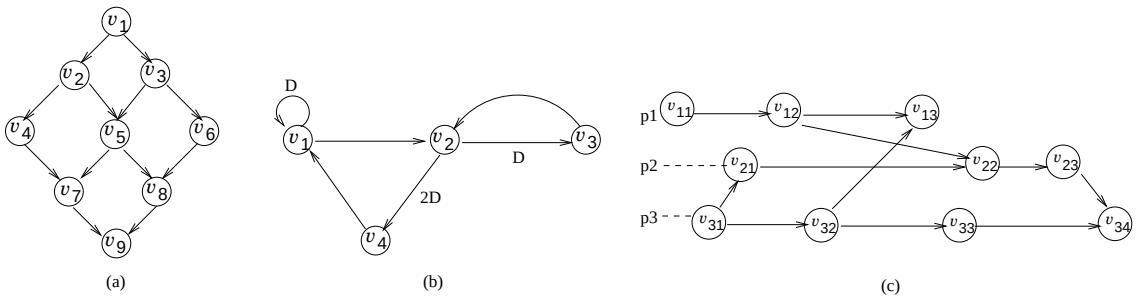

Figura 2.1: (a) GAD (b) ITG (c) TCG

Uma outra modelagem de um programa paralelo é através do Grafo de Fluxo de Dados [34] ou Grafo de Tarefas Iterativas (*Iterative Task Graph - ITG*) [157], usado para modelar fluxo de dados, como mostra a Figura 2.1(b). Apesar de também ser direcionado,

esse modelo permite a representação explícita de ciclos (descrevendo assim, uma forma mais compacta de um programa paralelo). Para especificar o tempo de execução de um ciclo (ou número de ciclos executados), um peso é associado a uma dada aresta, apesar de que nem sempre é possível conhecer este valor em tempo de compilação. A principal vantagem desse modelo é sua capacidade de representar ciclos explicitamente, porém, para uma heurística escalonar eficientemente um programa modelado como ITG é necessário utilizar técnicas como *retiming* [34] e *unfolding* [157], tornando estas heurísticas mais complexas. Por ser um modelo que foi pouco pesquisado e utilizado (principalmente se comparado com o GAD), uma outra desvantagem é a pouca quantidade de heurísticas de escalonamento e mapeamento baseada neste modelo.

O Grafo de Comunicação Temporal (*Temporal Communication Graph - TCG*) [104], baseado no diagrama espaço-tempo introduzido por Lamport, é também um grafo acíclico direcionado que é orientado de acordo com fases e processos (Figura 2.1(c)). O TCG foi projetado para modelar computação paralela definida como um conjunto de processos paralelos que se comunicam através de troca de mensagens. Cada processo pode ser visto como um sequência de eventos atômicos, onde cada evento só pode ser uma computação ou comunicação. Um nó do grafo v_{ij}, associado a um evento de computação de um processo executando no processo p_i, tem pelo menos uma aresta apontando para o próximo nó associado ao mesmo processo p_i (dependência interna ao processador) e pode ter também uma aresta apontando para um nó que esteja associado a um outro processo p_j, que representa as respectivas comunicações entre processos. Um peso associado a uma aresta entre nós de processos distintos reflete o custo de comunicação entre processos. O custo de comunicação entre nós associados a um mesmo processo é considerado desprezível. Uma computação paralela modelada através do TCG é dividida em processos paralelos, cada um sendo desmembrado em um conjunto de eventos sequenciais. Se por um lado, essa perspectiva orientada a processos limita a flexibilidade do modelo (pois limita o paralelismo ao número de processos), por outro lado ela é mais intuitiva para os programadores, onde um programa paralelo alterna fases de computação com fases de comunicação.

2.3.1 Gerenciamento e Geração de Programas representados por Grafos

É interessante mencionar um conjunto de ferramentas para geração, mapeamento, escalonamento e visualização, baseadas em um dos três modelos de grafos apresentados. Comumente, a especificação da representação de um programa paralelo através de um grafo é realizada utilizando uma ferramenta (ou linguagem) própria para criar ou descrever o programa. De uma maneira geral, a principal desvantagem dessa abordagem é que o usuário possui a responsabilidade de definir todas as informações sobre o programa paralelo (grafo,

pesos das tarefas e programa, por exemplo), informações estas que nem sempre são simples de se obter (por exemplo, cientistas que não estão aptos a fornecer tais informações).

A ferramenta PYRROS [158] possui uma linguagem de descrição de grafo através da qual um usuário constrói um GAD baseado em um dado programa, ou seja, o usuário tem que identificar as tarefas do programa, o peso de cada tarefa, as dependências de cada tarefa e o custo de comunicação. Já a linguagem LaRCS [104] é usada no ambiente de programação OREGAMI [105] para descrever programas usando o modelo TCG, porém, assim como na ferramenta PYRROS, é responsabilidade do usuário gerar o programa e sua representação (o grafo correspondente). Já, o ambiente para programação paralela CASCH [3], possui várias ferramentas para ajudar a um usuário construir um programa paralelo. Entretanto, além do usuário ter que saber usar o ambiente, o código gerado é específico para poucas plataformas.

Um outro exemplo mais recente é o gerenciador para grafos acíclicos direcionados DAGMan [145], que funciona como um escalonador para o Condor [67]. O DAGMan recebe um grafo (GAD) como entrada e repassa os nós (*jobs*) para o Condor seguindo a relação de precedência imposta pelo GAD. Existe também o sistema Chimera [66], que é um sistema para geração e acompanhamento de dados virtuais. As entradas no sistema Chimera são descritas em uma linguagem chamada *Virtual Data Language* (VDL) que consegue expressar as dependências entre as tarefas através de GADs.

Diferentemente das ferramentas anteriores, VAMPIR [114, 150] fornece um ambiente gráfico, baseado em um arquivo (*trace*) gerado a partir da execução real do programa, para analisar o desempenho de programas paralelos. Através da ferramenta VAMPIR é possível analisar um programa paralelo MPI para identificar seu comportamento e consequentemente, otimizar seu desempenho.

Mais específicos, os dois trabalhos descritos a seguir apresentam mecanismos para geração de um Grafo de Fluxo de Dados a partir de um programa MPI. Em [134], é apresentado um algoritmo que recebe como entrada um programa MPI e produz como saída um Grafo de Fluxo de Dados que representa esse programa. O estudo foi baseado em um conjunto de programas MPI, onde as linhas de código relacionadas a comunicação MPI representam, na maioria dos casos, menos de 1,5% do total de linhas de código. Além disso, para simplificar, esses programas não possuíam funções MPI contendo valores que só são conhecidos durante a execução do programa MPI (por exemplo, MPI_ANY_SOURCE [111]). Uma abordagem dinâmica é apresentada em [136], onde um Grafo de Fluxo de Dados é criado a partir de um programa MPI, usando o mecanismo de sobrecarga (*overload*) das funções primitivas de comunicação. Para que o Grafo de Fluxo de Dados seja gerado, basta executar a aplicação MPI compilada com uma biblioteca específica que intercepta

as chamadas das funções primitivas de comunicação. Porém, o estudo é limitado a apenas uma aplicação MPI. Apesar destes dois trabalhos mostrarem como é possível gerar uma representação de grafo para um programa MPI, devido a dificuldade do problema, os estudos se limitam a um conjunto restrito de programas MPI que representam apenas uma pequena parte das aplicações MPI existentes.

2.4 Modelos Arquiteturais

Para que programas paralelos eficientes sejam desenvolvidos, é necessário que este capture as características da arquitetura alvo que influenciam o desempenho da aplicação (características essas que devido aos avanços tecnológicos podem mudar). Um modelo arquitetural define quais são as características dos recursos computacionais e da rede de interconexão para um determinado tipo de máquina paralela. Com o objetivo de estabelecer um modelo de computação paralela padrão, vários trabalhos foram propostos. A seguir, um breve resumo dos principais modelos é apresentado.

Um dos primeiros modelos de computação paralela proposto foi o modelo PRAM (*Parallelism in Random Access Machine*) [57] em 1978. O principal objetivo deste modelo é ser bem simples e abstrato, facilitando o estudo, desenvolvimento e análise da complexidade de algoritmos paralelos. Neste modelo, o número de processadores é ilimitado e estes operam de maneira sincronizada. A capacidade de processamento é igual para todos os recursos computacionais e a comunicação entre os processadores é realizada através de uma memória global compartilhada. Assume-se que não existe custo associado tanto à comunicação quanto à sincronização de processos.

Com o passar do tempo, com o objetivo de maximizar o desempenho das aplicações paralelas e desenvolver algoritmos de escalonamento de tarefas mais eficientes, os pesquisadores sentiram necessidade de modelar características encontradas em máquinas reais que influenciavam na execução de aplicações paralelas. Desse modo, tanto extensões do modelo PRAM como também outros modelos foram sendo propostos e estudados. Por exemplo, para representar eficientemente as máquinas paralelas com memória distribuída, foi proposto o modelo de latência [125] que utiliza um único parâmetro para representar o custo da transferência de uma unidade de dado entre dois processadores quaisquer, sendo a execução da aplicação realizada de forma assíncrona. Nesse mesmo ano, foi proposto o modelo Bulk-Synchronous Parallel (BSP) [149] que assume que os programas paralelos são executados em uma sequência de *superpassos* paralelos, onde cada *superpasso* é composto de três fases. Na primeira fase cada processador executa o seu código, na segunda fase é realizada a comunicação entre os processadores que necessitam trocar dados e finalmente, na

última fase é realizada a sincronização entre os processadores para que um novo *superpasso* inicie. O modelo HBSP [153] é apenas uma extensão do modelo BSP, que permite modelar processadores heterogêneos. Também alternando fases de computação e comunicação, o modelo *Coarse Grained Multicomputer* (CGM) [7] acrescenta uma restrição, onde a fase de computação tem que ser maior do que a fase de comunicação.

Com o advento de *clusters* de estações de trabalhos, o modelo LogP foi proposto em [42]. Diferentemente do modelo BSP onde todos os processadores se comunicam durante uma mesma fase, no modelo LogP cada processador realiza sua comunicação de forma assíncrona. Além disso, o modelo LogP utiliza parâmetros que capturam os custos de comunicação e, diferentemente do modelo de latência, comunicação e computação não se sobrepõem totalmente. Os parâmetros utilizados no modelo são os seguintes: a latência (atraso) de comunicação no envio de uma mensagem considerando um número pequeno de *bytes*; a sobrecarga (*overhead*) de envio e recebimento de uma mensagem, que é o tempo que o processador gasta preparando uma mensagem para ser enviada ou recebida, tempo este no qual o processador não pode realizar nenhuma outra tarefa; *gap*, intervalo de tempo mínimo entre envios ou recebimentos consecutivos em um mesmo processador; e por último, o número de processadores disponíveis. Com o objetivo de melhorar a precisão do modelo LogP no envio de mensagens longas e sincronização entre processos, duas abordagens foram propostas. Em [5], foi proposto o modelo LogGP como uma extensão do modelo LogP para trabalhar com mensagens longas. Neste modelo, o parâmetro G (que representa o tempo necessário para o envio de cada *byte* da mensagem) é utilizado no envio de mensagens maiores que um tamanho pré-determinado. Uma outra extensão do modelo LogP, chamada LogGPS [91], utiliza o parâmetro S com o objetivo de tratar a sincronização entre dois processos no envio de mensagens longas (em [4]).

Mais recentemente, foi proposto o modelo HLogP [110] para escalonamento de aplicações MPI em grades computacionais. Um aspecto importante de um modelo arquitetural é como conseguir informações precisas sobre o estado dos recursos. Desse modo, a principal contribuição de [110] foi desenvolver um modelador para ser utilizado em ambientes grades [15]. Através desse modelador é possível descobrir a capacidade de processamento dos recursos disponíveis em uma grade, uma vez que esse modelo assume processadores heterogêneos. Além disso, o modelador também retorna uma estimativa de quanto tempo uma mensagem demora a ser entregue entre os recursos da grade, identificando separadamente latência e sobrecargas, que podem variar dependendo do tamanho da mensagem.

Em [22] são apresentados dois índices com o objetivo de medir a capacidade de sistemas heterogêneos. A principal característica desses índices é assumir que cada máquina é formada por vários recursos (por exemplo, CPU, memória, disco e rede), e com isso o

valor do índice depende das características (e da utilização) desses recursos. A partir disso, os índices representam as características dos recursos utilizando uma métrica euclidiana que utiliza vetores para obter as características dos recursos de uma máquina. Esses índices foram validados através de simulações para aplicações *bag-of-tasks* e mostraram um melhor desempenho quando comparado com índices específicos para cada tipo de recurso. Com o objetivo de conseguir um ambiente para avaliar índices de desempenho para serem utilizados no escalonamento de processos, o artigo descrito em [21] propõe um modelo de simulação, assim como um simulador baseado neste modelo. Vários experimentos, utilizando diferentes índices, foram executados para validar o modelo.

Já, Casanova em [29], discute três aspectos relevantes ao se especificar modelos arquiteturais para grades mais próximos da realidade: a sobrecarga de comunicação (*overhead*), o compartilhamento de largura de banda e a topologia da rede. A sobrecarga é considerada para fornecer um modelo arquitetural mais real, para que os escalonadores de tarefas possam produzir resultados mais eficazes. Com relação ao compartilhamento de largura de banda, a comunicação entre pares de processadores distintos normalmente causa interferência nas diferentes aplicações que são executadas simultaneamente, pois estas compartilham o mesmo recurso de rede (por exemplo, Ethernet ou internet). Este comportamento não acontece nos *clusters* de computadores, onde a comunicação é feita através de um *switch*. Com relação à topologia da rede, o artigo sugere que canais de comunicação locais (dentro de um *site*) devem ser diferenciados de canais de comunicação entre *sites* (que ligam diferente regiões), de modo que as diferentes características de cada tipo de canal possam ser modeladas.

É importante destacar que, apesar de não fazer parte de sua descrição, todos os modelos apresentados assumem que o modelo de execução utilizado é o tradicional um processo por processador. Como consequência, todos esses modelos assumem que o custo de comunicação dentro de um mesmo processador é desprezível. Além disso, até o momento, não existe um modelo padrão para representação de ambientes grades.

2.5 Heurísticas de Escalonamento

Nesta seção, as principais heurísticas de escalonamento de tarefas utilizadas nos gerenciadores de recursos ou aplicações para grades computacionais existentes são descritas brevemente. Apesar dos gerenciadores que serão mostrados serem capazes de realizar outras funções além do escalonamento da aplicação, apenas as principais características sobre o escalonador de tarefas serão abordadas.

O MyGrid [37, 41] é um gerenciador de aplicações do tipo *bag-of-tasks* (que só con-

tém tarefas independentes) para grades. Devido à dificuldade de se obter informações precisas sobre as tarefas das aplicações e sobre os recursos da grade, os desenvolvedores do MyGrid escolheram a heurística Workqueue [83] para escalonar dinamicamente as tarefas da aplicação a ser executada em uma grade computacional. Apesar dos autores dizerem que essa heurística não necessita de nenhuma informação, duas informações são, na verdade, utilizadas: como o MyGrid só executa aplicações do tipo *bag-of-tasks* implicitamente esta informação sobre a aplicação é disponibilizada para o *middleware* (informação sobre a aplicação); e o aviso de término de cada tarefa (informação sobre estado do recurso). Visto que o desempenho desse algoritmo é muito inferior aos de heurísticas que utilizam outras informações sobre a aplicação e os recursos, a heurística Workqueue com replicação (WQR) foi desenvolvida em [44]. Entretanto, a replicação de tarefas aplicada em WQR é razoavelmente gulosa: no momento em que não existem mais tarefas da aplicação a serem executadas, o algoritmo simplesmente replica as tarefas que ainda estão executando nos recursos ociosos. Logo, existe um desperdício dos recursos através da replicação de tarefas para se obter um bom desempenho na execução de aplicações *bag-of-tasks*. Sendo o compartilhamento uma das principais características de uma grade, essa utilização gulosa de recursos prejudica o desempenho das outras aplicações que estiverem executando no ambiente grade, sem oferecer uma garantia de que a aplicação vai ser executada mais rapidamente.

O Sistema Gerenciador de Recursos (RMS) Condor-G [67] é um ambiente de computação de alta vazão que consegue controlar uma grande coleção de máquinas interligadas através de redes distintas. Ele utiliza uma mecanismo chamado *ClassAd* para disponibilizar (oferecer) e pedir recursos que satisfaçam os requisitos (por exemplo, quantidade de memória). O principal objetivo do Condor-G é ter uma alta-vazão de *jobs* (tarefas) sem se preocupar especificamente com a execução eficiente de uma determinada aplicação.

Já, o Nimrod/G [2, 26] possui um gerenciador (*broker*) de recursos centralizado específico para controlar aplicações do tipo *parameter sweep* (um tipo de aplicação composta apenas por tarefas independentes que será explicada na Seção 3.2). As tarefas são alocadas nos recursos utilizando um modelo computacional baseado em economia da grade [27]. Além disso, o escalonamento deve respeitar o tempo máximo que cada aplicação pode terminar (*deadline*).

AppLeS [14] utiliza um algoritmo adaptativo baseado em eventos de escalonamento. O gerenciador AppLeS permite a utilização de várias heurísticas de escalonamento, porém ele está disponível apenas para aplicações do tipo *parameter sweep* uma vez que esse projeto foi terminado dando origem ao *middleware* GrADS. Através de experimentos, a heurística XSufferage foi a que obteve o melhor desempenho quando comparada com as

heurísticas MinMin, MaxMin e Sufferage, para o tipo de aplicação citada. O *middleware* GrADS [12, 39] está preparado para escalonar *jobs* de aplicações do tipo *Workflow* (uma aplicação composta por uma série de componentes que têm que ser executados em uma ordem parcial determinada por dependências de dados e controle). O objetivo do escalonador de tarefas do GrADS é minimizar o tempo de fim da aplicação ou *Workflow* (*makespan*). Para isso, é construído um modelo dos recursos da grade usando serviços específicos que coletam essas informações como MDS [56] e NWS [154]. A partir desse modelo e do modelo da aplicação, o escalonador do GrADS constrói uma matriz de desempenho que indica quanto custa executar cada tarefa em cada recurso disponível de acordo com o custo de execução da tarefa e a transferência dos dados necessários a sua execução. Por último, três heurísticas de escalonamento (Min-Min, Max-Min e Sufferage) são executadas e a que obtiver o mapeamento das tarefas com o melhor (menor) *makespan* é utilizada.

O gerenciador de recursos GridFlow [28] também oferece um serviço de escalonamento para aplicações do tipo *Workflow*. Porém, para realizar o escalonamento, que é dividido em dois níveis (global e local), várias ferramentas sofisticadas são necessárias. O escalonador global, após receber um pedido do portal para escalonar uma aplicação, simula a execução dessa aplicação, dividida em um conjunto de *sub-workflows*, para obter um escalonamento eficaz [28]. A aplicação é então alocada em *sites* locais onde os *sub-workflows* são escalonados. Diferentemente do gerenciador global, os gerenciadores locais têm que tratar possíveis conflitos por recursos gerados por tarefas pertencentes a *sub-workflows* distintos, utilizando algoritmos próprios. Em caso de conflito, algumas tarefas podem ser atrasadas devido à utilização do recurso, onde normalmente a prioridade utilizada para decidir que tarefa deve ser executada primeiro é FIFO (*First In First Out*).

Pelo descrito até então, a maioria dos gerenciadores existentes na literatura só são capazes de executar aplicações paralelas onde as tarefas não têm dependência ou poucas dependências (sendo que esta dependência é realizada através de arquivos de entrada e saída). Além disso, nenhum dos gerenciadores está apto para escalonar aplicações paralelas reais representadas através de GADs ou outros modelos de aplicações.

2.6 Resumo

Este capítulo apresentou a principal motivação desse trabalho que é a execução eficiente de aplicações MPI nas grades computacionais. Inicialmente foram apresentadas as principais características das grades que tornam esse ambiente muito mais complexo para execução de programas do que os ambientes para computação de alto desempenho tradicionais. Como consequência, os programas nas grades são executados através de sistemas

gerenciadores que servem de interface para a sua utilização e facilitam a utilização de mecanismos importantes como tolerância a falhas e escalonamento. Em seguida, foi descrito como a maioria das aplicações MPI vem sendo executadas nas grades computacionais, dando ênfase especial ao modelo tradicional de execução adotado que utiliza um processo por processador, onde todos os processos são criados estaticamente no início da execução aplicação. As limitações desse modelo, principalmente em função das características principais das grades, são apresentadas e uma possível solução introduzida.

Para que um sistema gerenciador possa controlar com eficiência a execução de uma aplicação, é preciso que ele conheça bem as características da aplicação e do ambiente de execução. Em geral essas informações são passadas para o gerenciador através do modelo da aplicação e do modelo arquitetural. Desta maneira, neste capítulo também foi apresentado um estudo sobre várias maneiras de se representar programas e ambientes de execução, disponíveis na literatura, destacando as suas principais vantagens e desvantagens. Além disso, neste capítulo também foram identificadas as heurísticas de escalonamento dos principais sistemas gerenciadores para grades existentes, uma vez que para que um programa execute eficientemente um bom escalonamento de suas tarefas é fundamental.

Capítulo 3

Modelo Alternativo para Execução de Aplicações MPI em Ambientes de Grades

Como apresentado no capítulo anterior, as grades computacionais são compostas de recursos heterogêneos, compartilhados, dinâmicos, interligados por diferentes tipos de rede. Todas essas características juntas tornam as grades ambientes de programação extremamente difíceis para os cientistas em geral. Com o objetivo de facilitar o trabalho dos programadores e, principalmente, propiciar o sucesso das grades computacionais, sistemas gerenciadores (*middleware*) vêm sendo desenvolvidos para abstrair essas complexidades [12, 14, 26, 28, 41, 67, 89, 119]. Devido ao comportamento dinâmico tanto das grades como das aplicações, esses sistemas são projetados para gerenciar a execução das aplicações, determinar suas necessidades e decidir a alocação de recursos mais apropriada. Durante a execução, eles gerenciam a aplicação, por exemplo, reescalonando processos para melhorar o desempenho ou reexecutando processos que falharam.

O capítulo anterior também motivou alguns questionamentos, como por exemplo: o modelo de execução padrão adotado pela maioria dos programadores e ferramentas (executar um processo por recurso) atende as características de uma grade; como representar uma aplicação paralela MPI a ser executada neste ambiente; e quais características relevantes de uma grade computacional devem ser capturadas no modelo arquitetural. Neste capítulo, a partir das características dos ambientes grades e das limitações do modelo de execução tradicionalmente adotado, é proposto um modelo alternativo para execução de aplicações paralelas nas grades computacionais, e ainda respostas aos questionamentos apresentados.

3.1 Modelo de Execução Alternativo

Esta seção apresenta as desvantagens e limitações do modelo de execução, adotado pela maioria dos programadores e ferramentas, um processo por recurso. Com o objetivo de

melhor adaptar a execução da aplicação as principais características dos ambientes grades, é proposto um modelo alternativo que executa um processo por tarefa.

3.1.1 Modelo de Execução Tradicional - 1PProc

Em geral, aplicações paralelas de alto desempenho projetadas para executar em *clusters* de computadores são especificadas baseadas na suposição que o ambiente é composto por um conjunto de recursos homogêneos, livres de falhas e interconectados através de uma rede rápida. Como o objetivo é maximizar o desempenho, os recursos são geralmente dedicados à execução de uma aplicação paralela por vez. Essas aplicações MPI, normalmente do tipo SPMD, são tipicamente projetadas para executar processos de longa duração idênticos (criados estaticamente ao se iniciar a aplicação). Nestes casos, cada processador disponível executa apenas um processo durante toda a execução do programa [49, 61], sendo chamado neste trabalho de tese de modelo de execução 1PProc, como mostra a Figura 3.1. Para um melhor entendimento é importante destacar a diferença entre processos e tarefas. Uma tarefa (ou trabalho) é uma unidade de código básica que faz parte de um programa ou pode ser um programa. Um processo por sua vez é uma unidade de execução criada em um processador que pode executar uma ou mais tarefas.

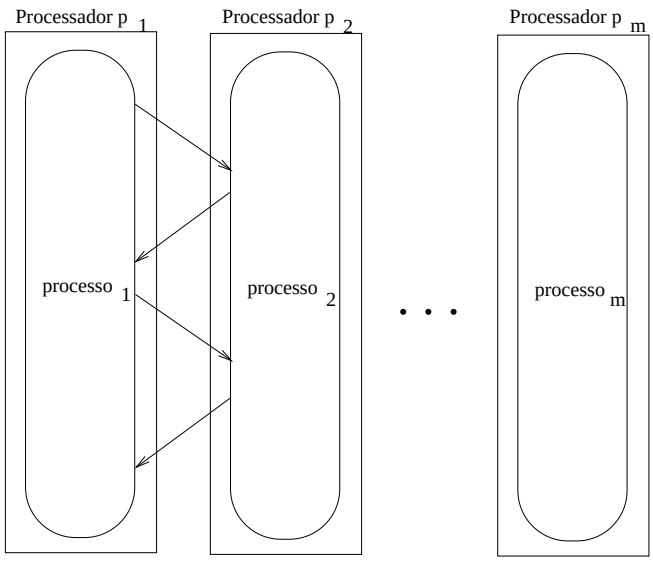

Figura 3.1: modelo de execução 1PProc.

O modelo 1PProc, apesar de simples, é bastante eficiente para ambientes homogêneos, dedicados e interconectados através de uma rede local ou um *switch* rápido. Outra vantagem é que ele simplifica o processo, muitas vezes complexo, de paralelizar uma aplicação. Entretanto, nas grades computacionais, este modelo se torna inapropriado. Uma vez que os recursos podem ser heterogêneos e compartilhados com tarefas locais a carga de trabalho de cada processo irá necessitar de ajustes. Além disso, por ser um ambiente

dinâmico onde recursos podem estar disponíveis ou não a qualquer momento e, ainda, sujeito a falhas o modelo 1PProc não somente acarreta uma execução ineficiente, que limita o número de processos a quantidade de recursos disponíveis no início da aplicação, mas também torna o gerenciamento da execução das aplicações extremamente complexo e custoso em termos de computação e armazenamento. Até o momento, mesmo os programas MPI implementados nas versões capazes de executar programas paralelos em grades computacionais, por exemplo, MPICH-G2 [113], MGF [79] e LAM/MPI [101], são baseados no modelo 1PProc e funcionam apenas como versões estendidas para grades, executando o mesmo código projetado para um ambiente dedicado e homogêneo (*cluster*).

Programas MPI, de uma maneira geral, são desenvolvidos baseados em uma arquitetura fixa (*cluster* disponível) e com isso, é natural que o número de processos criados seja proporcional ao número de processadores disponíveis. Nesses casos, o modelo de execução 1PProc é utilizado. Em [33] é apresentado uma modelagem alternativa para aplicações do tipo mestre/trabalhador, onde o número de tarefas da aplicação deve ser bem maior que o número de processadores disponíveis. Além disso, é feita uma análise do número ideal de trabalhadores considerando aplicações com tarefas de mesmo custo. É importante ressaltar que apesar de [33] considerar que a aplicação é composta de uma quantidade de trabalho muito maior que o número de processadores disponíveis, cada processador executa apenas um processo por recurso durante toda a execução da aplicação. Uma proposta para melhorar o desempenho de aplicações MPI iterativas executando em ambientes heterogêneos é apresentada em [106], onde processos MPI podem ser divididos (*split*) ou unidos (*merged*) com o objetivo de tornar a aplicação maleável e, com isso, conseguir uma execução mais eficiente. Entretanto, é responsabilidade do programador incluir o código específico para que a aplicação se torne maleável e, também, é necessário um sistema operacional especial que implemente as funções para dividir e juntar processos. Além disso, até o momento, as análises disponibilizadas foram realizadas considerando apenas *clusters*, de maneira que ainda não foi avaliado o impacto da maleabilidade nas grades.

3.1.1.1 Execução de mais de um processo por processador

Em um cenário onde o número de processos não se restringe ao número de recursos, como analisado em [119], o número de processos que podem ser criados simultaneamente em um programa implementado em LAM/MPI [101] é bastante limitado. O experimento a seguir apresenta o número máximo de processos que foi possível criar para uma execução em um *cluster* com até 6 recursos (processadores) disponíveis.

Os valores mostrados na Tabela 3.1 representam o número máximo de processos que um programa mestre/trabalhador implementado em LAM/MPI, sem criação dinâmica de

Tabela 3.1: Quantidade máxima de processos que podem ser criados simultaneamente no LAM/MPI considerando o número total de processadores (#processadores)

	quantidade máxima de processos	
#processadores	**TCP**	**lamd**
1	217	246
2	307	492
3	376	738
4	435	984
5	487	1230
6	534	1476

processos, foi capaz de executar. A coluna TCP representa a execução tradicional do MPI, onde todas as tarefas da aplicação são criadas estaticamente e a comunicação entre processos utiliza o protocolo TCP. Já a coluna lamd, representa a execução do MPI, onde os processos também são criados estaticamente, porém a comunicação é feita através de um *daemon* do MPI. Enquanto que na comunicação TCP, cada processo se comunica diretamente com outro processo, na comunicação lamd cada recurso possui um *daemon* que serve de intermediário na entrega das mensagens entre processos. Apesar da comunicação via TCP ser mais rápida, pois não precisa de intermediário, ela necessita de uma conexão entre cada par de processos, o que a torna menos escalável que a comunicação lamd para uma quantidade grande de processos. Pode-se concluir que os limites da Tabela 3.1 são relativamente baixos, principalmente quando se almeja executar aplicações de alto desempenho em grades computacionais, onde se espera que um programa possa ter milhares de tarefas.

Além do limite na quantidade de processos, um programa MPI pode não executar eficientemente devido a política *round robin* tradicional, que cria estaticamente todos os processos da aplicação. Por exemplo, em um programa MPI composto por vários processos dependentes (que se comunicam), processos prontos (que não dependem de outros processos ou que todos os seus antecedentes já tiverem finalizados) executam junto com processos que ainda esperam por alguma mensagem. Essa sobrecarga de execução de processos pendentes degrada bastante o tempo total de execução da aplicação, principalmente se for usada a comunicação via TCP, como pode ser observado no experimento a seguir que utilizou um *cluster* com 4 máquinas.

A Tabela 3.2 apresenta o tempo de execução de aplicações MPI sintéticas, criadas a partir de Grafos Acíclicos Direcionados (GADs) que representam aplicações paralelas reais (conforme serão descritas na Seção 3.2). Além disso, cada tarefa tem o mesmo peso (executa exatamente o mesmo código sobre a mesma quantidade de dados). A coluna Aplicação mostra os três tipos de aplicações que foram executadas: Árvores Binárias Invertidas (InTree), Árvores Binárias (OutTree) e Diamantes (Di). A coluna #Tarefas determina a quantidade total de processos da aplicação. Cada uma das aplicações foi executada de três

Tabela 3.2: Tempo de execução de aplicações paralelas implementadas através do LAM/MPI em 4 máquinas

		tempo de execução (segundos)		
Aplicação	#Tarefas	TCP	lamd	EasyGrid
InTree	15	7,5	6,0	5,3
	127	66,3	40,0	38,3
	511	ne	207,6	138,4
OutTree	15	8,8	6,3	6,3
	127	172,3	41,7	35,1
	511	ne	192,4	132,9
Di	100	243,9	46,8	29,0
	400	2033,6	141,5	116,7
	1024	ne	ne	267,5

maneiras distintas: TCP, lamd e EasyGrid. Nas duas primeiras, os processos são criados estaticamente e a comunicação entre eles utiliza TCP e lamd, respectivamente. Na terceira, EasyGrid, a aplicação foi executada utilizando o sistema gerenciador de aplicações *EasyGrid AMS* (Seção 4.1.1), que utiliza a criação dinâmica de processos e fornece mecanismos de tolerância a falhas e escalonamento de tarefas. Os resultados obtidos indicam que a execução de aplicações paralelas MPI com vários processos por máquina, onde os processos são criados estaticamente é ineficiente e não escalável, principalmente se for utilizada a comunicação TCP, onde processos esperando por mensagens gastam tempo de CPU. Os valores ne da Tabela 3.2 significam que não foi possível executar a aplicação utilizando a criação estática de processos, devido ao limite do número de processos que podem ser criados estaticamente.

Diferentemente, as aplicações executadas através do *EasyGrid AMS* obtiveram um bom desempenho e não sofreram limitações com relação a quantidade de processos executados. Isso se deve ao modelo alternativo de execução adotado pelo *EasyGrid AMS*, assim como a criação dinâmica de processos, como será mostrado na subseção a seguir.

3.1.2 Modelo de Execução Alternativo - 1PTask

Tendo em vista as limitações do modelo 1PProc para ambientes heterogêneos em geral, especialmente para as grades computacionais, esse trabalho de tese propõe um modelo alternativo de execução *um processo por tarefa*, chamado de 1PTask, onde as suas principais características são: ser independente da arquitetura do ambiente de execução (ou seja não associa o número de processos da aplicação ao número de recursos disponíveis) e permitir o desenvolvimento de aplicações paralelas que explorem ao máximo o paralelismo da aplicação, sem se importar com o ambiente de execução disponível.

Uma consequência desse novo modelo é que as aplicações podem ser compostas por

centenas, ou até milhares, de tarefas. No entanto, aplicar este modelo em aplicações MPI com criação estática de processos pode ser inviável (Tabela 3.1) [119]. Além disso, como foi mostrado na Tabela 3.2, quando todos os processos são criados estaticamente no início da aplicação, a concorrência entre vários processos de uma mesma aplicação executando simultaneamente em um mesmo processador pode prejudicar o seu desempenho.

Para que o modelo alternativo possa ser utilizado para a execução de aplicações MPI nas grades computacionais, esse trabalho propõe não só a utilização do modelo 1PTask, mas que as tarefas (processos) de uma aplicação MPI sejam criadas dinamicamente, somente no momento em que forem ser executadas. No modelo alternativo de execução 1PTask cada tarefa da aplicação é executada em um processo distinto, e por sua vez cada processador executará centenas (milhares) de processos, como mostra a Figura 3.2. Dessa maneira, apesar da sobrecarga de gerenciar uma grande quantidade de processos pequenos, esse novo modelo permite que sistemas gerenciadores para grades possam manipular com mais facilidade as tarefas de uma aplicação com o intuito de adaptar sua execução às mudanças ocorridas no ambiente e com isso conseguir uma execução eficiente e robusta.

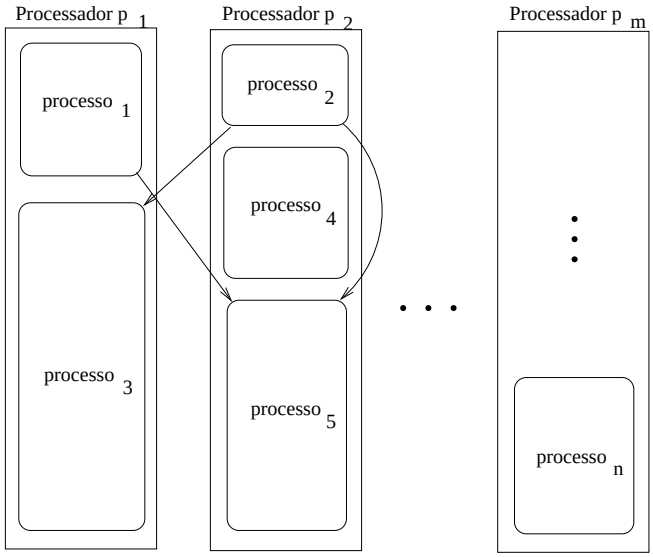

Figura 3.2: modelo de execução 1PTask.

Segundo o Capítulo 21 de [13], que apresenta uma visão geral sobre modelos de programação para grades, um modelo de programação para grades para ter sucesso deve permitir a portabilidade, interoperabilidade, adaptação, tolerância a falhas e um alto desempenho. A seguir são listadas algumas vantagens do modelo 1PTask:

- Por ser uma grade um ambiente dinâmico, onde a disponibilidade dos recursos depende do momento, uma maior quantidade de processos permite uma melhor utilização desses recursos [133];

- Como os recursos da grade são (provavelmente) heterogêneos, esta flexibilidade no número e tamanho das tarefas permite que um escalonador estático as distribua proporcionalmente de acordo com o poder computacional de cada recurso [133];
- Por ser uma grade um ambiente compartilhado e com políticas de acesso distintas, o modelo de execução alternativo 1PTask permite que um escalonador dinâmico possa reescalonar as tarefas da aplicação de acordo com as mudanças que ocorram no ambiente [18, 118];
- Por sua natureza dinâmica e descentralizada, os recursos de uma grade estão mais sujeitos a falhas, o modelo 1PTask permite o uso de mecanismos de recuperação baseado em *log* de mensagens [45], ao invés de mecanismos custosos como o *checkpoint*.

É importante ressaltar que a granularidade de um processo MPI, assim como o modelo de execução adotado, afeta diretamente o projeto de mecanismos de escalonamento de tarefas e tolerância a falhas que devem ser empregados. A seguir os benefícios do modelo 1PTask no projeto desses mecanismos são discutidos.

3.1.2.1 Escalonamento de Tarefas e Tolerância a Falhas no modelo 1PTask

Na maioria dos programas que executam sob o modelo 1PProc, todos os processos são criados estaticamente no início da execução da aplicação. Logo, uma solução natural para que a aplicação paralela se adapte a qualquer variação no ambiente que afete o desempenho é a migração de processos (aplicações mais simples sem dependências, como mestre/trabalhador, geralmente utilizam o mestre para balancear a carga entre os trabalhadores disponíveis). Ou seja, um procedimento custoso que requer que a execução do processo seja interrompida, seu contexto guardado e transferido e, por fim, o processo restaurado em um outro recurso. Um exemplo de migração de processos para aplicações MPI iterativas pode ser visto em [135], onde processos mudam de processador para melhorar o desempenho da aplicação de acordo com uma política pré-determinada. O trabalho citado não avaliou a sobrecarga da migração, porém duas desvantagens dessa abordagem podem ser identificadas: a necessidade de processadores sobressalentes e a possibilidade de um processo ficar migrando de um processador para outro, a cada variação de carga do ambiente. Além disso, o trabalho citado também identificou que o tempo de migração do processo tem que ser menor do que o tempo de cada iteração da aplicação para que a migração proporcione uma execução eficiente da aplicação.

Como no modelo 1PTask as aplicações se caracterizam por uma quantidade grande de processos menores criados dinamicamente, uma abordagem mais simples e leve pode

ser adotada. Ao invés da migração de processos, as tarefas que ainda não foram criadas podem ser alocadas em recursos menos sobrecarregados, de acordo com as decisões de escalonamento do sistema gerenciador. Neste modelo, um escalonador estático é capaz de distribuir as tarefas da aplicação entre os processadores disponíveis conforme o poder computacional oferecido antes da execução da aplicação e um escalonador dinâmico pode modificar essa distribuição (apenas tarefas ainda não criadas) caso ocorram mudanças no ambiente. Deste modo, problemas como a entrada ou saída de um recurso da grade, heterogeneidade dos recursos, disponibilidade dos recursos, compartilhamento de recursos, entre outros podem ser resolvidos sem a necessidade de migração de processos e, mais importante, permitindo uma execução eficiente da aplicação. O escalonador dinâmico só reescalona tarefas que ainda não entraram em execução utilizando um escalonamento próativo [116] diminuindo a sobrecarga no sistema.

Com relação a tolerância a falhas, aplicações compostas por processos de longa duração (modelo 1PProc) devem ser executadas em um ambiente que forneça um mecanismo de *checkpointing* eficiente para se evitar a reexecução de processos. O mecanismo de *checkpointing* requer esquemas custosos e sofisticados para garantir a consistência e permitir que um processo MPI consiga retornar sua execução ao último estado salvo [52]. Entretanto, o tempo gasto para reexecutar processos menores (modelo 1PTask) pode ser curto o suficiente para dispensar o uso de esquemas de *checkpointing*. Neste caso, o uso de um mecanismo de *log* de mensagens é suficiente para permitir a recuperação de falhas em programas executando sob o modelo 1PTask. Como a aplicação é composta de tarefas leves, o mecanismo de *log* de mensagens consiste em reexecutar processos que falharem na mesma máquina, ou caso não consigam, em uma outra máquina [45, 119].

3.1.2.2 Gerenciamento de Aplicações Paralelas através do Modelo 1PTask

Devido à complexidade de se executar aplicações nas grades, esta vem sendo feita através de sistemas gerenciadores que facilitam o acesso e otimizam o seu desempenho. Esse controle é feito baseado nas características da aplicação a ser executada e do ambiente de execução disponível (grade). As características da aplicação são passadas para o gerenciador através do modelo da aplicação, enquanto as características do ambiente através do modelo arquitetural, como mostra a Figura 3.3. Baseado nestes dois modelos e no modelo de execução adotado, um sistema gerenciador é capaz de executar eficientemente uma aplicação paralela fornecendo escalonamento dinâmico e tolerância a falhas.

As duas seções a seguir apresentam, respectivamente, uma proposta de modelo de aplicação e de um modelo arquitetural para programas paralelos executando em ambientes grades, através do modelo de execução alternativo 1PTask, idealizado nesta tese de

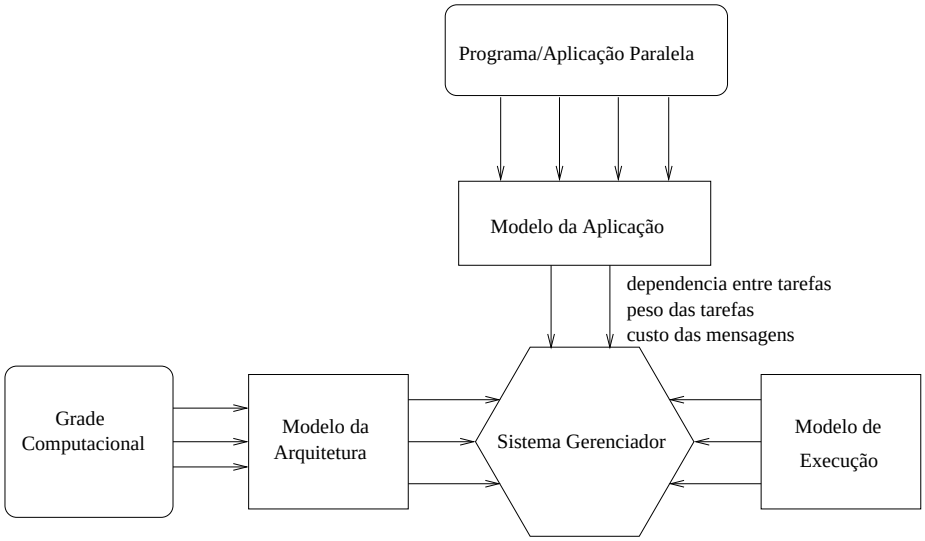

Figura 3.3: Requisitos necessários para um sistema gerenciador controlar a execução de uma aplicação.

doutorado.

3.2 Modelo da Aplicação

Esta seção descreve uma forma de representar aplicações paralelas, de maneira que um sistema gerenciador possa aproveitar as características do modelo de execução alternativo proposto, 1PTask, para melhorar o desempenho da aplicação, através de mecanismos de escalonamento de tarefas e tolerância a falhas. Logo, do ponto de vista do modelo 1PTask uma aplicação paralela é um conjunto de (possivelmente milhares de) tarefas/processos (que independe do número de processadores) que se comunicam através de troca de mensagens. Cada tarefa é definida como uma entidade que inicialmente recebe dados, caso necessite, computa e envia dados, como mostra a Figura 3.4.

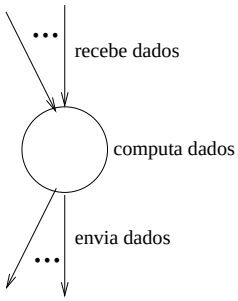

Figura 3.4: Exemplo de tarefa do modelo 1PTask.

Para um controle preciso e eficaz sobre a aplicação, o modelo de aplicação a ser escolhido tem que ser capaz de representar a relação de precedência entre as tarefas, os pesos

das tarefas e os custos das mensagens enviadas entre as tarefas. Através das relações de precedências, é possível saber se uma tarefa está pronta para ser executada ou não, permitindo a execução de programas paralelos em geral. Como no modelo de execução 1PTask, as tarefas são criadas dinamicamente, essa informação é ainda mais importante, pois, determina o momento a partir do qual a tarefa está apta a ser criada e consequentemente executada. O peso das tarefas, assim como o custo das mensagens são informações que podem ajudar os escalonadores de tarefas tomarem decisões mais acertadas na distribuição das tarefas entre os recursos.

As características acima foram determinantes para a escolha do modelo de aplicação GAD adotado neste trabalho. A seguir, é apresentada uma breve discussão sobre modelos de aplicação existentes e outras motivações para a escolha do modelo GAD.

3.2.1 Escolha do Modelo de Aplicação

Um programa pode ser dividido em uma coleção de tarefas e através de uma avaliação sobre suas dependências um sistema de precedência pode ser construído, representando o paralelismo interno do algoritmo [40]. Uma tarefa de um programa paralelo pode ser composta por uma única instrução, por uma sequência de instruções ou até um ciclo de instruções. Além disso, uma tarefa só pode executar quando receber, de suas tarefas predecessoras, todos os dados necessários para sua execução e ao terminar, ela deve enviar os dados necessários para suas tarefas sucessoras.

A Seção 2.3 apresentou três representações distintas de uma aplicação paralela: Grafo Acíclico Direcionado (GAD), Grafo de Fluxo de Dados e Grafo de Comunicação Temporal (TCG). O TCG utiliza uma abordagem orientada a processos que, apesar de ser mais intuitiva para os programadores, limita o paralelismo da aplicação. Nessa abordagem, cada processo é composto por fases de computação e comunicação alternadas e o paralelismo fica limitado ao número de processos. Na programação tradicional MPI, onde em cada processador executa apenas um processo, essa abordagem seria bastante representativa, porém para representar uma aplicação paralela no modelo 1PTask, que não se limita a utilizar apenas um processo por processador, ela não é adequada.

Uma segunda opção seria adotar o modelo de Grafo de Fluxo de Dados, cuja principal diferença em relação ao GAD é sua capacidade de representar ciclos explicitamente. Porém, nem sempre é possível saber o tempo de execução de um ciclo em tempo de compilação. As principais desvantagens desse modelo são a pouca quantidade de heurísticas disponíveis e a alta complexidade das heurísticas existentes, que necessitam utilizar técnicas como *retiming* [35] e *unfolding* [34, 157] para produzir escalonamentos eficientes.

A principal desvantagem do modelo GAD é a incapacidade de representar ciclos explicitamente. Em um GAD, um ciclo tem que ser encapsulado em um único nó (tarefa), ou ser expandido, o que pode não ser viável por não se saber a quantidade de iterações em tempo de compilação. Porém, como será apresentado na Seção 3.3.1, os resultados dos experimentos realizados em ambientes grades reais mostraram que tarefas com granularidade muito pequena sofrem uma maior interferência (sobrecarga) do *middleware*, o que favorece o encapsulamento de ciclos dentro de uma única tarefa. As principais vantagens de se trabalhar com GAD são: a sua capacidade representar programas paralelos em geral, modelando facilmente a relação de dependência entre tarefas, seus custos de execução e de comunicação; o grande número de heurísticas [11, 17, 48, 99, 100, 121, 125, 144, 147] para mapeamento e escalonamento já existentes; e toda teoria sobre GAD já estudada.

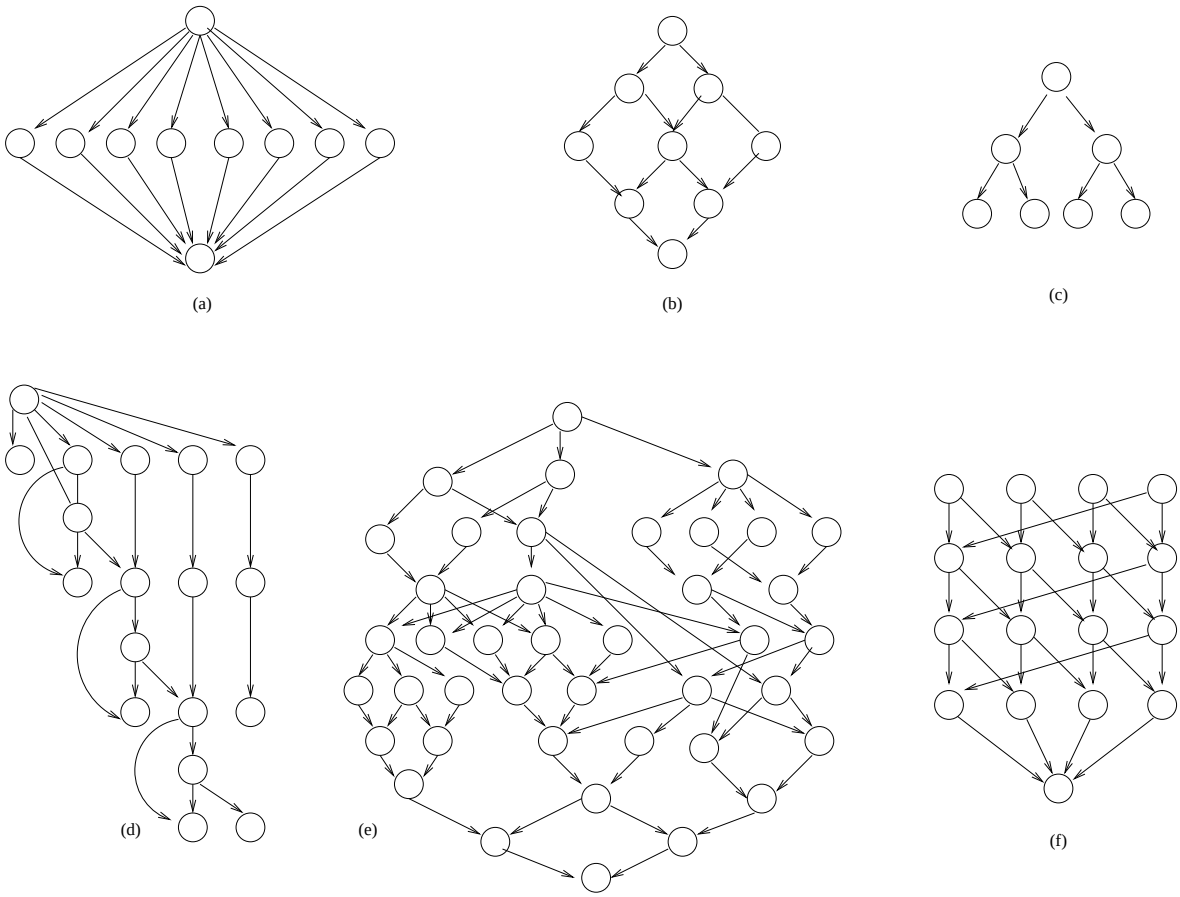

Figura 3.5: Exemplos de GADs que representam aplicações paralelas reais (a) aplicações mestre/trabalhador ou *parameter sweep*; (b) decomposição LU; (c) algoritmos de difusão; (d) eliminação gaussiana; (e) aplicação da física molecular; (f) aplicação N-Corpos

A Figura 3.5 mostra exemplos de aplicações paralelas reais representadas através de GADs. Aplicações sem dependências onde as tarefas necessitam apenas receber os dados iniciais para iniciarem sua execução, como por exemplo mestre/trabalhador [127] e *parameter sweep* [2, 14] são facilmente representadas através de GAD do tipo **fork/join**

(Figura 3.5(a)). Outros programas paralelos, como a computação da sequência comum mais longa ou uma subcomputação na decomposição LU [121], multiplicação de matrizes e *sistolic arrays* [9] podem ser representados através de grafos do tipo **diamante** (Figura 3.5(b)). Já, grafos do tipo **árvore binária** (Figura 3.5(c)) podem ser usados para representar algoritmos de difusão e divisão e conquista, onde os dados vão da raiz até as folhas da árvore. Em outros algoritmos, como por exemplo soma em paralelo [128], os dados vão das folhas para a raiz (caminho inverso), podendo ser representados através de **árvores binárias invertidas**. Um outro exemplo seriam os grafos **irregulares**, em [40] é mostrada uma maneira de como paralelizar a eliminação gaussiana através de um GAD irregular (Figura 3.5(d)) e [96] mostra uma representação de uma aplicação da física molecular através de um GAD irregular de 41 tarefas (Figura 3.5(e)). Como último exemplo, uma representação para a aplicação N-Corpos, que será avaliada neste trabalho, pode ser vista na Figura 3.5(f).

Assim, tendo em vista, principalmente, as características de uma aplicação paralela a ser executada através do modelo 1PTask, foi adotado o modelo de Grafo Acíclico Direcionado para representar uma aplicação paralela.

3.2.2 Modelo Adotado: GAD

Nesta tese, um GAD é denotado por $G = (V, E, \varepsilon, \omega)$, onde $V = \{v_0, v_1, v_2, ..., v_{n-1}\}$ é o conjunto de vértices que representa as tarefas da aplicação e $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ é o conjunto de arcos que representa as relações de precedências entre as tarefas e consequentemente, as comunicações entre estas. A cada nó $v_i \in V$, é associado um peso de execução $\varepsilon(v_i)$ que corresponde à quantidade de trabalho que a tarefa realiza. O custo de comunicação dos dados que são transmitidos do vértice v_i para o vértice v_j é denotado por $\omega(v_i, v_j)$, representando a quantidade de dados a ser transmitida entre as tarefas. A Figura 3.6 apresenta um pequeno exemplo dessa representação.

3.3 Modelo Arquitetural

O modelo arquitetural representa as características do ambiente onde o programa paralelo será executado, como por exemplo a capacidade de processamento dos recursos disponíveis e a velocidade dos canais de comunicação que interconectam esses recursos. Esta seção, apresenta as vantagens e desvantagens dos modelos existentes e, de acordo com as características dos ambientes grades e do modelo de execução 1PTask, propõe um modelo arquitetural, que foi dividido em duas partes: computação, que representa essencialmente as características dos processadores disponíveis na grade; e comunicação,

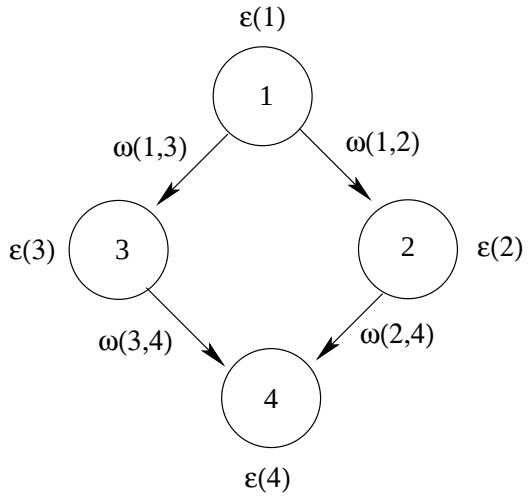

Figura 3.6: Exemplo de GAD, onde as tarefas aparecem com pesos de execução e as mensagens com custos de comunicação.

que especifica as características da comunicação entre duas tarefas distintas.

3.3.1 Modelo de Computação

É essencial definir uma forma precisa de representar o poder computacional dos processadores disponíveis em uma grade computacional, onde a aplicação paralela (especialmente programas MPI) será executada através de um sistema gerenciador que utiliza o modelo 1PTask. Nesse modelo, uma grade computacional é composta pelo conjunto em $P = \{p_0, p_1, p_2, ..., p_{m-1}\}$ de m processadores heterogêneos.

3.3.1.1 Índice de Retardo Computacional

Como visto em [6, 30, 87], quando se considera processadores heterogêneos (também chamado de não relacionados ou não coerentes), o tempo estimado de execução de uma tarefa em um dado processador pode ser representado através de uma matriz $M_{n,m}$, onde n é o número de tarefas e m o número de processadores disponíveis. Através dessa matriz é possível representar a heterogeneidade de cada processador, assim como a afinidade (capacidade de executar mais rapidamente) de cada tarefa com cada processador. Por exemplo, dois processadores de fabricantes distintos com velocidades similares podem apresentar comportamentos bem diferentes com relação as operações aritméticas com números inteiros e operações aritméticas de ponto flutuante.

Para gerar essa matriz é necessário que cada tarefa da aplicação seja executada em cada máquina disponível, de modo que o tempo de execução de cada tarefa em cada recurso seja capturado. Tendo em vista que uma grade computacional é um ambiente dinâmico,

onde recursos (que podem estar executando outros programas) podem estar disponíveis ou não a qualquer momento e que grande parte das aplicações que serão executadas são aplicações de larga escala, a geração dessa matriz pode se tornar inviável, principalmente devido ao tempo necessário para executar todas as tarefas em todos os recursos.

Entretanto, duas características importantes do modelo 1PTask apontam para a não utilização dessa matriz: o grande número de tarefas em consequência da utilização de 1PTask e, principalmente, que cada sistema gerenciador executará uma única aplicação. Ou seja, cada instância é responsável por apenas uma aplicação MPI, sendo natural que essas tarefas possuam características similares, de maneira que o comportamento de cada uma delas seja semelhante (consistente), nos diferentes recursos.

Um modelo computacional mais simplificado, similar ao modelo chamado de processadores uniformes [68, 87] foi adotado neste trabalho. Ao invés da matriz $M_{n,m}$, é utilizado um índice de retardo computacional $csi(p_j)$ para cada processador $p_j \in P$. O índice de retardo computacional de uma máquina é um valor que representa o grau de diferença da capacidade de processamento dos recursos de uma grade computacional. Para que o tempo de execução $te(v_i)$ de cada tarefa $v_i \in V$ da aplicação seja estimado, basta multiplicar o peso da tarefa $\varepsilon(v_i)$ pelo índice $csi(p_j)$, como mostra a Equação 3.1. A principal diferença entre o índice de retardo computacional e o modelo de processadores uniformes é que através do primeiro não só é possível estimar a diferença de processamento entre as máquinas como também utilizá-lo para estimar o tempo de execução de uma tarefa e, consequentemente, o tempo total de execução de uma aplicação.

$$te(v_i, p_j) = \varepsilon(v_i) \times csi(p_j) \tag{3.1}$$

Para capturar o índice $csi(p_j)$ de cada máquina $p_j \in P$ é necessário utilizar um programa modelador (por exemplo, *EasyMod* [15, 110]) que executa uma dada tarefa com as mesmas características das tarefas da aplicação paralela em questão em cada $p_j \in P$ e retorna o $csi(p_j)$ correspondente. Esse valor é proporcional ao poder computacional da máquina e também ao número de processos que estiverem executando nela, caso o processador p_j esteja sendo compartilhado com outro usuário/aplicação. De acordo com a nomenclatura utilizada, quanto menor for este valor melhor o desempenho.

Como o desempenho de uma tarefa varia em função dos tipos de instruções executadas, duas abordagens podem ser adotadas para se obter o valor do índice de retardo csi. No primeiro caso, como mostrado em [110], o programa modelador executa um código padrão, contendo diversos tipos de instruções, que retorna o índice de retardo computacional para as máquinas da grade. Esse código seria formado por instruções que representem os

tipos mais comuns de funções utilizadas nas aplicações científicas implementadas através da biblioteca MPI, como por exemplo, instruções aritméticas inteiras e de ponto flutuante, acesso a matrizes e vetores, entrada/saída, criação e acesso a lista encadeadas, entre outras. De acordo com experimentos realizados em [110], a utilização de um código contendo instruções de soma, soma de matrizes e eliminação gaussiana consegue capturar com precisão a capacidade de processamento dos recursos. Em especial, a utilização deste código é capaz de medir com precisão a diferença de processamento entre máquinas iguais, mas com cargas distintas. Porém, não foram feitos experimentos utilizando aplicações com códigos distintos, de forma que se pudesse avaliar a precisão do valor retornado para diferentes tipos de aplicações. Uma segunda abordagem é disponibilizar um conjunto de instruções de acordo com o tipo de aplicação a ser executada. Deste modo, o programa modelador é executado nos recursos disponíveis, utilizando o código escolhido pelo usuário. Caso o usuário não saiba que tipo de instrução escolher, é utilizado o código padrão da primeira abordagem.

3.3.2 Modelo de Comunicação

Na Seção 2.4 foi apresentado um resumo dos principais modelos arquiteturais existentes na literatura. Em relação a comunicação entre dois processos, o modelo deve não só representar o custo da troca de mensagens, mas também as diferentes características de uma grade computacional, que pode possuir canais com latência de comunicação variadas, principalmente devido à distância geográfica que pode existir entre recursos da grade.

Outro fator importante a ser destacado é que, como as aplicações são executadas nas grades computacionais através de sistemas gerenciadores (*middleware*) que controlam sua execução, o modelo de comunicação deve também representar a interferência deste *middleware* na comunicação, caso exista. Por exemplo, é necessário identificar se a utilização do sistema gerenciador aumenta a latência de comunicação entre dois processos ou a sobrecarga ao se enviar/receber mensagens.

Visto que o modelo de comunicação a ser definido neste trabalho de tese será utilizado baseado no modelo de execução alternativo 1PTask e em conjunto com o modelo de aplicação GAD propostos, onde as tarefas executam assincronamente e se comunicam através da troca de mensagens, um modelo composto por fases alternadas de computação e comunicação (como o modelo BSP) não faz sentido, pois não representa o ambiente de execução proposto neste trabalho.

Sendo um ambiente grade composto de recursos distribuídos geograficamente e redes de diferentes velocidades, a latência de comunicação é um fator essencial a ser modelado,

principalmente considerando que as aplicações MPI se comunicam através da troca de mensagens assincronamente.

Um outro aspecto a ser considerado é a representação ou não da sobrecarga. Como apresentado no modelo LogP [42], antes de enviar ou receber uma mensagem o processador paga uma sobrecarga (não consegue comunicar e executar tarefas simultaneamente). Entretanto, como LogP é baseado no modelo de execução tradicional 1PProc considera-se que a comunicação entre processos localizados no mesmo processador são instantâneas (não tem custo nenhum).

Considerando as características dos principais modelos de comunicação existentes na literatura e do modelo alternativo de execução proposto para ambientes grades, o modelo de comunicação adotado deve ser assíncrono, ou seja, cada processo pode se comunicar com outro processo a qualquer momento (não é necessário uma fase de comunicação). Para uma mensagem ser enviada de um processo para outro existe uma latência de comunicação, que representa o custo de se transferir uma unidade de dado de um processo para outro. Além disso, para enviar ou receber uma mensagem existe uma sobrecarga paga pelo processador, de maneira que a comunicação e computação não podem ser efetuadas simultaneamente.

Entretanto, existem ainda alguns aspectos a serem analisados. Por exemplo, por ser uma grade um ambiente dinâmico e compartilhado onde a quantidade de recursos e poder computacional variam bastante durante a execução da aplicação, um modelo de comunicação mais simplificado poderia ser utilizado levando se em conta que durante a execução ajustes terão que ser feitos para que a aplicação consiga se adaptar as mudanças do ambiente. Outro ponto importante a ser investigado é se a comunicação entre processos localizados em um mesmo processador pode ser considerada sem custos, principalmente em um modelo de execução que considera um processo por tarefa (1PTask). Por último, deve ser verificada a influência dos sistemas gerenciadores de grades no modelo de comunicação, caso ela exista.

No Capítulo 4 será apresentado um sistema gerenciador que será utilizado para validar o modelo alternativo proposto nessa tese e também serão analisadas as questões levantadas sobre o modelo de comunicação.

3.4 Resumo

Este capítulo apresentou o modelo de execução adotado pela maioria das aplicações, inclusive nas grades computacionais, que executa um processo por recurso durante toda execução da aplicação (1PProc). Mais especificamente, foram destacadas as desvantagens

deste modelo tradicional quando utilizado em ambientes heterogêneos, dinâmicos e compartilhados, como as grades. Em seguida, com o objetivo de utilizar ambientes heterogêneos, dinâmicos e compartilhados eficientemente, é proposto o modelo de execução alternativo 1PTask que executa um processo por tarefa. As principais vantagens desse novo modelo são destacadas, especialmente, na criação de mecanismos de escalonamento dinâmico e tolerância a falhas.

Complementando o modelo de execução proposto, este capítulo também apresentou um modelo de aplicação e um modelo arquitetural que têm a finalidade de fornecer as informações necessárias para que um sistema gerenciador possa controlar a execução da aplicação eficientemente, através do modelo de execução alternativo proposto.

Capítulo 4

Validação do Modelo Alternativo para Aplicações bag-of-tasks

O objetivo dos dois próximos capítulos é avaliar e validar o modelo de execução alternativo proposto, através de experimentos em uma grade real. A importância de se utilizar experimentos em um ambiente real, e não simulações, reside no fato de se poder analisar o verdadeiro impacto da solução proposta, inclusive avaliando a sua sobrecarga. Mais ainda, por se tratar de uma nova abordagem, não se sabe da existência de simuladores que conseguissem representar de maneira adequada o modelo de execução proposto. A partir dos resultados apresentados neste trabalho de tese, é possível a criação de tais ambientes de simulação. Desse modo, primeiramente é apresentado o sistema gerenciador de aplicações *EasyGrid AMS*, que foi desenvolvido baseado no modelo de execução 1PTask e que tem como função controlar a execução de aplicações MPI em grades computacionais.

Este capítulo especificamente avalia o modelo 1PTask para aplicações *bag-of-tasks*. A primeira seção deste capítulo apresenta uma descrição geral do *middleware EasyGrid* que é utilizado para validar o modelo alternativo proposto. Logo após, é apresentada uma análise das aplicações *bag-of-tasks*, que se caracterizam por serem composta por tarefas que não se comunicam entre si. Em seguida, é feita uma análise do modelo de computação proposto, apresentando as características mais relevantes para o modelo alternativo. Finalizando, é realizada a validação do modelo de execução alternativo 1PTask para aplicações *bag-of-tasks* através de diversos experimentos em ambientes reais.

4.1 O Projeto *EasyGrid*

Para um melhor entendimento do que foi feito neste trabalho de tese e seu impacto no projeto *EasyGrid* como um todo, serão apresentas as áreas de pesquisa dentro do projeto especificando o que se encontra dentro do escopo desta tese de doutorado e o que já foi

(está sendo) desenvolvido por outros pesquisadores. Uma visão geral do projeto *EasyGrid* pode ser visto na Figura 4.1. O foco desta tese de doutorado é a criação de um modelo de execução alternativo que possibilite a execução, principalmente, das aplicações paralelas fortemente acopladas nas grades computacionais. Como pode ser visto na Figura 4.1 o modelo de execução se encontra na base do projeto *EasyGrid*, de maneira que toda a implementação foi baseada no modelo alternativo proposto nesta tese.

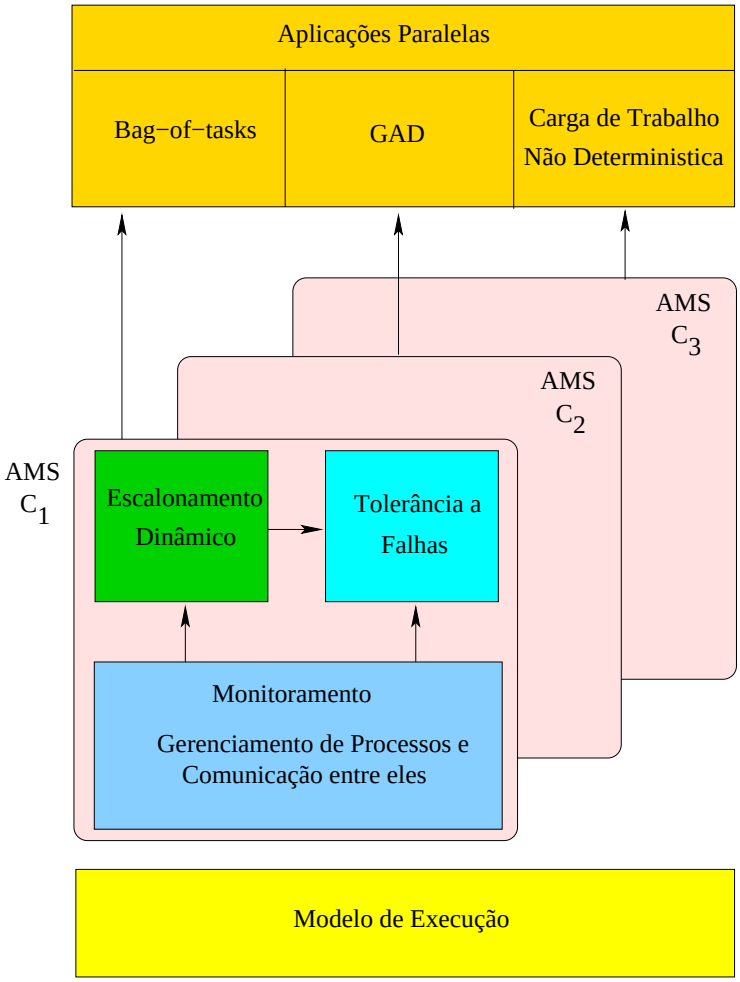

Figura 4.1: Áreas de pesquisa dentro do projeto EasyGrid.

Logo acima do modelo de execução se encontram as implementações do *EasyGrid AMS* e, como pode ser visto na Figura 4.1, existe uma implementação para cada classe de aplicação. As principais funções de cada uma dessas implementações são: monitoramento, gerenciamento de processos e a comunicação entre eles, escalonamento dinâmico e tolerância a falhas. Uma visão geral da arquitetura do *EasyGrid* pode ser encontrada em [19]. A implementação inicial do *middleware EasyGrid*, assim como a descrição básica dos serviços de monitoramento, gerenciamento e comunicação de processos, escalonamento dinâmico e tolerância a falhas podem ser encontrados em [119, 120, 152]. Uma explicação mais detalhada sobre tolerância a falhas para aplicações *bag-of-tasks* pode ser encontrada

em [45].

O estudo do escalonamento dinâmico já se encontra mais adiantado e sua implementação para aplicações *bag-of-tasks* pode ser vista em [18, 118] enquanto sua implementação para aplicações GAD pode ser encontada em [117]. Uma explicação completa sobre o escalonamento dinâmico no projeto EasyGrid pode ser vista em [116]. Com relação as aplicações composta de carga de trabalho não determinísticas, por exemplo meta-heurísticas, uma descrição detalhada pode ser encontrada em [46].

Finalmente, para conseguir mostrar a viabilidade de se executar aplicações fortemente acopladas nas grades através do modelo alternativo proposto, esta tese também apresenta um estudo com a aplicação N-corpos *ring* tornando-a uma aplicação evolutiva, ou seja, capaz de tomar decisões por si própia e com isso modificar sua execução em caso de mudanças no ambiente. A partir deste estudo, o desempenho das aplicações MPI fortemente acopladas não fica mais limitado ao processador mais lento, mas depende do grau de paralelismo que mazimiza sua execução assim como o escalonamento de tarefas dos seus processos. Neste trabalho de tese, o grau de paralelismo é definido como W, além disso, é proposto um algoritmo para calcular o seu valor ideal em conjunto com um bom escalonamento de suas tarefas, e com isso para maximizar o desempenho da aplicação N-corpos *ring*.

As duas próximas seções descrevem brevemente o sistema gerenciador *EasyGrid AMS* e seu escalonamento dinâmico para uma melhor compreensão do estudo apresentado nesta tese.

4.1.1 O *middleware EasyGrid* AMS

A finalidade do Sistema Gerenciador de Aplicações *EasyGrid AMS* [19, 119, 120] é executar e gerenciar eficientemente aplicações MPI em qualquer grade computacional que ofereça serviços básicos, como Globus [63] e uma biblioteca de comunicação MPI [58, 111]. Para atingir tal objetivo, o middleware *EasyGrid* adota o modelo de execução alternativo proposto nesta tese [133] juntamente com criação dinâmica de processos para prover escalonamento dinâmico e tolerância a falhas, tornando as aplicações capazes de se adaptarem às características das grades.

A arquitetura do *EasyGrid AMS* é composta por uma hierarquia de três níveis de processos gerenciadores, conforme apresentado na Figura 4.2: no nível mais alto se encontra o Gerenciador Global (*Global Manager - GM*) que é encarregado de supervisionar os *sites* onde a aplicação está executando (ou poderia executar); em cada um destes *sites*, um processo Gerenciador do Site (*Site Manager - SM*) é responsável pela alocação dos processos

da aplicação nos seus recursos; por último, no nível mais baixo da hierarquia, o Gerenciador da Máquina (*Host Manager - HM*), um para cada recurso, tem a responsabilidade de escalonar, criar e executar os processos da aplicação associados a cada máquina.

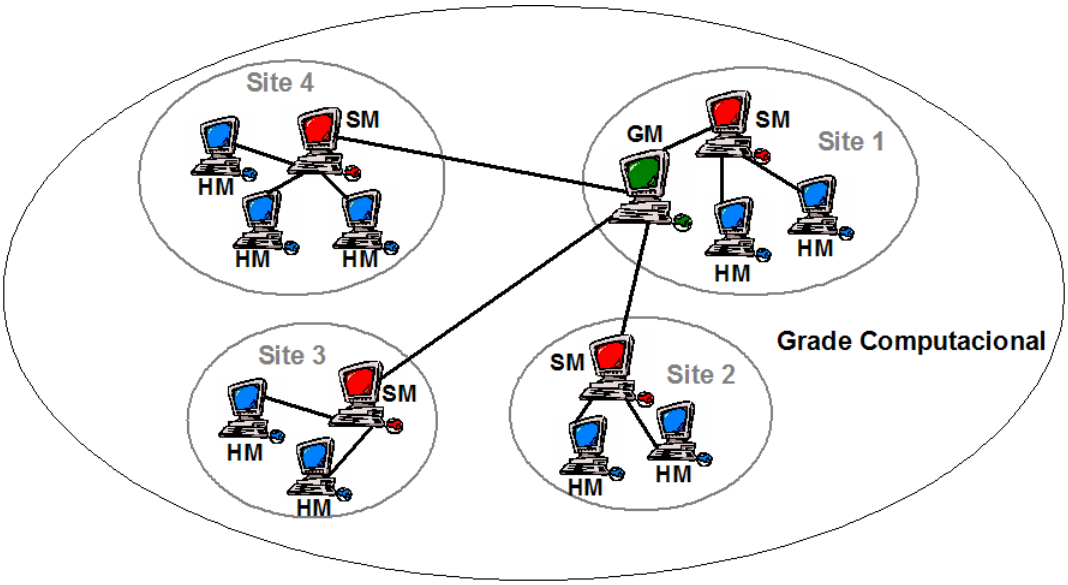

Figura 4.2: Hierarquia de gerenciadores do EasyGrid AMS.

Cada um dos processos gerenciadores do EasyGrid AMS está estruturado em uma arquitetura de camadas [24], como mostra a Figura 4.3, onde cada camada ou subsistema é responsável por um serviço específico e essencial para execução eficiente de uma aplicação MPI em uma grade computacional. Através de informações fornecidas pela camada de monitoramento, além de modificar seu próprio comportamento, é possível que uma camada de um nível mais alto modifique o comportamento de outra situada em um nível mais baixo.

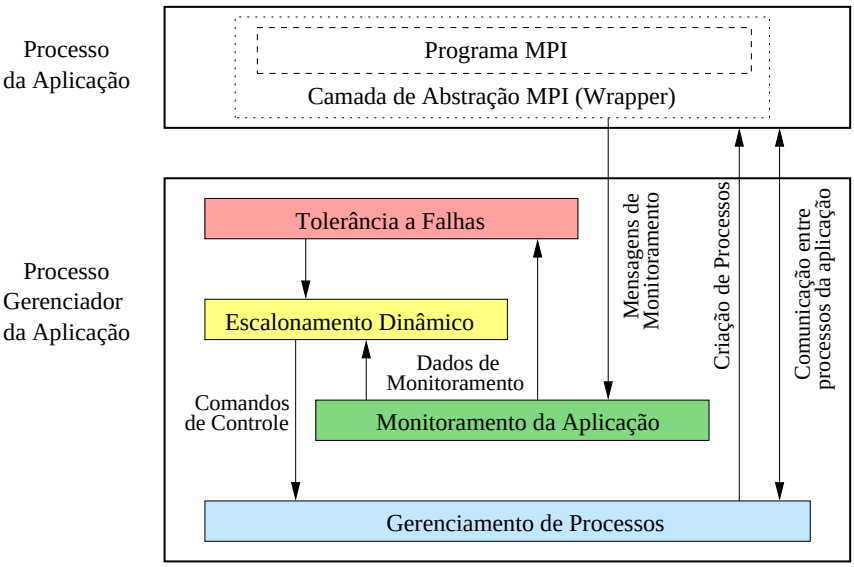

Figura 4.3: Arquitetura de camadas de cada processo gerenciador AMS.

A camada de **gerenciamento de processos** é responsável pela criação de processos MPI durante a execução da aplicação e pelo roteamento de mensagens entre eles. O subsistema de **monitoramento** disponibiliza as informações necessárias para que as camadas de **escalonamento dinâmico** e **tolerância a falhas** possam reescalonar os processos da aplicação e detectar e tratar falhas, respectivamente, quando houver necessidade.

É importante destacar que cada instância do *EasyGrid AMS* gerencia a execução de apenas uma aplicação, de modo que as políticas utilizadas em cada um dos níveis da hierarquia são específicas para esta aplicação. Por exemplo, para executar duas aplicações através do *EasyGrid AMS* duas instâncias são criadas, onde cada uma pode ter necessidades e políticas de gerenciamento distintas.

Diferentemente da maioria dos sistemas gerenciadores [12, 41, 146] (que executa apenas um processo por vez em cada processador), o *EasyGrid AMS* está preparado para executar um ou mais processos por vez, concorrentemente ou em paralelo (por exemplo, processadores *multicore* ou máquinas multiprocessadas). Desta forma, objetiva-se minimizar o custo da criação dinâmica através da sobreposição deste custo com a execução de uma outra tarefa.

Outra característica importante do *middleware EasyGrid* é a escalabilidade. A princípio, não existe limite no número de processos (tarefas) que uma aplicação pode ter. Por se tratar de um ambiente para executar aplicações em grades, espera-se que essas aplicações sejam de larga escala. O *EasyGrid AMS* está apto para executar uma aplicação paralela MPI que possa ser representada através de um GAD. Além disso, a sua estrutura hierárquica em três níveis permite uma escalabilidade maior com relação ao número de recursos do que ambientes que possuam uma estrutura centralizada [97, 120, 159, 116].

O conceito do *EasyGrid AMS* e sua arquitetura foram propostos em [19]. Já, a sua implementação que é capaz de transformar aplicações legadas MPI específicas para *clusters* em versões cientes ao ambiente (*system-aware*) para execução em grades computacionais, pode ser encontrada em [119, 120, 152]. Em [18, 116, 118] o desempenho do escalonamento híbrido foi avaliado, com enfoque no escalonador dinâmico para a execução de aplicações paralelas em uma grade computacional real. O modelo de execução alternativo adotado pelo *EasyGrid AMS* e a sua validação para aplicações *bag-of-tasks* foram apresentados em [133]. Neste artigo é feita uma comparação do desempenho de aplicações MPI executando através do modelo alternativo e aplicações que utilizam o modelo MPI tradicional (um processo por recurso). Por último, um estudo de caso sobre a aplicação N-corpos mostrando a viabilidade da execução de aplicações paralelas fortemente acopladas nas grades, pode ser visto em [132].

4.1.2 O Escalonamento de Tarefas do EasyGrid AMS

Um programa paralelo pode ser visto como um conjunto de tarefas que trocam dados entre si. É sabido que um bom escalonamento de tarefas é fundamental para que o programa obtenha um bom desempenho durante sua execução. Na maioria dos modelos de escalonamento existentes [11, 17, 29], se tarefas dependentes forem alocadas em um mesmo processador, o custo de comunicação associado é considerado nulo. Em contrapartida, a alocação deve ser cuidadosa o bastante para que o grau de paralelismo da aplicação seja explorado ao máximo, de modo que o tempo total de execução da aplicação seja o menor (melhor) possível. O escalonamento de tarefas na sua forma geral é um problema NP-completo [71, 125, 138], até mesmo quando todas as tarefas da aplicação são independentes [87, 148]. Logo, é necessário o uso de heurísticas [100, 131, 147] para que seja possível obter escalonamentos quase ótimos em tempos aceitáveis (polinomiais).

O problema de escalonar tarefas de uma aplicação paralela é normalmente definido através de três elementos essenciais [29, 36, 138]: um modelo da aplicação (Seção 3.2), um modelo arquitetural (Seção 3.3) e uma função objetivo. O modelo da aplicação define as principais características da aplicação paralela. O modelo arquitetural é usado para representar as características dos recursos onde a aplicação será executada, e também os canais de comunicação que interligam esses recursos. O terceiro e último elemento necessário para definir o problema de escalonar tarefas é a função objetivo utilizada, como por exemplo minimizar o tempo de execução (conhecido como minimizar o *makespan*) [125], minimizar o custo monetário de se executar uma aplicação [27], maximizar a vazão [86], entre outros.

Devido à natureza dinâmica das grades computacionais, o *middleware EasyGrid* utiliza um escalonamento híbrido [16, 107] que busca juntar as vantagens oferecidas pelos escalonadores estático e dinâmico. A função do escalonador estático é gerar um escalonamento de tarefas inicial, baseado nas características correntes ou estimadas das máquinas e nas características da aplicação. Como o escalonamento estático é realizado antes da execução da aplicação, ele pode utilizar heurísticas mais complexas que analisem a aplicação como um todo e, ainda, coletar informações sobre os recursos que estão disponíveis inicialmente na grade. Porém, durante a execução da aplicação, as estimativas utilizadas pelo escalonador estático podem se tornar defasadas devido às mudanças do ambiente. Por outro lado, o escalonador dinâmico utiliza informações atualizadas sobre a condição dos recursos e processos para tomar decisões de reescalonamento. Para não prejudicar a execução da aplicação, essas heurísticas tendem a ser mais simples.

Inicialmente, todas as tarefas da aplicação paralela executada através do *EasyGrid*

AMS são alocadas nos recursos disponíveis seguindo exatamente o escalonamento estático inicial. Porém, essas tarefas (processos MPI) não são criadas de uma só vez. Somente as tarefas prontas (conjunto de tarefas onde todas as mensagens foram recebidas pelos respectivos gerenciadores) podem ser executadas, sendo que execuções simultâneas em um mesmo recurso são controladas pelo *middleware* com o objetivo de otimizar o desempenho da aplicação. Devido à dificuldade de se conseguir medidas exatas sobre as condições dos recursos e prever a sua disponibilidade (grau de utilização), modificar a alocação estática inicial é fundamental para que a aplicação obtenha um bom desempenho. A migração de processos em execução é muito custosa, especialmente se houver a necessidade de *checkpoint* [39]. Até o momento, o *EasyGrid AMS* só permite a realocação de processos que ainda não foram criados (não estão em execução), apesar dele ser capaz de reexecutar processos que falharam no mesmo recurso ou em até um outro recurso.

A realocação de processos durante a execução é feita pelos escalonadores dinâmicos do *EasyGrid AMS*, que são associados aos processos gerenciadores da hierarquia de três níveis. No topo da hierarquia se localiza um único Escalonador Dinâmico Global (GDS), em cada um dos *sites* da grade existe pelo menos um Escalonador Dinâmico do Site (SDS) e, por último, em cada um dos recursos disponíveis Escalonador Dinâmico do Host (HDS). Como as grades se caracterizam por serem ambientes dinâmicos e compartilhados, a estimativa de desempenho dos recursos realizada durante a execução deve ser utilizada para especificar um escalonamento eficiente de modo que o tempo de fim (*makespan*) da aplicação seja minimizado.

Em conjunto, os escalonadores dinâmicos realizam as seguintes tarefas: calculam uma estimativa de quanto tempo falta para terminar a execução das tarefas em cada recurso, e consequentemente, o *makespan* da aplicação; verificam se a alocação necessita ser ajustada, caso onde um evento de escalonamento deve ser disparado. As políticas de escalonamento que serão utilizadas são implementadas em cada um dos três níveis da hierarquia e dependem da classe da aplicação e da necessidade dos usuários.

O GDS é responsável pelo reescalonamento de tarefas entre *sites*. Para decidir se deve ou não reescalonar tarefas, o GDS analisa as informações fornecidas por cada *site*. Essas informações são coletadas por cada SDS, que recebe dados de seus respectivos recursos através da camada de monitoramento. O reescalonamento de tarefas dentro de cada *site* é responsabilidade do SDS. Por fim, a ordem e o instante que uma tarefa deve ser executada em um recurso é determinado pela política de escalonamento do HDS.

O reescalonamento de tarefas é executado através de um evento de escalonamento. A frequência com que esses eventos ocorrem depende do grau de instabilidade do sistema (por exemplo, uma medida de quanto o *makespan* da aplicação mudou em relação a última

previsão). Em cada evento de escalonamento, um subconjunto de tarefas, avaliado pelo escalonador dinâmico, é reescalonado nos processadores disponíveis de acordo com a política implementada. A previsão do desempenho dos processadores [110, 154] é feita através de informações (tempo de duração e tempo de execução das tarefas) fornecidas pela camada de monitoramento, enquanto que mensagens *time-stamped* de monitoramento são usadas para estimar o custo de comunicação. Uma explicação detalhada sobre o escalonamento dinâmico do *EasyGrid AMS* pode ser encontrada em [18, 116, 118].

4.2 Avaliação do Modelo GAD para Aplicações *bag-of-tasks*

Esta seção apresenta uma avaliação do modelo de aplicação GAD para aplicações *bag-of-tasks*. Apesar da classe de aplicações do tipo *bag-of-tasks* possuir uma topologia simples, vários tipos de aplicações paralelas se encontram nessa categoria, como por exemplo *parameter sweep* [2, 14], que consiste de uma série de experimentos que utilizam diversos parâmetros distintos sobre um ou mais programas, e também mestre/trabalhador [142], que é constituída de um programa mestre que distribui dados para trabalhadores (tarefas) processá-los. Além disso, programas desse tipo são utilizados em diversas áreas, como por exemplo, física [142], biologia computacional [143], arqueologia, astronomia [8], computação de imagens [141], entre outras. Programas paralelos *bag-of-tasks* vem sendo amplamente executados em ambientes heterogêneos, inclusive nas grades computacionais, devido à ausência de comunicação entre as tarefas, o que simplifica a sua execução nesse tipo de ambiente.

Como proposto neste trabalho de tese, as informações sobre a aplicação, necessárias para que o *EasyGrid AMS* consiga uma execução eficiente, são passadas através de um GAD. As aplicações *bag-of-tasks* são facilmente representadas através de um GAD do tipo fork/join (Figura 4.4), onde o nó fonte/destino é usado apenas para enviar/recolher dados das tarefas que executam os experimentos *parameter sweep*, ou representa o nó mestre nas aplicações mestre/trabalhador. Neste trabalho, como exemplo de uma aplicação *bag-of-tasks*, foi utilizada uma aplicação *parameter sweep* para avaliar algoritmos de escalonamento [130, 151], chamada de *Sched*.

Por se tratar de uma aplicação *parameter sweep*, a maior dificuldade para gerar o GAD do tipo fork/join que representa essa aplicação é como estimar com precisão o peso de cada tarefa. Se todas as tarefas executam exatamente as mesmas instruções e recebem como entrada exatamente as mesmas quantidade de dados então todas as tarefas são associadas a um mesmo peso. Entretanto, as tarefas podem ser compostas por instruções distintas e principalmente, por parâmetros e arquivos distintos. Logo, é necessário formular uma

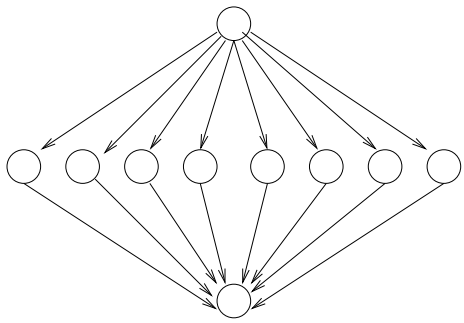

Figura 4.4: Exemplo de um GAD do tipo fork/join.

metodologia para avaliar o peso associado a cada tarefa da aplicação *bag-of-tasks*. A seção a seguir mostra uma abordagem para geração de pesos para tarefas da aplicação paralela *Sched* do tipo *parameter sweep*.

4.2.1 Metodologia para Estimar o Peso das Tarefas da Aplicação *Sched*

A aplicação *parameter sweep Sched* é constituída por tarefas ou *jobs* (nesta subseção, cada tarefa será chamada de *job*), onde cada *job* na verdade executa um algoritmo de escalonamento de tarefas para um dado grafo de entrada e uma matriz que representa a heterogeneidade do ambiente e parâmetros específicos a cada algoritmo de escalonamento [130, 151]. Em outras palavras, cada *job* da aplicação *parameter sweep* representa a execução de um programa de escalonamento de tarefas, sendo que cada algoritmo de escalonamento pode ter uma complexidade distinta e parâmetros completamente diferentes. Além disso, cada execução realiza entrada/saída, o que torna a avaliação do peso de cada *job* ainda mais difícil.

O ideal seria conseguir uma metodologia simples para avaliar o peso de cada *job*, por exemplo, uma única equação que incorporasse todas as características do *job* que recebesse como entrada a quantidade de tarefas da aplicação paralela a ser escalonada e retornasse o peso de cada *job*. Porém, através de experimentos, foi observado que três características realmente influenciavam no tempo de execução de cada *job* da aplicação *Sched*: a complexidade do algoritmo de escalonamento utilizado pelo respectivo *job*, o tipo de aplicação paralela a ser escalonada por esse algoritmo (GAD de entrada) e a quantidade de tarefas da respectiva aplicação. No entanto, a quantidade de processadores da arquitetura alvo onde a aplicação será escalonada praticamente não afetou o tempo de execução da tarefa, devido a seu valor baixo onde nenhum *job* recebeu mais do que 32 processadores. Assim, para conseguir modelar essas três características foram necessárias três equações para cada algoritmo de escalonamento utilizado, uma para cada classe de aplicação de entrada (diamante, árvores ou aleatórios) utilizada nos testes, onde cada uma

dessas equações têm como entrada o número de tarefas da aplicação.

Como exemplo, foram geradas equações para três algoritmos de escalonamento de tarefas (DCP [99], ETFRGC [93], LIST SCHEDULING [15]). A seguir será descrita a metodologia utilizada para gerar a equação associada ao algoritmo ETFRGC, para a classe de aplicação representada por um grafo do tipo árvore binária, onde x_i representa o número de tarefas da aplicação e y_i o tempo de execução (em milesegundos) do ETFRGC para escalonar a aplicação dada (árvore binária) em um conjunto de não mais do que 32 processadores.

Inicialmente, foi escolhido um subconjunto das instâncias da aplicação sendo avaliada (para árvores binárias foram escolhidas instâncias com 3, 7, 15, 31, 63, 127, 255 e 511 tarefas), informação esta armazenada no vetor *vet_inst* (linha 4 do Algoritmo 1). O escalonamento de cada uma dessas instâncias (execução do algoritmo ETFRGC) foi executado três vezes em uma determinada máquina e a média aritmética dos tempos dessas três execuções armazenada no vetor *vet_tempo* (para as árvores binárias as médias foram 9, 10, 11, 12, 13, 19, 47 e 170 milesegundos, respectivamente) (linha 5 do Algoritmo 1). Utilizando esses dois vetores como entrada, o comando $polyfit(vet_inst, vet_tempo, g)$ (linha 7 do Algoritmo 1) do programa OCTAVE do Linux gera uma equação de grau g (nesse exemplo a equação gerada foi y_i = 16.138 - 0.13526x_i + 0.0014655 ${x_i}^2$). Essa equação recebe como entrada o número de tarefas da classe da aplicação que está sendo avaliada, x_i, e retorna uma estimativa do tempo de execução do algoritmo, y_i.

Algoritmo 1 : AchaEquacao*()*

1 // vet_inst - contêm a quantidade de tarefas de cada instância utilizada
2 // vet_tempo - contêm o tempo de execução para cada instância utilizada
3
4 vet_inst=$[i_1, i_2, ..., i_n]$;
5 vet_tempo=$[t_1, t_2, ..., t_n]$;
6
7 p = polyfit(vet_inst, vet_tempo, g);
8 p(x) = A + Bx ... N x^g ;

Obviamente, o tempo que cada uma das equações retorna é proporcional ao poder computacional da máquina utilizada. Logo, para se obter o peso $\varepsilon(v_i)$ da tarefa v_i, independentemente da máquina utilizada, basta dividir o tempo de execução de uma tarefa v_i, y_i, pelo índice de retardo computacional $csi(p_j)$ da máquina utilizada, como mostra a Equação 4.1. O índice de retardo computacional de uma máquina (recurso) é um valor que representa o grau de diferença da capacidade de processamento do dado recurso de uma grade computacional, como foi explicado na Seção 3.3.

$$\varepsilon(v_i) = \frac{y_i}{csi(p_i)} \tag{4.1}$$

Para verificar a precisão dos valores estimados a partir das equações geradas, o programa *Sched* foi executado em apenas uma máquina da grade Sinergia [137], identificada como sn19. O objetivo desse experimento é comparar o tempo real de execução da aplicação com o tempo previsto, calculado através das equações. Para isso foram executadas várias instâncias dessa aplicação variando-se a quantidade de tarefas. Cada *job* da aplicação *Sched* é uma execução do algoritmo de escalonamento de uma aplicação paralela, onde o tempo de execução na sn19 varia entre 10 e 10.000 milesegundos, dependendo do escalonador utilizado (DCP, ETFRGC, LIST SCHEDULING), tipo de aplicação paralela a ser escalonada (GAD de entrada) e quantidade de tarefas do GAD de entrada. A Tabela 4.1 mostra a comparação entre o tempo real de execução desses experimentos na máquina sn19 através do *middleware EasyGrid* e uma previsão utilizando as equações geradas através do OCTAVE.

Tabela 4.1: Tempo de execução real e estimado da aplicação *Sched* em segundos na máquina sn19

#tarefas	tempo de execução	previsão	erro (%)
10584	3151,7	3008,5	4,8
5328	1519,5	1501,0	1,2
3150	857,2	840,9	1,9
1176	339,8	348,3	2,5

Como mostrado na Tabela 4.1, a maior diferença (erro) entre o tempo de execução real da aplicação usando o *middleware EasyGrid* e a estimativa (calculada usando apenas as equações) é de apenas 4,8%. Esses resultados mostram que as equações conseguem prever o tempo de execução de cada tarefa com precisão, uma vez que esta aplicação contém tarefas com duração entre 10 e 10.000 ms, e que ainda realizam entrada e saída. Além disso, o tempo de execução real da aplicação contém o custo de sobrecarga do *middleware EasyGrid* que, como mostrado em [119], é de mais ou menos 2%. Logo, se excluirmos a sobrecarga do *middleware EasyGrid*, a precisão das equações é ainda maior.

Através da metodologia apresentada nesta subseção, é possível a geração de equações para outros algoritmos de escalonamento, possibilitando uma estimativa do peso de um *job* que utilize esse algoritmo.

4.3 Avaliação e Análise do Modelo de Computação

Esta seção, baseada em experimentos utilizando o *middleware EasyGrid*, analisa e avalia o modelo de computação proposto para execução de aplicações MPI nas grades

computacionais. O modelo de computação representa essencialmente as características dos processadores disponíveis na grade. O desempenho das aplicações *bag-of-tasks* depende quase que exclusivamente do poder computacional dos processadores disponíveis, uma vez que essas aplicações se caracterizam por possuírem tarefas que não se comunicam entre si. No próximo capítulo será apresentada a validação das aplicações paralelas composta de tarefas que se comunicam e também, uma análise do modelo de comunicação. É importante destacar que todos os resultados que serão apresentados ao longo desta tese foram obtidos através da média aritmética de três execuções, no mínimo. Isso se deve a proximidade dos valores obtidos, não sendo necessário calcular um intervalo de confiança para maioria dos experimentos realizados. Entretanto, em alguns experimentos, ocorreu uma variação maior entre os resultados obtidos. Nestes casos, o número de execuções realizadas se encontra descrito explicitamente no experimento, juntamente com o cálculo do desvio padrão ou intervalo de confiança para garantir uma melhor análise dos resultados.

Como exemplo de um ambiente real, a Figura 4.5 apresenta os processadores disponíveis na UFF da Grade Sinergia [137], assim como os canais que interligam esses recursos. Essa grade é composta por máquinas com processadores Intel na sua maioria, as máquinas sn possuem processadores Pentium IV 2.6 GHz com 512Mb de memória RAM, as máquinas gr possuem processadores Pentium IV 3.2 GHz com 512Mb e as máquina com nome de frutas possuem processadores e quantidade de memória variados. Todas as máquinas da grade possuem instalado o sistema operacional Linux Fedora, Globus toolkit 2.4 e LAM/MPI 7.0.6.

Como explicado na subseção 3.3.1, esse trabalho adota um modelo computacional mais simplificado, que utiliza um índice de retardo computacional $csi(p_j)$ para estimar o poder computacional de cada processador $p_j \in P$ da grade. O índice de retardo computacional de uma máquina é um valor que representa o grau de diferença da capacidade de processamento dos recursos de uma grade computacional. Para que o tempo de execução $te(v_i)$ de cada tarefa $v_i \in V$ da aplicação seja estimado, basta multiplicar o peso da tarefa $\varepsilon(v_i)$ pelo índice $csi(p_j)$, como mostra a Equação 4.2.

$$te(v_i, p_j) = \varepsilon(v_i) \times csi(p_j) \tag{4.2}$$

Para capturar o índice $csi(p_j)$ de cada máquina $p_j \in P$ é utilizado um programa que executa uma dada tarefa com as mesmas características das tarefas da aplicação paralela em questão em cada máquina de P e retorna o poder computacional de cada máquina testada. Esse valor é proporcional ao poder computacional da máquina, assim como o número de processos que estiverem executando nela. A Tabela 4.2 apresenta o índice de retardo

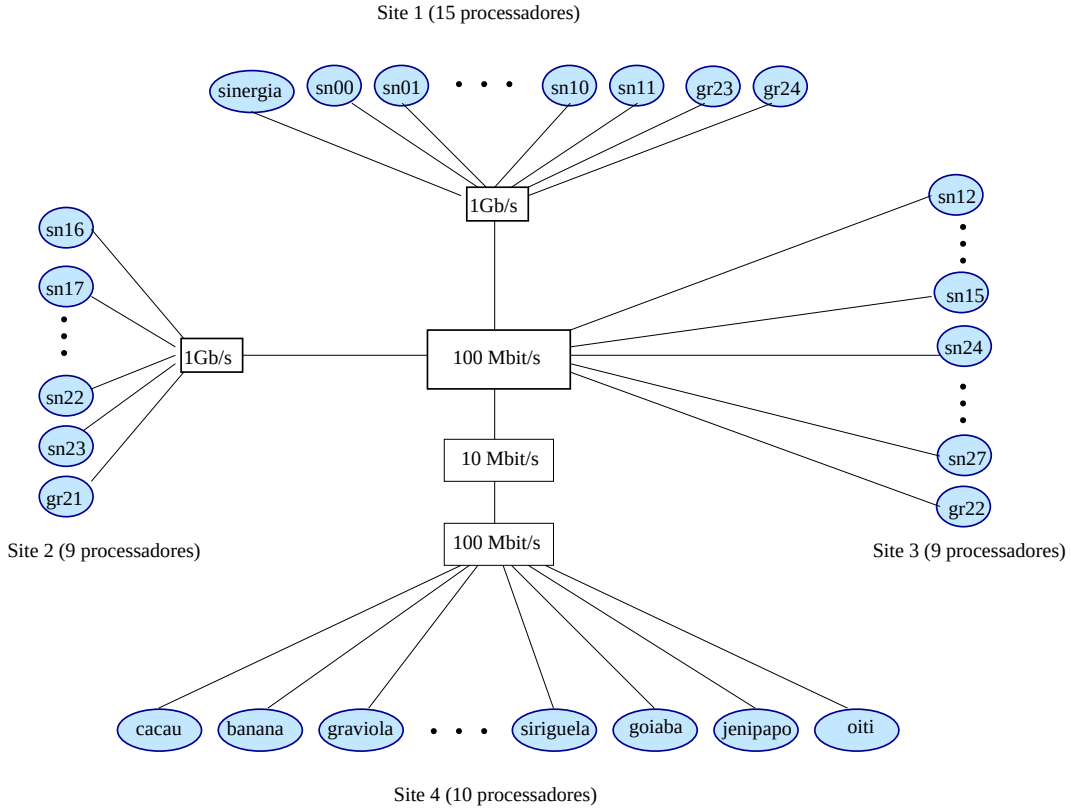

Figura 4.5: Arquitetura dos recursos da grade Sinergia, disponíveis na UFF.

computacional para as máquinas da grade UFF da Figura 4.5, quando elas estão dedicadas a execução do programa modelador. De acordo com a nomenclatura utilizada, quanto menor for este valor melhor o desempenho. A coluna csi CPU bound apresenta o índice de retardo calculado através de um programa *CPU bound*, que executa as seguintes operações: soma, soma de matrizes e eliminação gaussiana. Já a coluna csi Sched apresenta o índice de retardo calculado utilizando uma pequena amostra do programa paralelo *Sched* (Seção 4.2), onde cada tarefa especifica um escalonamento de tarefas de um GAD de entrada e realiza entrada/saída. Como pode ser observado, o desempenho dos processadores varia de acordo com o tipo de tarefa que eles executam. Por exemplo, as máquinas sn00 a sn27 obtiveram um índice de retardo menor para tarefas *CPU bound* do que para tarefas do *Sched*. Ao contrário, as máquinas gr21 a gr24 obtiveram um índice de retardo menor para tarefas do *Sched* do que para tarefas *CPU bound*.

Tendo em vista que o desempenho de uma tarefa varia em função dos tipos de instruções executadas, duas abordagens foram propostas na subseção 3.3.1: a execução de um código padrão, contendo diversos tipos de instruções, que retorna o índice de retardo computacional para as máquinas da grade; e a execução de um conjunto de instruções específicas para o tipo de aplicação a ser executada. A vantagem da primeira abordagem é ser mais genérico e simples de implementar. Porém, como apresentado na Tabela 4.3, ela

Tabela 4.2: Índice de retardo computacional dos recursos da grade Sinergia, disponíveis na UFF

recurso	csi CPU bound	csi Sched
sinergia	23	23
sn00 - sn27	23	26
gr21 - gr24	22	19
cacau, banana e graviola	21	24
goiaba	46	45
jenipapo	42	41
siriguela	44	43
sapoti	117	97
pinha	45	44
melancia	300	227
oiti	127	104

não apresenta resultados tão precisos quanto a segunda abordagem que utiliza um código muito similar ao que vai ser realmente executado. Um atenuante ao se utilizar a primeira abordagem para o escalonamento de tarefas no *EasyGrid AMS* é que, devido à estratégia de escalonamento híbrida adotada, parte dessa imprecisão pode ser corrigida pelo escalonador dinâmico que é capaz de adaptar a execução da aplicação a mudanças no ambiente como será mostrado no Capítulo 6.

Um pequeno exemplo da diferença de precisão entre as duas abordagens para aplicações do tipo *bag-of-tasks* é apresentado a seguir. De acordo com o tipo de instrução a ser executada, a máquina pode apresentar comportamento distinto, como pode ser visto nos resultados da Tabela 4.3.

Tabela 4.3: Previsão do tempo de execução (em segundos) com diferentes tipos de instruções

recurso	tempo de execução		csi	tempo estimado	erro(%)
sn19	3151,7	Instruções Similares	26	3008,5	4,7
		Instruções Distintas	23	2661,3	18,4
oiti	12477,3	Instruções Similares	104	12033,9	3,7
		Instruções Distintas	127	14495,3	16,2
banana	2860,1	Instruções Similares	24	2777,1	3,0
		Instruções Distintas	21	2429,9	17,7

A Tabela 4.3 mostra o índice csi e o tempo estimado calculado ao utilizar o modelador com instruções de dois tipos. Em primeiro lugar, o modelador foi executado com instruções correspondentes a uma pequena amostra da aplicação *Sched* (Instruções Similares). Em seguida, o modelador foi executado com instruções de soma, multiplicação de matrizes e uma eliminação gaussiana (Instruções Distintas). Como apresentado na Tabela 4.3, em Instruções Similares o modelador avaliou a capacidade de processamento das máquinas com uma boa precisão, sendo o erro sempre menor do 4,7%. Por outro lado, em Instruções Distintas o erro não foi nunca maior do que 20%. Desse modo, é possível concluir que a utilização de instruções similares as da aplicação a ser executada na grade proporciona

uma melhor precisão no calculo dos tempos de execução das tarefas e consequentemente na previsão do tempo de execução da aplicação. Entretanto, como nem sempre é possível ou viável de se obter tais instruções, a utilização de um conjunto de instruções comumente usadas em programas científicos permite também calcular uma previsão dos tempos das tarefas, apesar desse cálculo apresentar uma precisão menor. Porém, essa perda na precisão pode ser atenuada através da utilização de um escalonamento dinâmico como será mostrado no Capítulo 6.

4.3.1 Grau de concorrência

Uma consequência natural deste trabalho de tese foi avaliar o grau de concorrência entre processos executando em um mesmo recurso. Ou seja, quantos processos uma aplicação deve executar concorrentemente em cada recurso? Como a maioria das aplicações adota o modelo tradicional *um processo por processador*, é natural que elas executem apenas um processo por recurso. Além disso, como foi visto na Subseção 3.1.1 a execução de mais de um processo por recurso para aplicações MPI com tarefas dependentes, pode prejudicar o seu desempenho. Outros fatores que devem ser observados são: o número de processadores (núcleos) de uma máquina e outras características dos processadores, como por exemplo *HyperThreading*. Diferentemente, o modelo alternativo adota a execução de um processo por tarefa, sendo natural que a quantidade de tarefas da aplicação seja muito maior que o número de recursos disponíveis. Logo, o número de processos a ser executado em cada recurso concorrentemente deve ser tal que maximize o desempenho da aplicação.

Como mostrado em [119, 120], o framework *EasyGrid AMS* não limita o número de tarefas (processos) que podem ser criados e executados concorrentemente em um recurso, como pode ser observado na Tabela 4.4. Esse resultados foram obtidos utilizando apenas três processadores (Figura A.3 no Apêndice), onde o primeiro executa o GM, o segundo o SM e o terceiro o HM. Nesse experimento, apenas o processador com o HM executa tarefas da aplicação (apesar do GM executar o mestre ou tarefa 0, origem da aplicação). Uma aplicação *parameter sweep*, chamada de PSwp, foi utilizada nesta avaliação, composta de 840 tarefas independentes (sem contar a tarefa 0), onde cada tarefa possui o mesmo peso, ou seja, executa exatamente o mesmo código.

As colunas da Tabela 4.4 representam respectivamente o número de tarefas da aplicação PSwp (N), o número de processos que estão executando concorrentemente (#processos) e a granularidade de cada tarefa da aplicação. Os resultados da tabela mostram que, na prática, o número de processos da aplicação executando concorrentemente não deve ser grande, pois o desempenho degrada devido à troca de contexto. Para processos de granularidade fina, o melhor desempenho foi alcançado executando dois processos

Tabela 4.4: Variação do tempo de execução da aplicação PSwp com o aumento do número de processos em uma máquina com apenas 1 processador.

		granularidade da tarefa (em segundos)				
N	#processos	0,01	0,1	0,5	1,0	10,0
840	1	23,61	99,59	436,57	856,54	**8.424,31**
	2	**15,77**	91,67	**428,25**	**849,24**	8.429,48
	3	15,91	90,89	428,84	850,49	8.443,16
	5	15,94	90,91	430,09	851,16	8.477,65
	10	15,95	**90,87**	432,61	852,33	8.540,80
	15	15,99	90,91	433,69	855,74	8.614,30
	20	15,79	90,91	436,04	860,45	8.685,91
	50	15,91	91,00	438,63	875,10	9.177,01
	70	15,94	91,03	444,02	887,44	9.585,14
	100	15,90	90,92	461,30	899,12	10.212,10
	200	15,98	91,53	475,53	1.080,21	13.313,52

concorrentemente. Entretanto, para processos de granularidade grossa (10 segundos), a melhor abordagem é executar apenas um processo por vez, evitando um número grande de troca de contextos. O *middleware EasyGrid AMS* permite a execução de uma grande quantidade de processos/tarefas concorrentemente em cada processador, sendo limitado apenas pelo número máximo de processos que podem ser criados simultaneamente na plataforma MPI (Subseção 3.1.1). Através desses resultados também é possível verificar a sobrecarga do *EasyGrid AMS* (incluindo a sobrecarga associada ao MPI). Para processos com granularidade muito pequena (tarefas de 10 ms) a sobrecarga é muito alta em relação ao tempo de execução esperado (em torno de 87%), porém, à medida que a granularidade do processo aumenta a sobrecarga diminui drasticamente, para processo de 10 segundos a sobrecarga é de apenas 0,3%. É importante destacar que para processos de granularidade fina (menores ou iguais a 1,0 segundo) a execução de apenas um processo por processador concorrentemente é ruim, pois o custo da criação dinâmica tem um impacto maior no tempo de execução da tarefa. Logo, a execução de dois ou mais processos concorrentemente possibilita a sobreposição desse custo com a execução de um outro processo.

Uma observação importante que deve ser destacada é que nem sempre é possível executar muitos processos simultaneamente, pois a aplicação pode não possuir tantas tarefas independentes. Além disso, a execução de multiplos processos simultaneamente na mesma máquina dificulta (diminui o número de processos disponíveis para) o escalonamento dinâmico baseado em processos que ainda não foram criados, conforme descrito na seção 3.1. Logo, apesar do resultado apresentado na coluna `0,1` da Tabela 4.4 indicar que o melhor tempo de execução foi alcançado com 10 processos concorrentemente (90,87), do ponto de vista de um gerenciador é muito mais simples, além de beneficiar o modelo de escalonamento dinâmico proposto, executar 3 processos e conseguir um desempenho muito similar (90,89).

Os resultados do experimento anterior foram obtidos utilizando uma máquina com apenas um processador, máquina (1) descrita a seguir. Porém, a realidade atual aponta para máquinas com mais de um processador. Deste modo, o mesmo experimento foi repetido utilizando máquinas com 2 e 4 processadores. Além disso, foi investigado também a função *HyperThreading* [115] presente em grande parte dos processadores atuais. As máquinas utilizadas neste experimento foram as seguintes:

(1) Pentium IV 2.6 Ghz, 512 Mb de memória e executando o sistema operacional Linux Fedora;

(2) Intel Xeon Dual processado 3 Ghz, 4 Gb de memória e executando o sistema operacional Linux CentOS;

(3) AMD Opteron Dual processado Dual core 1.8 Ghz, 4 Gb de memória e executando o sistema operacional Linux CentOS.

Como objetivo desse experimento é mostrar a influência do número de processos executando concorrentemente em máquinas com mais de um processador (ou mais de um núcleo), a granularidade das tarefas foi ajustada especificamente para a capacidade de processamento de cada máquina. Ou seja, o tempo de execução de cada tarefa (granularidade da tarefa) é o mesmo em todas as máquinas testadas. Deste modo, é possível analisar a influência do aumento no número de processadores no desempenho da aplicação. Entretanto, é importante observar que cada uma das máquinas analisadas possuíam configurações distintas não só em relação a velocidade de processamento, mas também em relação a fabricante, a quantidade de memória principal e memória cache. Logo, é esperado que essas diferenças entre as máquinas utilizadas influenciem os experimentos realizados.

Tabela 4.5: Variação do tempo de execução da aplicação PSwp com o aumento do número de processos na máquina (2) com 2 processadores sem *HyperThreading*.

		granularidade da tarefa (em segundos)				
N	#processos	0,01	0,1	0,5	1,0	10,0
840	1	13,10	88,86	424,69	845,10	8.404,18
	2	8,50	46,35	216,01	429,03	**4.255,30**
	3	7,75	45,19	215,54	**427,66**	4.258,57
	5	7,70	44,70	215,17	428,55	4.262,41
	10	7,71	44,80	215,30	428,64	4.269,79
	15	7,76	**44,68**	215,57	429,05	4.275,56
	20	7,73	44,82	215,78	429,51	4.282,03
	50	7,74	44,89	216,49	433,10	4.330,85
	70	**7,69**	44,84	216,80	435,03	4.376,49
	100	7,72	44,93	215,89	437,97	4.459,95
	200	7,73	45,01	**214,79**	443,33	4.604,96

Como pode ser visto na Tabela 4.5, o desempenho da aplicação PSwp foi praticamente duas vezes melhor que a instância correspondente no experimento anterior, quando #processos ≥ 2, o que era esperado uma vez que a máquina do experimento corrente possui dois processadores. Para processos com granularidade fina o ideal é a utilização de 3 ou 5 processos concorrentemente. Porém, para processos com granularidade mais grossa o ideal é utilizar 2 ou 3 processos para evitar a troca de contexto e, consequentemente, perda de desempenho. Comparando os tempos de execução com apenas 1 processo por vez desse experimento com a respectiva linha da Tabela 4.4 é possível perceber uma diferença razoável (em torno de 10 segundos) para todas as granularidades. Apesar das tarefas possuírem praticamente os mesmos tempos de execução nos dois experimentos, existe uma pequena diferença, e mais importante, por se tratarem de máquinas distintas com sistemas operacionais distintos, os custos associados a criação e execução dos processos são ligeiramente diferentes, o que causa as diferenças nos tempos totais de execução.

Tabela 4.6: Variação do tempo de execução da aplicação PSwp com o aumento do número de processos na máquina (2) com 2 processadores com *HyperThreading*.

		granularidade da tarefa (em segundos)				
N	#processos	0,01	0,1	0,5	1,0	10,0
840	1	13,80	90,01	428,01	850,51	8.448,92
	2	7,90	46,30	215,90	428,23	4.235,61
	3	7,10	43,66	208,70	413,70	4.118,33
	5	**6,60**	42,33	202,61	402,40	**3.995,15**
	10	6,62	42,11	**201,88**	**401,85**	3.999,80
	15	6,62	**42,02**	202,05	402,20	4.003,24
	20	6,70	42,05	202,20	402,65	4.009,20
	50	6,65	42,05	202,99	404,86	4.041,31
	70	6,65	42,03	203,10	406,68	4.064,67
	100	6,67	42,05	203,76	409,36	4.095,12
	200	6,70	42,07	203,96	412,10	4.198,43

O mesmo experimento anterior foi repetido na mesma máquina com 2 processadores, porém com a função *HyperThreading* habilitada. O desempenho foi similar, sendo em geral duas vezes melhor que o desempenho da máquina com apenas 1 processador. Porém, os resultados da Tabela 4.6, mostram que devido à tecnologia *HyperThreading* utilizar dois *threads* virtuais para cada processador, a execução de um maior número de processos concorrentemente consegue melhorar ainda mais o desempenho. Para processos com granularidade fina a utilização de mais de 5 processos simultaneamente normalmente proporcionou o melhor desempenho. Para processos com granularidade grossa, por exemplo 10,0 segundos, a execução de 5 processos concorrentes maximizou o desempenho.

Por último, como pode ser visto na Tabela 4.7, de uma maneira geral, quando foram executados mais de 4 processos simultaneamente, a aplicação teve um desempenho quatro vezes melhor em relação a máquina com um processador e duas vezes melhor do que

Tabela 4.7: Variação do tempo de execução da aplicação PSwp com o aumento do número de processos na máquina (3) com 4 *cores*.

		granularidade da tarefa (em segundos)				
N	#processos	0,01	0,1	0,5	1,0	10,0
840	1	20,73	97,20	435,47	855,71	8.417,82
	2	12,24	48,40	218,10	435,92	4.230,70
	3	11,48	32,61	145,32	286,93	2.822,90
	5	11,50	25,76	109,91	215,57	**2.115,20**
	10	11,48	25,77	109,42	215,63	2.115,21
	15	**11,46**	25,34	109,66	215,60	2.116,51
	20	11,51	25,68	109,43	215,43	2.120,23
	50	11,52	**25,26**	109,65	215,39	2.133,63
	70	11,51	25,70	109,44	215,38	2.146,67
	100	11,53	25,62	**109,27**	**215,37**	2.161,76
	200	11,56	25,70	109,48	215,48	2.217,81

a máquina com dois processadores, o que já era esperado. Porém, devido as características/configuração da máquinas utilizada neste experimento, para tarefas com granularidade muito fina (0.01 segundos) o desempenho foi inferior ao conseguido com a máquina com dois processadores. Com relação ao número de processos que devem ser executados concorrentemente, neste experimento (que não utilizou a função *HyperThreading*) a utilização de 5 processos proporcionou um bom desempenho para todas as granularidades.

4.3.2 Desempenho e Escalabilidade

O próximo experimento investiga os custos associados à biblioteca MPI, empregando a mesma arquitetura e aplicação do experimento anterior. O objetivo desse experimento é avaliar o impacto do aumento do número de processos/tarefas no desempenho da aplicação, sendo esta executada de três maneiras distintas:

- estática - modelo tradicional de execução MPI onde todas os processos são criados estaticamente e a comunicação entre processos é feita através de RPI-TCP;
- lamd - que é exatamente igual a anterior, porém a comunicação entre processos é feita através de um processo intermediário (*deamon*);
- AMS - onde o middleware *EasyGrid AMS* é utilizado para criar dinamicamente os processos e gerenciá-los.

No entanto, no caso agora estudado, ao invés de apenas um recurso executar tarefas da aplicação, todos os três recursos disponíveis são utilizados. Desta forma, para a aplicação AMS, os gerenciadores local (SM) e global (GM) executam concorrentemente

com as tarefas da aplicação (nesse experimento dois processos da aplicação são executados concorrentemente).

Tabela 4.8: Comparação entre o AMS e os modos estático e lamd (em segundos)

		Granularidade das Tarefas				
N	Modo	0,01s	0,1s	0,5s	1,0s	10,0s
50	estático	1,30	2,85	9,57	18,40	170,18
	lamd	1,26	2,59	9,36	17,78	169,40
	AMS	0,99	2,51	9,35	18,29	171,98
100	estático	2,94	5,91	19,33	36,34	343,23
	lamd	1,95	4,99	18,48	35,40	340,40
	AMS	1,35	4,39	18,05	35,20	341,87
376	estático	52,01	63,56	113,22	176,20	1.327,5
	lamd	25,20	35,76	86,42	148,56	1.307,9
	AMS	3,37	15,10	65,60	129,91	1.256,6
500	estático	**ne**	**ne**	**ne**	**ne**	**ne**
	lamd	53,09	68,22	135,57	218,41	1.722,9
	AMS	4,40	19,64	86,44	170,87	1.679,1
738	estático	**ne**	**ne**	**ne**	**ne**	**ne**
	lamd	154,16	183,65	280,04	402,55	2.622,7
	AMS	5,87	28,85	128,07	252,64	2.472,9
10^3	estático	**ne**	**ne**	**ne**	**ne**	**ne**
	lamd	**ne**	**ne**	**ne**	**ne**	**ne**
	AMS	7,90	38,54	175,47	344,01	3.422,1
10^4	estático	**ne**	**ne**	**ne**	**ne**	**ne**
	lamd	**ne**	**ne**	**ne**	**ne**	**ne**
	AMS	88,32	401,58	1.778,0	3.530,1	33.515,2

A Tabela 4.8 compara o tempo de execução da aplicação PSwp quando seus processos são criados estaticamente e dinamicamente. A primeira coluna da tabela mostra o número N de tarefas independentes da aplicação PSwp. A segunda coluna especifica o modo como a aplicação foi executada (estático, lamd ou AMS), as colunas seguintes especificam o tempo de execução da aplicação à medida que a granularidade dos processos aumenta. As células com as letras **ne** representam o fato de que, para os modos estático e lamd, não foi possível executar aplicações com a biblioteca LAM/MPI com mais de 376 e 738 processos, respectivamente, utilizando três processadores. Duas características importantes podem ser destacadas a partir dos resultados apresentados na tabela: bom desempenho e escalabilidade do *middleware EasyGrid*. Com um pequeno número de processos, o desempenho da aplicação executada através do *AMS* é um pouco melhor do que o modo estático. À medida que o número de processos aumenta, o desempenho do *AMS* melhora significantemente e, mais importante, não degrada para um número grande de processos. Por outro lado, a execução de uma aplicação com muitos processos através da execução tradicional MPI, apresenta resultados piores e não é escalável, se limitando a um número pequeno de processos.

Comparando as duas últimas linhas da Tabela 4.8 executando através do modo AMS é possível perceber uma pequena degradação no tempo total de execução para tarefas com granularidade fina (menores ou iguais a 1,0 segundos). Devido ao grande aumento na quantidade de tarefas da aplicação (10 vezes mais tarefas), o *middleware EasyGrid* produziu uma sobrecarga um pouco maior ao gerenciá-las. Entretanto, essa sobrecarga foi pequena (no máximo 11% para tarefas de granularidade de 0,01 segundos), diminuindo para granularidades maiores (para tarefas de granularidade de 0,5 e 1.0 segundos foi de 1,3% e 2,6%, respectivamente). Ao contrário, para tarefas com granularidade de 10,0 segundos a sobrecarga do gerenciador *EasyGrid AMS* diminuiu 2,0% mostrando a escalabilidade do gerenciador a medida que a quantidade de tarefas e a granularidade das tarefas aumenta.

4.3.3 Desempenho do *EasyGrid AMS* na presença de outras aplicações

Uma das características de uma grade computacional é ser um ambiente compartilhado entre diversas aplicações. Logo, é fundamental realizar avaliações do desempenho do *EasyGrid AMS* em um ambiente compartilhado com o objetivo de identificar características importantes para otimizar o desempenho da aplicação.

O objetivo do próximo experimento é avaliar o número ideal de processos que devem executar concorrentemente em uma mesma máquina, mostrando que esse número depende do peso das tarefas da aplicação e da quantidade de processos de outras aplicações que estejam executando junto.

Apenas três processadores (Figura A.3 no Apêndice) foram utilizados, onde o primeiro executa o GM, o segundo o SM e o terceiro o HM. Nesse experimento, apenas o processador com o HM executa tarefas da aplicação (apesar do GM executar o mestre ou tarefa 0, origem da aplicação). A aplicação *parameter sweep*, PSwp, foi utilizada nesta avaliação, composta de 500 tarefas independentes (sem contar a tarefa 0), onde cada tarefa tem o mesmo peso (executa exatamente o mesmo código).

Na Tabela 4.9 a coluna cenário apresenta quatro situações distintas para a máquina que contém o HM (e também, tarefas da aplicação): (-) a máquina dedicada para a aplicação PSwp; (1) com um processo externo ao PSwp (de outra aplicação); (2) com dois processos externos; (3) e com três processos externos. De acordo com os resultados apresentados na Tabela 4.9 conclui-se que, para tarefas com tempo de execução abaixo de 200 ms, o *middleware* deve permitir que duas ou mais tarefas executem concorrentemente para que o custo da criação dinâmica de um processo MPI (em torno de 10 ms) seja sobreposto com a execução de tarefas. Porém, para tarefas com tempo de execução maior do que 10 segundos, esse custo se torna irrisório e a concorrência passa a acarretar uma quantidade

Tabela 4.9: Tempo de Execução (em segundos)

		Granularidade da tarefa (em segundos)				
cenário	#processos	0,05s	0,1s	0,2s	1,0s	10,0s
(-)	1	37,75	63,41	112,93	513,91	**5.026,63**
	2	30,40	54,89	105,16	**506,30**	5.033,84
	3	29,19	54,23	104,82	507,40	5.037,70
	4	29,31	54,18	104,91	507,72	5.041,84
	5	29,27	**54,17**	104,92	508,18	5.048,82
	10	**29,25**	54,19	**104,55**	511,76	5.101,77
(1)	1	37,78	62,35	136,56	1.015,28	10.054,73
	2	38,04	58,98	121,48	760,90	7.550,17
	3	34,10	55,68	116,05	676,84	6.719,49
	4	33,08	**55,13**	114,09	634,70	6.326,40
	5	31,29	55,15	112,66	609,96	6.117,96
	10	**29,98**	55,15	**106,37**	**563,92**	**5.619,44**
(2)	1	37,87	63,67	169,33	1.525,60	15.094,59
	2	48,56	60,90	139,24	1.016,13	10.071,56
	3	33,18	**58,58**	129,23	847,12	8.417,77
	4	35,52	55,86	123,23	762,19	7.587,26
	5	33,43	55,84	119,77	711,85	7.069,73
	10	**30,82**	55,81	**111,83**	**614,94**	**6.140,61**
(3)	1	37,15	62,75	199,88	2.034,47	20.143,04
	2	50,68	62,80	155,24	1.270,56	12.620,73
	3	38,66	56,48	140,28	1.016,79	10.111,30
	4	37,43	**56,14**	132,99	889,53	8.836,11
	5	34,95	56,14	127,97	813,03	8.086,23
	10	**31,01**	56,37	**109,32**	**665,46**	**6.628,82**

maior de troca de contexto. Deste modo, o *middleware* deve criar e executar apenas um processo por vez em cada processador.

Uma outra característica interessante das tarefas com tempo de execução menor ou igual a 200 ms é que sua execução é pouco afetada pela presença de tarefas de outras aplicações. Esse comportamento acontece devido à proximidade de seu tempo de execução à fatia de tempo (*time slice*) do sistema operacional e a criação dinâmica de processos. Nos sistemas operacionais existe (pelo menos) uma fila de processos prontos para executar na CPU. Normalmente é adotada uma política preemptiva, onde o processo que entra na CPU executa durante uma fatia de tempo (no sistema operacional utilizado era de mais ou menos 100ms) e depois perde a CPU e vai para o final da fila. No *EasyGrid AMS* os processos são criados dinamicamente, e desse modo têm prioridade de executar na CPU em relação aos processos que já se encontram executando. Logo, como o seu tempo de execução é muito próximo da fatia de tempo, assim que eles terminam de executar, um novo processo criado dinamicamente ganha a CPU.

Ainda como conclusão deste mesmo experimento sob o impacto de cargas externas, o desempenho da aplicação melhora à medida que o número de processos executados concorrentemente aumenta (Na maioria dos casos #processo = 10 forneceu o melhor resultado).

Isso se deve ao fato que com este aumento, maior o percentual de CPU disponibilizado para a respectiva aplicação, em detrimento das cargas externas. No entanto, é importante lembrar que quanto maior o número de processos, maior a troca de contexto, o que acarreta gradativamente uma piora do desempenho em situações onde a aplicação recebe a mesma porcentagem de CPU. Este tipo de comportamento pode ser claramente notado na coluna `10s` da Tabela 4.9. Quando executamos o *EasyGrid AMS* com apenas um processo e uma carga externa (cenário (1), onde 50% da CPU fica disponível para o *EasyGrid AMS* quando executamos apenas um processo por vez), o tempo final de execução da aplicação é de 10054,73 segundos. Quando se executa com dois processos e duas cargas externas (cenário (2), onde 50% da CPU fica disponível para o *EasyGrid AMS*, sendo dois processos executados por vez) o tempo final da aplicação aumenta para 10071,56 segundos e com três processos e três cargas o tempo aumenta para 10111,30 segundos(cenário (3), onde 50% da CPU fica disponível para o *EasyGrid AMS*, sendo três processos executados por vez).

Um segundo experimento, utilizando a mesma configuração de máquinas do experimento anterior, foi realizado com a intenção de avaliar o grau de intrusão do *middleware EasyGrid* ao executar aplicações de mesma duração, porém, com número de tarefas distintos e com pesos distintos. Nesse experimento foram utilizadas aplicações PSwp com 5.000, 1.000, 100 e 10 tarefas, onde cada tarefa tem a duração de 0,2, 1, 10 e 100 segundos, respectivamente. Logo, o tempo ótimo (sem sobrecarga) de execução de cada aplicação em apenas um processador dedicado seria de 1.000 segundos. Cada uma dessas aplicações foi executada em:(-) uma máquina dedicada; (1) com uma carga externa; (2) com duas cargas externas; (3) e com três cargas externas. Os resultados desse experimento se encontram na Tabela 4.10.

Tabela 4.10: Tempo de Execução (em segundos)

		Total de Tarefas x Duração da Tarefa			
cenário	#processos	5.000 × 0,2	1.000 × 1	100 × 10	10 × 100
(-)	1	1.102,5	1.022,4	1.005,0	1.003,6
	2	1.043,2	1.012,1	1.005,8	1.005,3
(1)	1	1.440,3	2.030,0	2.011,5	2.006,2
	2	1.292,7	1.521,2	1.509,8	1.507,4
(2)	1	1.881,9	3.050,1	3.019,2	3.008,1
	2	1.500,2	2.030,6	2.014,1	2.009,9
(3)	1	2.170,2	4.072,5	4.025,8	4.011,0
	2	1.856,9	2.544,2	2.518,9	2.512,0

Assim como no experimento anterior, para tarefas maiores ou iguais a 10s a utilização de apenas um processo executando por vez é mais vantajosa para ambientes dedicados. Porém, ao utilizar dois ou mais processos o *EasyGrid* consegue uma maior porcentagem da CPU ao executar em ambientes compartilhados. Tarefas com tempo de execução pequeno

(próximo ao *time slice* do sistema operacional) praticamente não são afetadas pela execução compartilhada com outros programas. Por serem criadas dinamicamente, as tarefas rapidamente ganham a CPU e, devido ao seu tamanho, praticamente não sofrem preempção. Por outro lado, a intrusão do *middleware* é maior para tarefas pequenas. Em um ambiente dedicado, linha (-) da Tabela 4.10, os resultados indicam que quanto menor o número de tarefas melhor o desempenho da aplicação. Isso acontece, pois quanto maior o número de tarefas, maior quantidade de trabalho o *middleware* tem que realizar, aumentando sua sobrecarga. Entretanto, em média, essa sobrecarga corresponde somente a 2% do tempo necessário para executar a aplicação, e diminui com o aumento do peso das tarefas [119, 120], como mostrado no início desta subseção. Além disso, a utilização de uma grande quantidade de tarefas pequenas, ao invés de poucas tarefas grandes, permite que o balanceamento de carga possa ser feito através da distribuição dos processos entre os recursos por um gerenciador externo [18, 118] (transparente para o usuário) e simplifica o tratamento de falhas, sendo necessário reexecutar apenas os pequenos processos que falharem [119], como será avaliado na próxima seção.

4.4 Avaliação do Modelo 1PTask para Aplicações *bag-of-tasks*

Esta seção avalia o modelo proposto nesta tese através da execução de aplicações paralelas em uma grade computacional. As vantagens do modelo alternativo são destacadas utilizando experimentos que comparam a execução de aplicações através dos modelos: 1PProc e 1PTask. Até o momento, o único sistema gerenciador (que se tem conhecimento) para execução de aplicações MPI nas grades computacionais, que adota o modelo de execução 1PTask, é o *middleware EasyGrid AMS* (Seção 4.1.1), que foi criado com o objetivo de simplificar as tarefas do programador e facilitar a execução de aplicações MPI nas grades computacionais.

Os experimentos que serão apresentados comparam o desempenho da abordagem proposta com o desempenho do modelo tradicional, através da execução de aplicações MPI. Em todos os casos, primeiramente é feita uma avaliação em ambientes dedicados e homogêneos (para os quais as aplicações MPI tradicionais foram desenvolvidas) com o intuito de avaliar a sobrecarga oriunda do modelo de execução proposto nesta tese, e com isso permitir a comparação com outros gerenciadores para grades. Além disso, em vários experimentos a execução das aplicações MPI tradicionais é feita de maneira que utilize os recursos que maximizam sua execução ou a distribuição das tarefas é feita levando-se em conta a heterogeneidade dos recursos melhorando o seu desempenho.

Esta seção avalia o modelo 1PTask especificamente para as aplicações *bag-of-tasks*

que, por serem compostas de tarefas que não se comunicam entre si, são amplamente executadas nas grades computacionais. No próximo capítulo o modelo 1PTask é validado para as aplicações paralelas compostas de tarefas que se comunicam, aplicações estas que ainda não foram executadas eficientemente em ambientes grades.

É importante destacar que o modelo de aplicação e o modelo arquitetural definidos nesta tese têm fundamental importância nesta seção, pois são através deles que as aplicações e os ambientes de execução são representados, respectivamente. Desse modo, esses dois modelos adotados influenciam diretamente no desempenho das aplicações e consequentemente nos resultados obtidos.

4.4.1 Aplicações Utilizadas

Primeiramente, são avaliadas aplicações do tipo mestre/trabalhador sob os modelos de execução 1PProc e 1PTask. Esse tipo de aplicação foi escolhido, especialmente, por três motivos:

1. É um paradigma comumente adotado na implementação de programas paralelos;
2. Já existem várias versões para esse paradigma, inclusive versões que ajustam dinamicamente a carga enviada para cada processador em ambientes heterogêneos, de modo que é possível comparar a estratégia proposta nesta tese com soluções já existentes;
3. Esse tipo de aplicação tem um padrão de comunicação que sobrecarrega a hierarquia do *EasyGrid AMS* devido a grande quantidade de comunicações no início (todas as tarefas recebem os dados do mestre) e assim que cada tarefa termina (envia a resposta para o mestre).

A aplicação mestre/trabalhador analisada, denotada como MWork, é uma versão sintética onde a quantidade de trabalho executada por cada processo MPI é controlada para que se avalie o desempenho em relação a granularidade da aplicação. A quantidade mínima de trabalho executada por cada trabalhador é chamada de unidade de trabalho.

Em implementações tradicionais mestre/trabalhador em MPI, onde é criado estaticamente um processo MPI por processador [61], um processo mestre distribui unidades de trabalho entre os processos trabalhadores disponíveis. Nesse trabalho de tese são consideradas três implementações distintas:

1. estática (S) - onde o processo mestre distribui todas as unidades de trabalho igualmente entre os processos trabalhadores, assumindo que eles estão executando em recursos homogêneos;

2. estática avançada (SE) - onde o processo mestre distribui todas as unidades de trabalho entre os processos trabalhadores, proporcionalmente ao poder computacional de cada recurso;

3. sob demanda (OD) - onde o processo mestre distribui unidades de trabalho à medida que os recursos ficam ociosos;

Na versão MWork gerenciada pelo *EasyGrid AMS*, cada processo trabalhador executa apenas uma unidade de trabalho. Embora o número de total de processos trabalhadores seja pré-definido e sua alocação definida antes da execução pelo escalonador estático do *EasyGrid AMS*, cada processo trabalhador MPI é criado dinamicamente no recurso considerado mais apropriado pelos gerenciadores local e global, no tempo determinado pelo gerenciador de recurso da máquina.

Além da aplicação sintética, uma aplicação paralela SPMD real que computa a dispersão térmica macroscópica em meio poroso (aplicação *Thermions*) [142] é também analisada. Dado que um meio poroso é composto por elementos sólidos e fluidos, a dispersão termal é avaliada através do movimento de um número grande de partículas hipotéticas, chamadas *Thermions*, a partir de um ponto de início fixo, através do meio. A posição, energia, um componente aleatório e as propriedades térmicas dos sólidos ou a velocidade do fluido determina a distância viajada dentro de um período de tempo, por cada *Thermion*. O custo computacional para calcular o caminho para cada *Thermion* varia e não pode ser determinado com precisão antes da execução. Para esta aplicação, a unidade de trabalho corresponde ao cálculo do caminho de um único *Thermion*. Duas versões da aplicação *Thermions* em MPI são avaliadas: sob-demanda, chamada de *Thermions* MPI ; e uma versão executada através do *EasyGrid AMS*, chamada de *Thermions* AMS.

4.4.2 Ambiente de execução

Os experimentos foram realizados em uma grade geograficamente distribuída, composta de quatro *sites* interconectados através de *switches gigabit e fast Ethernet*. Todos os processadores disponíveis executam Linux Fedora Core 2, Globus Toolkit 2.4 e LAM/MPI 7.0.6. *Sites* 1, 2 e 3 são compostos de 28 processadores Pentium IV 2.6 GHz e 3 processadores Pentium IV 3.2 GHz (dois no *Site* 2 e um no *Site* 3), onde *Site* 1 e 3 contém 8 processadores cada e *Site* 2, 15 processadores. Site 4 é composto de 22 processadores Pentium II 400 MHz. Os *Sites* 1, 2 e 3 estão localizados na Universidade Federal Fluminense, enquanto que o Site 4 se encontra localizado na PUC-Rio.

Do total de 53 recursos disponíveis para execução das aplicações 52 executaram processos trabalhadores, enquanto que um recurso localizado no Site 2 foi escolhido para

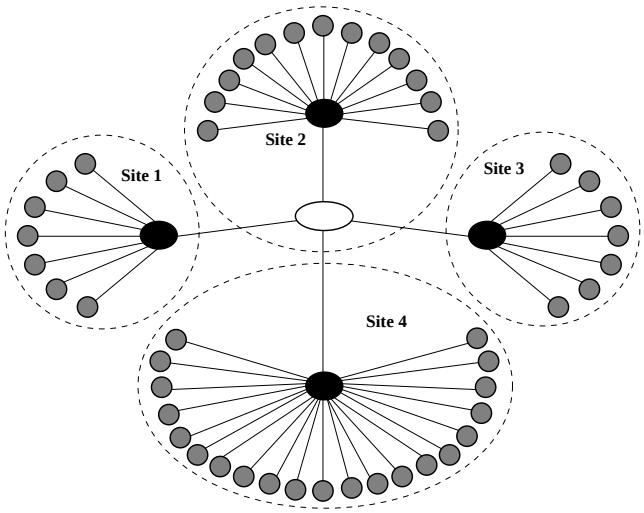

Figura 4.6: Ambiente grade para *EasyGrid AMS*

executar o processo mestre. A estrutura de gerenciadores do *EasyGrid AMS* é apresentada na Figura 4.6. A elipse branca executa apenas o GM e o processo mestre, enquanto que cada um dos recursos restantes executa um processo HM que gerencia a criação e execução de cada processo trabalhador alocado no seu respectivo recurso. Além disso, cada recurso destacado na cor preta executa também um SM responsável pela execução no Site. Desse modo, os recursos em preto tem uma sobrecarga extra acarretada pela execução de dois processos gerenciadores (SM e HM).

4.4.3 Análise da Execução em Recursos Heterogêneos

O objetivo do primeiro conjunto de experimentos é analisar o impacto da heterogeneidade dos recursos no desempenho das quatro versões MPI implementadas. Três cenários são comparados:

(1) representa um cluster dedicado e praticamente homogêneo, onde somente recursos do Site 2 são utilizados;

(2) representa uma grade computacional heterogênea, porém estática (sem mudança de carga), compostas pelos recursos dos Sites 1,2 e 3;

(3) representa uma grade heterogênea e dinâmica, também composta pelos recursos dos Sites 1,2 e 3.

Como os recursos de todos os Sites utilizados têm quase o mesmo poder computacional, heterogeneidade foi introduzida através da execução de cargas externas concorrentemente com as aplicações MWork. Os recursos do Site 2 fornecem 100% do seu poder

computacional para a aplicação, enquanto que os Sites 1 e 3 fornecem apenas 25% e 50%, respectivamente. Para aumentar a heterogeneidade dentro dos sites, duas máquinas de cada site foram escolhidas para executar cargas externas extra, de modo que essas máquinas fornecem apenas metade do poder computacional das máquinas de seu respectivo site. No cenário (3) essas cargas extra migram para diferentes máquinas durante a execução da aplicação MWork, representando o comportamento dinâmico da grade.

A Tabela 4.11 apresenta a média aritmética do tempo final de três execuções para cada implementação MWork, onde cada implementação contém 500 unidades de trabalho de mesmo peso. Três instâncias com unidades de trabalho de pesos diferentes foram testadas, equivalentes a 5, 10 e 20 segundos em uma máquina Pentium IV 2.6 GHz dedicada.

Os resultados obtidos para o cenário (1) mostram que mesmo em um ambiente homogêneo e dedicado, totalmente favorável ao modelo de execução MPI 1PProc, a versão MWork AMS alcança muito bons tempos de execução considerando a sobrecarga para gerenciar cada processo trabalhador, criando um por vez em cada processador, assim como o monitoramento da aplicação. A diferença foi no máximo de 1% comparando com os melhores valores produzidos pelo MWork MPI. As implementações S e SE obtiveram os mesmos resultados, pois, por se tratar de um ambiente homogêneo, a distribuição das tarefas foi a mesma.

Tabela 4.11: Comparação dos tempos de execução obtido pelas implementações *MWork* MPI e *MWork* AMS (em segundos)

cenário cluster	unidade de trabalho	*MWork MPI*			*MWork*
		S	SE	OD	AMS
(1)	5s	180,4		180,3	181,9
	10s	359,6		359,0	362,7
	20s	718,5		717,1	723,8

cenário grid	unidade de trabalho	*MWork MPI*			*MWork*
		S	SE	OD	AMS
(2)	5s	643,8	140,4	158,0	141,8
	10s	1312,3	279,9	316,4	282,6
	20s	2632,9	559,6	634,8	563,9

cenário grid	unidade de trabalho	*MWork MPI*			*MWork*
		S	SE	OD	AMS
(3)	5s	492,0	210,1	157,8	142,7
	10s	986,2	420,7	317,1	282,9
	20s	1943,1	838,2	633,5	573,4

No cenário (2), os melhores tempos de execução foram alcançados pela versão MWork SE-MPI e MWork AMS. Nesse caso, como o ambiente é estático e dedicado, as estimativas utilizada pela versão MWork SE-MPI permitem que o algoritmo divida a carga (antes da execução) entre os recursos disponíveis de maneira ótima e, desse modo, obtenha um bom

desempenho. Entretanto, é importante ressaltar que esta informação deve ser fornecida pelo programador ou algum outro *middleware* capaz de verificar o poder computacional dos recursos disponíveis e o mestre deve ser programado para balancear a distribuição das unidades de trabalho. Porém, todo esse trabalho extra não acarreta uma execução eficiente quando o ambiente é dinâmico, como mostra o cenário (3). Nesse cenário, comum em ambientes grades, os melhores resultados foram alcançados pela versão MWork AMS em razão da sua capacidade de redistribuir dinamicamente as unidades de trabalho de acordo com as mudanças ocorridas. Com relação as implementações que utilizam o modelo de execução 1PProc, MWork OD-MPI alcançou o melhor resultado por ser capaz de ajustar sua execução a variações no poder computacional dos recursos, ela manda as tarefas à medida que os recursos se tornam ociosos.

Por ter alcançado o melhor desempenho entre as implementações MPI para plataformas grades, a versão MWork OD-MPI será usada nos experimentos a seguir para ser comparada com a versão MWork AMS.

Um segundo experimento foi realizado utilizando todos os quatro sites da grade. Além disso, com o intuito de aumentar a heterogeneidade, cargas externas extras foram executadas em quatro máquinas distintas (uma no Site 1, uma carga no Site 3 e duas no Site 4). É importante ressaltar que esse experimento avalia apenas a heterogeneidade da grade pois não há variação no poder computacional dos recursos durante a execução da aplicação. Os resultados desse segundo experimento são apresentados na Tabela 4.12 através da média aritmética do tempo final de três execuções da aplicação MWork com 1.000 unidades de trabalho não uniformes, onde 50% das unidades de trabalho requerem 20 segundos de computação, 25% 10 segundos e as restantes 25% requerem 5 segundos.

Tabela 4.12: Comparação dos tempos de execução alcançados pela aplicação *MWork* com tarefas não uniformes (em segundos)

Random *MWork* OD-MPI	*Sorted MWork* OD-MPI	*MWork* AMS
584,4	538,4	438,4

Duas implementações MWork OD-MPI são consideradas. A primeira, chamada de Random MWork OD-MPI, não tem conhecimento sobre a distribuição da carga. Nesse caso, cinco ordenações distintas foram especificadas e cada uma executada três vezes e a média aritmética calculada. Na segunda versão, chamada de Sorted MWork OD-MPI, as unidades de trabalho são distribuídas pelo mestre seguindo uma ordenção decrescente em relação ao seu custo computacional. Na versão MWork AMS, os escalonadores estático e dinâmico ordenam as tarefas (unidades de trabalho) decrescentemente em relação ao seu custo computacional. Os resultados da Tabela 4.12 confirmam os resultados apresentados

no experimento anterior, ou seja, mostram que o tempo de execução médio alcançado pelas duas versões MWork OD-MPI foram piores do que os obtidos pela versão MWork AMS, pois apesar das versões MWork OD-MPI conseguirem se ajustar a variações do ambiente, a tarefa só é cedida quando o recurso se torna ocioso. A falta de informação sobre a aplicação e os recursos faz com que essa distribuição seja ineficiente, prejudicando o desempenho da aplicação.

Baseado no modelo alternativo 1PTask, a aplicação MWork AMS é capaz de adaptar sua execução a heterogeneidade do ambiente, inclusive à própria não uniformidade da aplicação (que é composta de tarefas de tamanhos variados), conseguindo uma utilização dos recursos apropriada que permite um desempenho eficiente. O mesmo não pode ser dito das versões MWork OD-MPI, uma vez que elas adotam uma abordagem gulosa (que entrega a próxima unidade de trabalho ao trabalhador que termina primeiro, independentemente da capacidade do recurso que ele está executando) minimizando tempo de ínicio da tarefa ao invés do tempo de fim. É possível implementar versões dessa aplicação, baseada no modelo MPI padrão 1PProc, mais inteligentes, porém as especificidades provenientes de um ambiente grade não deveriam ser responsabilidade do programador.

O próximo experimento, executado no mesmo ambiente do experimento anterior, investiga a escalabilidade do modelo através de uma aplicação de larga escala real. Os resultados, apresentados na Tabela 4.13, foram obtidos através da execução da aplicação *Thermions*, variando o número de tarefas (N). É difícil prever com exatidão o custo computacional de cada *thermion*, mas varia entre 4 e 4,5 segundos em uma máquina dedicada de 2.6 GHz. A versão *Thermions AMS* tem que gerenciar mais de N processos (incluindo os processos gerenciadores). Os resultados mostram que o modelo 1PTask é competitivo nesse ambiente estável, conseguindo resultados levemente melhores que o modelo 1Pproc.

Tabela 4.13: Comparação entre os tempos de execução das versões OD-MPI e AMS para a aplicação *Thermions* (em segundos)

N	*Thermions* OD-MPI	*Thermions* AMS
1000	136,1	130,7
3000	377,2	373,2
5000	619,9	615,8
10.000	1.355,6	1.352,6

4.4.4 Dinamismo da Grade x Políticas Distintas

Com o objetivo de avaliar o desempenho dos dois modelos de execução em ambientes dinâmicos, o seguinte experimento foi executado, utilizando a mesma grade do experimento anterior (grade com quatro sites) com políticas de uso de recursos distintas. Para estes

testes, os recursos dos Sites 2 e 3 estavam dedicados ao experimento, enquanto que os recursos dos Sites 1 e 4 estavam compartilhados com outras aplicações. A política de uso de recursos dos Sites 1 e 4 somente permite aceitar *jobs* da grade quando eles estiverem ociosos.

Neste experimento semi-controlado, inicialmente foi executada uma carga extra em cada um dos recursos dos Sites 1 e 4, de maneira que somente os recursos dos Sites 2 e 3 estivessem disponíveis para a aplicação *Thermions*. Além disso, na média, 10 máquinas do Site 4 estavam sendo utilizadas por usuários locais. A versão *Thermions OD-MPI* pode executar somente nos Sites 2 e 3, uma vez que seus processos são criados estaticamente no início da aplicação somente nos recursos disponíveis. Em um determinado instante da execução de cada instância da aplicação *Thermions*, todos as cargas extra executadas inicialmente nos Sites 1 e 4 terminam. Os processos gerenciados pela versão *Thermions AMS* são capazes de detectar todos os recursos do Site 1 e do Site 4 que se tornaram ociosos, e capazes de redistribuir parte dos processos trabalhadores para estes recursos.

A Tabela 4.14 apresenta a média do tempo de três execuções obtidos pela aplicação *Thermions* composta de 1.000 unidades de trabalho. O resultado para a versão *Thermions OD-MPI* é apresentado na primeira coluna, enquanto que os resultados alcançados pela versão *Thermions AMS*, através de quatro cenários distintos, são apresentados nas demais colunas. Cada cenário indica o tempo aproximado que os recursos compartilhados ficaram disponíveis para os processos da aplicação *Thermions*, contando a partir do início da sua execução:

(1) logo após o início da aplicação;

(2) após 50 segundos do início;

(3) após 100 segundos;

(4) nunca.

Tabela 4.14: Comparação dos tempos de execução alcançados pela aplicação *Thermions* em um ambiente dinâmico com diferentes políticas de acesso a recursos (em segundos)

Thermions OD-MPI	*Thermions* AMS (1)	(2)	(3)	(4)
197,9	139,8	152,6	170,2	199,3

O tempo de execução da versão *Thermions OD-MPI* foi melhor do que a versão *Thermions AMS* somente no cenário (4), onde a grade disponível (recursos dos Sites 2 e 3 somente) é ideal para o modelo MPI padrão 1Pproc: estável, dedicado e homogêneo. Como

nos experimentos anteriores, a diferença de 1% se deve a sobrecarga do AMS para inicializar a estrutura de gerenciamento e implementar o modelo de execução alternativo. Por outro lado, os resultados obtidos pela aplicação *Thermions AMS* nos cenários 1, 2 e 3 mostram que o *middleware* é capaz de lidar com o dinamismo e compartilhamento das grades e se adaptar a diferentes políticas de acesso a recursos. Repare que o cenário (1) apresenta um tempo de execução maior do que o apresentado na primeira linha da Tabela 4.13 (somente 2,7% maior). Esta pequena sobrecarga se deve a necessidade do gerenciador da máquina do *EasyGrid AMS* (HM) ter que detectar e estar certo que cada recurso está realmente ocioso para que ele possa ser utilizado. Para isso, o HM verifica duas vezes se existe algum processo local executando durante um intervalo de tempo o que causa a sobrecarga.

4.4.5 Falha de Processos nas Grades

Ambientes grades são mais sujeitos a falhas do que ambientes de computação de alto desempenho tradicionais como os supercomputadores e *clusters*. O objetivo do próximo experimento é avaliar o desempenho da aplicação MWork AMS (que utiliza o modelo alternativo 1PTask) quando múltiplas falhas de processos ocorrem. É importante destacar que nesses experimentos é utilizado o mecanismo proposto por [45]. Esta análise destaca os benefícios de recriar trabalhadores dinamicamente e escaloná-los cuidadosamente. Nos experimentos, quando um processo falha, sua execução será reiniciada desde o começo, uma vez que nenhum mecanismo de *checkpointing* é considerado. Dois experimentos foram realizados para investigar os benefícios do modelo de execução alternativo na tolerância a falhas de processos da aplicação.

O primeiro experimento investiga a sobrecarga de se utilizar um mecanismo de tolerância a falhas baseado em *log* de mensagens, através da execução das aplicações MWork e *Thermions*. Para isto, foram utilizadas 30 máquinas disponíveis na UFF com uma configuração diferente da Figura 4.6, formada por apenas um *site*, onde uma máquina executava o gerenciador global *GM*, uma máquina o gerenciador local *SM* e as outras 28 máquinas executavam o gerenciador da máquina *HM*. O tempo de duração das tarefas da aplicação MWork é de 5 segundos e da aplicação *Thermions* varia entre 4 e 4,5 segundos, considerando um processador Pentium IV 2.6 GHz dedicado. Além disso, foram testadas aplicações MWork com 1.000 e 10.000 tarefas e a aplicação *Thermions* com 1.000 tarefas. Três cenários foram analisados:

(1) nenhuma tarefa falhou durante a execução da aplicação;

(2) 10% das tarefas da aplicação falharam;

(3) 20% das tarefas da aplicação falharam.

Neste experimento, os processos falharam (através de um *script*, para simular as falhas) em quatro dos 28 recursos disponíveis para executar tarefas da aplicação. Para isso, o comando *kill* foi utilizado com intervalo de 6,5 segundos para evitar que a mesma tarefa fosse abortada. Este período de tempo é mais do que suficiente para recuperar e reexecutar um processo que falhou.

A Tabela 4.15 mostra o tempo real de execução (R), um valor estimado (E) que considera apenas o tempo de execução de cada tarefa e o percentual de sobrecarga (S) do mecanismo de tolerância a falhas, para as aplicações MWork e *Thermions*, respectivamente. Apesar da regularidade na execução do comando *kill* é muito difícil garantir que os processos vão ser finalizados exatamente no mesmo tempo. Para calcular o tempo estimado (E), é assumido que, na média, as tarefas são finalizadas após completarem metade do seu tempo computacional e que um balanceamento de carga perfeito para as tarefas da aplicação (por exemplo, a carga computacional adicional, criada em consequência da reexecução dos processos, é distribuída igualmente entre todos os processadores disponíveis).

Os resultados da Tabela 4.15 mostram que a sobrecarga para o mecanismo de tolerância a falhas, através do modelo de execução 1PTask é sempre menor do que 7%. Mais importante, esse percentual diminui para um número maior de tarefas, pois o escalonador de tarefas consegue distribuí-las melhor. Esta sobrecarga pode ser considerada baixa, principalmente levando-se em conta que ela inclui a detecção do erro e recuperação [45], além dos custos dos gerenciadores do *EasyGrid AMS* (GM, SM e HM) [119]. Além disso, o tempo esperado considera também que os mecanismos de balanceamento de carga e tolerância a falhas são perfeitos (geram resultados ótimos).

Tabela 4.15: Comparação entre o tempo de execução real (R) e estimado (E) em um ambiente com falhas. A terceira coluna apresenta a sobrecarga do mecanismo de tolerância a falhas.

Cenários	MWork						Thermions		
	N = 10.000			N = 1.000			N = 1.000		
	R	E	S	R	E	S	R	E	S
(1)	1808,2	1790,0	1,0%	183,2	180.00	1,8%	161.9	158.4	2,2%
(2)	1929,2	1875,0	2,9%	197.2	187.50	5,2%	172.5	165.0	4,6%
(3)	2043,3	1964,3	4,0%	209.2	196.43	6,5%	184.9	172.9	7,0%

O segundo experimento avalia, principalmente, o impacto do momento em que a falha ocorre no desempenho da aplicação. Os tempos de execução a seguir foram obtidos através da execução das aplicações MWork AMS e MWork OD-MPI com 1.000 unidades de trabalho de 5 segundos cada, no ambiente grade completo (Figura 4.6). Os resultados obtidos para execução sem falhas foram 164,00 e 202,65 segundos, respectivamente. Três cenários com falhas foram então avaliados:

- 1% dos processos falharam (10 processos);
- 3% dos processos falharam (30 processos);
- 5% dos processos falharam (50 processos)

Para simular as falhas nos processos, um *script* foi especificado para interromper a execução da quantidade de processos relativa a cada cenário em cinco recursos diferentes nos Sites 1, 2 e 3. Além disso, o número total de processos que falharam foi igualmente distribuído entre esses cinco recursos. Duas situações diferentes foram consideradas com relação ao momento em que se inicia a falha nos processos (momento em que os processos são mortos): 50 segundos após o início da aplicação e 100 segundos após o início da aplicação. Os processos em cada um dos recursos escolhidos foram interrompidos (*killed*) utilizando intervalos de 6 segundos. Desse modo, por exemplo, se um total de 50 processos falharem (10 por recurso) a aplicação irá sofrer falhas de processos durante 60 segundos aproximadamente em cinco recursos.

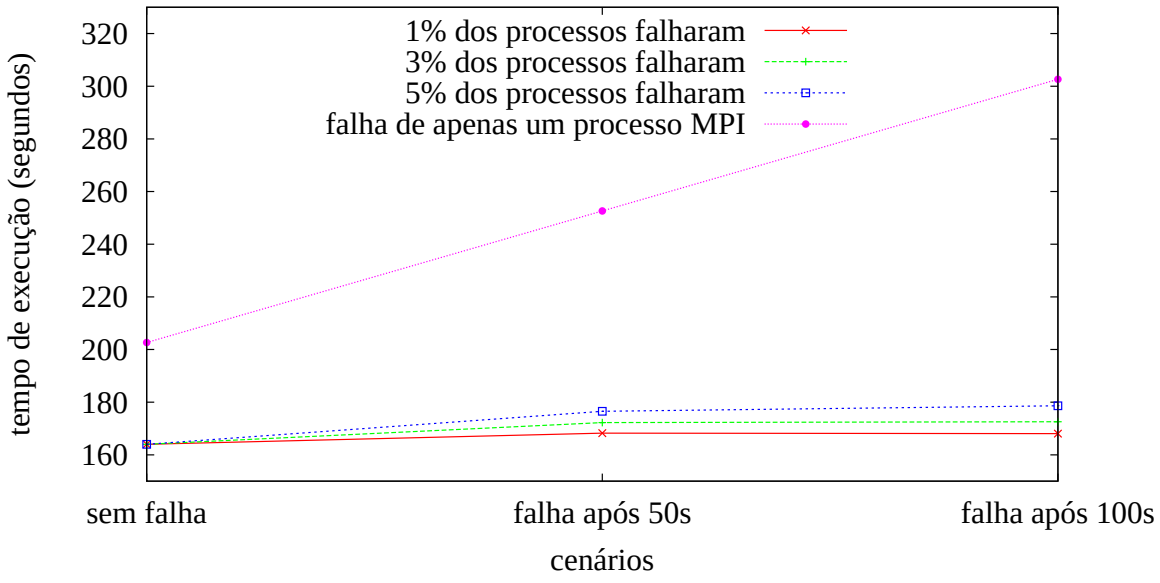

Figura 4.7: Desempenho da aplicação MWork na presença de falhas.

A Figura 4.7 apresenta o desempenho para cada um dos três cenários. Duas importantes características do mecanismo de tolerância a falhas embutido na aplicação MWork AMS, baseado no modelo de execução 1PTask, podem ser destacadas. Primeiramente, o custo de reexecutar processos que falharam é muito pequeno, proporcional a quantidade de falhas e a média de trabalho perdido (2,5 segundos). Embora exista uma sobrecarga nos recursos onde as falhas ocorrem, o tempo extra gasto na reexecução de processos que falharam é bem distribuído pelo escalonador dinâmico. Segunda característica identificada,

os resultados mostram que o desempenho da aplicação não depende do momento em que a falha ocorre, desde que o escalonador dinâmico tenha tempo para balancear a carga igualmente. Mesmo em uma situação extrema (5% dos processos falharem após 100 segundos), quando a última falha de processo ocorre quase no fim da execução, o aumento no tempo de execução foi menor do que 1,1%. A Figura 4.7 também apresenta o desempenho teórico da versão MWork OD-MPI onde toda a aplicação tem que ser reexecutada caso pelo menos um processo falhe.

4.5 Resumo

Este capítulo apresenta a validação do modelo alternativo de execução, proposto neste trabalho de tese, para aplicações *bag-of-tasks*. Para isso foi utilizado o sistema gerenciador de aplicações MPI *EasyGrid AMS*, desenvolvido baseado no modelo de execução proposto nesta tese, 1PTask, e que tem como função controlar a execução de aplicações MPI nas grades computacionais.

A validação foi dividida em três partes: análise e validação do modelo GAD para aplicações *bag-of-tasks*; avaliação do modelo de computação; avaliação do modelo alternativo de execução para grades. Primeiramente foi apresentado uma análise das aplicações *bag-of-tasks*. A segunda parte deste capítulo avalia e analisa o modelo de computação proposto. Por último, é apresentada a validação do modelo 1PTask para aplicações *bag-of-tasks*. Os resultados apresentados mostram que o modelo proposto permite que a aplicação se adapte as variações do ambiente, utilizando de maneira mais apropriada os recursos, aumentando o seu desempenho. Por outro lado, as aplicações que utilizam o modelo tradicional 1PProc não conseguem se adaptar, o que prejudica o seu desempenho.

Capítulo 5

Validação do Modelo Alternativo para Aplicações Paralelas

O objetivo deste capítulo é avaliar e validar o modelo de execução alternativo proposto para as aplicações paralelas cujas tarefas se comunicam entre si. Inicialmente é definido o modelo de comunicação que melhor representa as trocas de mensagens entre os processos. Em seguida, é apresentado um estudo de caso da aplicação N-corpos *ring*. Este estudo apresenta uma análise do algoritmo N-corpos *ring* desenvolvido para executar em ambientes estáticos e homogêneos, através do modelo de execução tradicional. A partir das limitações desse algoritmo, é proposta uma implementação alternativa baseada no modelo de execução 1PTask, juntamente com uma estratégia inovadora para execução eficiente de aplicações fortemente acopladas nas grades.

Finalizando este capítulo, baseado na estratégia proposta, é apresentada a viabilidade da execução do algoritmo N-corpos *ring* em ambientes reais heterogêneos e não dedicados, mostrando de maneira real a possibilidade de se executar aplicações fortemente acopladas nas grades computacionais.

5.1 Avaliação do Modelo de Comunicação

Na Seção 4.3 foi analisado e avaliado o modelo de computação proposto para representar as características dos processadores das grades. Esta seção, define como representar as trocas de mensagens entre processos executando através do modelo 1PTask.

Na subseção 3.3.2 foi apresentado um resumo dos principais modelos arquiteturais existentes na literatura e, baseado no modelo de execução alternativo, foram identificadas as principais características a serem representadas em um modelo de comunicação para grades computacionais. A utilização de um modelo que represente de maneira adequada

as características que influenciam o tempo de comunicação entre dois processos, é fundamental para o escalonamento de tarefas de programas paralelos. Esse modelo deve não só representar o custo da troca de mensagens entre processos em um mesmo processador e entre processadores distintos, mas também as diferentes características de uma grade computacional, ambiente este que apresenta canais com latências de comunicação variadas, principalmente devido à distância geográfica que pode existir entre recursos de uma grade. Além disso, como a execução das aplicações nas grades vem sendo feita através de sistemas gerenciadores que tem o objetivo de facilitar o acesso e otimizar o desempenho da aplicação, o modelo proposto deve ser capaz de representar a influência (ou característica) desses sistemas gerenciadores na comunicação entre as tarefas, caso essa característica seja determinante no desempenho da aplicação.

Esta seção utiliza o sistema gerenciador de aplicações *EasyGrid AMS* para validar as características identificadas na subseção 3.3.2 e, a partir delas, definir um modelo de comunicação a ser utilizado para execução de aplicações utilizando o modelo alternativo em ambientes grades.

O *middleware EasyGrid AMS*, como explicado na seção 4.1.1, é um sistema gerenciador de aplicações para programas MPI. Esse controle é feito através de processos gerenciadores, em três níveis distintos, onde uma das funções desses processos é o encaminhamento das mensagens entre duas tarefas da aplicação. Deste modo, a comunicação entre dois processos de um programa MPI executando através do *EasyGrid* é mais complexa do que entre dois processos de um programa MPI executando sem o *middleware*. Seguindo a hierarquia do *EasyGrid* (Seção 4.1.1), se dois processos se encontram em um mesmo processador, então a mensagem é encaminhada através do respectivo HM. Caso as mensagens se encontrem em processadores distintos, mas pertencentes a um mesmo *site*, então o encaminhamento é feito através do HM de cada máquina e do gerenciador do *site*, SM. Por último, caso os processos se encontrem em máquinas localizadas em *sites* distintos, então o encaminhamento é feito através do HM de cada máquina, do SM de cada *site* e do gerenciador global (GM). A Figura 5.1 apresenta um esboço desse mecanismo.

Uma descrição mais detalhada do encaminhamento de mensagens no *middleware EasyGrid* se encontra a seguir. O processo v_1 envia uma mensagem para v_2, que é interceptada pelo gerenciador da máquina (HM). O HM, ao receber esta mensagem, verifica qual é o destino final desta, processo v_2, e se este processo está alocado para ser executado no mesmo recurso ou em um outro. Se estiver alocada no mesmo recurso, o HM pode então criar esse processo (quando todas as mensagens necessárias ao processo já tiverem sido recebidas pelo HM) e enviar a mensagem para v_2 (Figura 5.1 (a)). Caso o processo não esteja alocado no mesmo processador, o HM envia a mensagem para o gerenciador do *site*

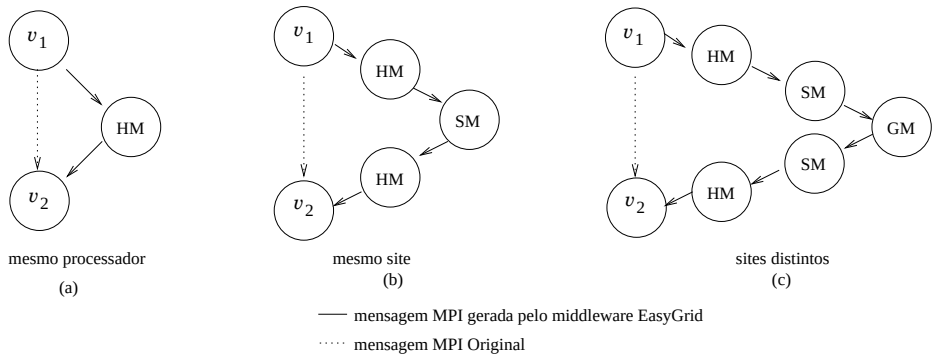

Figura 5.1: Comunicação entre dois processos MPI, através do *EasyGrid AMS*.

(SM) que, ao recebê-la, envia para o seu destino final, caso esse processo vá executar em algum dos seus recursos (Figura 5.1 (b)). Caso contrário, o SM envia para o gerenciador global (GM) que transmite a mensagem para seu destino final seguindo a hierarquia do *EasyGrid AMS* (Figura 5.1 (c)).

Sobrecargas de Comunicação no *EasyGrid AMS*

Conforme descrito nos trabalhos [4, 43, 110], antes de uma mensagem ser enviada ou recebida existe um custo (para preparar a mensagem) que incide sobre o processador. Com o objetivo de identificar esse custo de sobrecarga entre dois processos MPI que se comunicam através do *EasyGrid AMS*, o seguinte experimento foi realizado. Inicialmente, uma aplicação MPI composta por apenas dois processos dependentes conforme o GAD apresentado na Figura 5.2 (a) (tanto v_1 quanto v_2 têm 1 segundo de duração), foi executada através do *EasyGrid AMS* em um mesmo recurso, como visto na Figura 5.2 (b). O *timestamp* do início e término de cada um dos processos, assim como o tempo de CPU, foram coletados utilizando a camada de monitoramento do *EasyGrid AMS*. A partir desses dados foram identificadas as seguintes sobrecargas: envio, criação e recebimento.

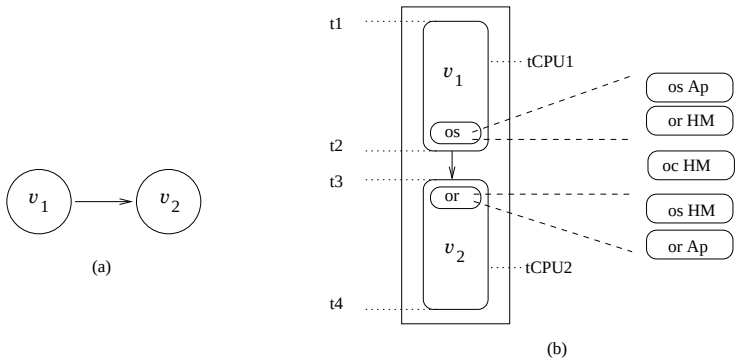

Figura 5.2: Sobrecargas de comunicação entre dois processos MPI executando em um mesmo processador através do EasyGrid AMS.

A sobrecarga de envio de uma mensagem da aplicação é denotada por o_s, enquanto que a sobrecarga de recebimento e de criação são denotadas por o_r e o_c, respectivamente. De acordo com a Figura 5.2 (b), a sobrecarga de envio o_s é a diferença entre o tempo de execução total do processo v_1 $(t2 - t1)$ e o tempo de execução de v_1 sem enviar mensagens que é de 1 segundo, conforme especificado na Equação 5.1.

$$o_s = (t2 - t1) - 1 \tag{5.1}$$

Baseado no *middleware EasyGrid*, o_s corresponde na verdade à sobrecarga do envio a partir da tarefa da aplicação, chamada de o_sAP, seguido da sobrecarga de recebimento da mensagem pelo respectivo gerenciador HM, denotado por o_rHM.

Ao receber a mensagem, o HM cria o processo v_2, incorrendo então o custo de criação o_c, calculado através da diferença entre o tempo de início de v_2 e o tempo de término de v_1, especificado na Equação 5.2.

$$o_c = t3 - t2 \tag{5.2}$$

Por último, a sobrecarga de recebimento o_r é calculada (Equação 5.3) através da diferença entre o tempo de execução total de v_2 (t4 - t3, como mostra a Figura 5.2) e o tempo de processamento sem envio de mensagens. No entanto, análogo a o_s, a sobrecarga o_r é na verdade composta da sobrecarga de envio da mensagem pelo gerenciador para v_2, denotada por o_sHM, e da sobrecarga associada ao recebimento da mensagem por v_2, denotada por o_rAP.

$$o_r = (t4 - t3) - 1 \tag{5.3}$$

Desse modo, em razão da utilização dos gerenciadores como roteadores das mensagens da aplicação, as sobrecargas de envio o_s e recebimento o_r são compostas pelas sobrecargas geradas pelas tarefas da aplicação e pelo processo gerenciador HM. É importante destacar que na comunicação entre processos MPI, conforme descrito em [110], a sobrecarga varia em função do tamanho da mensagem e dos processadores que enviam e recebem os dados.

Se v_1 e v_2 são alocadas em máquinas ou *sites* distintos, o SM e o GM também participam da comunicação, respectivamente. Nestes casos, as sobrecargas geradas por esses processos devem ser contabilizadas nas sobrecargas o_s e o_r. Supondo que os que os gerenciadores SM e GM não serão alocados em recursos que executam tarefas da aplicação, as sobrecargas desses gerenciadores podem ser modeladas como latência ou atraso da

mensagem entre v_1 e v_2, como visto na Figura 5.3.

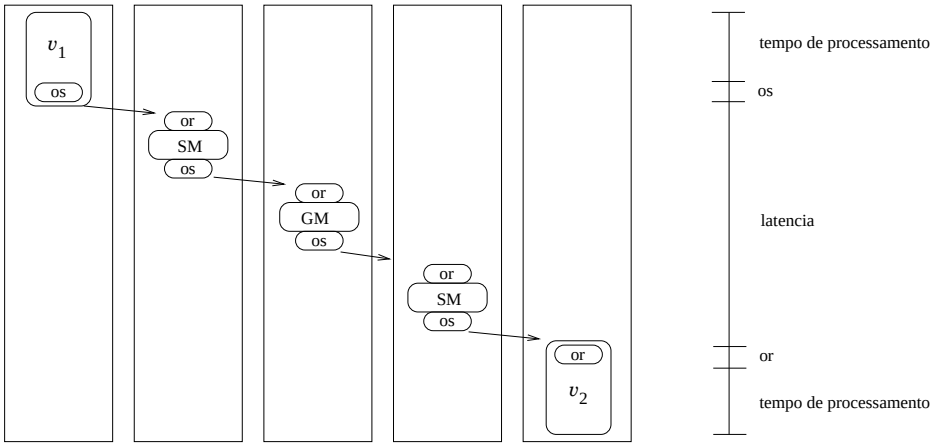

Figura 5.3: Comunicação entre dois processos MPI executando em *sites* distintos através do EasyGrid AMS.

Apesar desses gerenciadores também produzirem sobrecargas que incidem em processadores que não executam tarefas da aplicação, a entrega da mensagem é atrasada. Desta forma, do ponto de vista de um escalonador de tarefas este atraso é visto como latência de comunicação.

Tabela 5.1: Sobrecarga e latência entre dois processos que se comunicam através do EasyGrid

		Tempo em milesegundos		
#mensagem (MB)	tipo de custo	sn17	sn17 - sn18	sn17 - oiti
2	o_s	44	64	79
	latência	-	75	2390
	o_r	54	53	63
4	o_s	73	116	112
	latência	-	127	4910
	o_r	101	100	290
8	o_s	126	237	218
	latência	-	235	9830
	o_r	194	195	1054
16	o_s	237	360	426
	latência	-	477	20050
	o_r	386	385	8036
32	o_s	454	694	808
	latência	-	1047	43720
	o_r	767	809	26860

A Tabela 5.1 mostra os valores das sobrecargas de envio (o_s) e recebimento (o_r) para mensagens de tamanho 2, 4, 8, 16 e 32 *megabytes*, assim como a latência de comunicação entre pares de processadores para as máquinas especificadas. Nesse experimento, a comunicação é feita de três maneiras distintas: em um mesmo processador (sn17), entre dois processadores similares utilizando o mesmo *switch* (sn17 e sn18) e, por último, entre dois processadores com capacidade de processamento bem diferente e localizados em sites distintos (sn17 e oiti). Os resultados mostram que a sobrecarga para enviar mensagens entre

processos localizados em um mesmo processador é menor (em quase todos os cenários foi cerca de 50% mais rápido) do que para enviar mensagens entre processos no mesmo *site* e também em *sites* distintos. Por sua vez, para mensagens muito grande (maior ou igual a 16 Mb) a sobrecarga de envio entre sites distintos é maior do que entre processos localizados no mesmo site (≈20% mais rápido), pois é computado a sobrecarga do gerenciador HM enviar para o gerenciador SM, como no detalhe da Figura 5.4. Além disso, para os três casos a sobrecarga varia com o tamanho da mensagem. A latência também varia com o tamanho da mensagem e com o tipo de conexão entre as máquinas. Neste exemplo, a conexão entre sn17 e oiti é muito mais lenta do que a conexão entre sn17 e sn18 devido a diferença de velocidade entre canais que interligam essas máquinas (Figura 4.5). Além disso, as sobrecargas de envio e recebimento dependem da capacidade da máquina que está envolvida na comunicação, como observado em relação as sobrecargas de recebimento para sn18 (Pentium IV 2.6 GHz) e oiti (Pentium III 550 MHz) na Tabela 5.1.

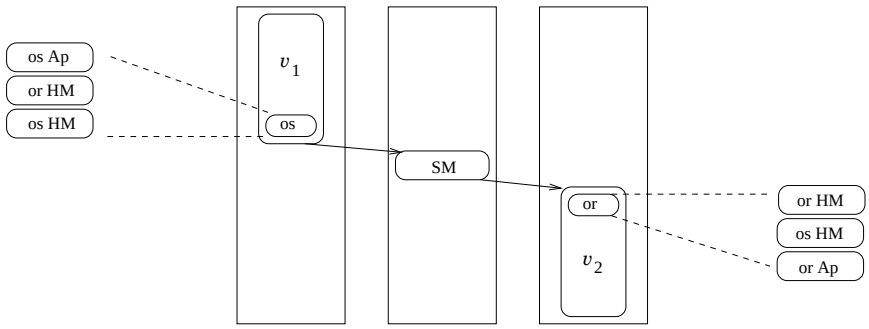

Figura 5.4: Sobrecarga de comunicação entre dois processos MPI executando em recursos distintos mas do mesmo *site*, através do EasyGrid AMS.

Como conclusão dos experimentos acima descritos, foi identificado que antes de enviar e receber qualquer mensagem existe uma sobrecarga de envio e recebimento, respectivamente, que incide sobre o processador e que varia em função do tamanho da mensagem. Porém, diferentemente dos modelos de comunicação apresentados na seção 2.4 que utilizam o modelo de execução tradicional 1PProc, foi identificado que a comunicação entre dois processos escalonados em um mesmo processador através do modelo 1PTask também produz sobrecarga que, de acordo com os resultados da Tabela 5.1, é um pouco menor do que a sobrecarga para enviar mensagens entre processos localizados em processadores distintos. Para a mensagem ser transmitida de um processador para outro, existe uma latência de comunicação que depende tanto da velocidade do canal quanto do tamanho da mensagem.

Considerando os aspectos discutidos, o modelo arquitetural para escalonamento de tarefas de uma aplicação paralela MPI em uma grade computacional proposto neste trabalho, é definido a seguir. A grade é composta por $P = \{p_0, p_1, p_2, ..., p_{m-1}\}$ processadores

heterogêneos, onde cada processador possui sua própria memória local, interconectados segundo alguma topologia. O modelo especifica o índice de retardo computacional $csi(p_j)$ para cada $p_j \in P$ que corresponde a capacidade de processamento de cada recurso p_j da grade. Para se transmitir uma mensagem de uma tarefa v_x para uma tarefa v_y, uma sobrecarga de envio o_s é imposta no processador p_i onde v_x estiver alocada, assim como, a sobrecarga o_r ocorre no processador p_j quando v_y receber a mensagem para qualquer p_i e $p_j \in P$, mesmo se $p_i = p_j$. Essas sobrecargas variam em função do tamanho da mensagem e da capacidade de processamento do processador. Além disso, para a mensagem ser transmitida de um processador p_i para um processador p_j existe uma latência de comunicação $l_{i,j}(\omega(v_x, v_y))$ que depende da velocidade do canal e varia de acordo com o tamanho da mensagem.

5.2 Avaliação e Análise de Aplicações Paralelas representadas através de GADs

Na seção 4.2, foi apresentada uma análise de como representar aplicações *bag-of-tasks* através de GADs. Para se representar uma aplicação paralela qualquer, além dos pesos das tarefas, é necessário também sua especificação através de um GAD. Se esse grafo já existe (por exemplo, se for uma aplicação bastante estudada como a multiplicação de matrizes) então, assim como nas aplicações *bag-of-tasks*, basta descobrir o peso das tarefas, além também dos custos das mensagens. Porém, para aplicações novas (pouco estudadas), um grafo tem que ser gerado.

A solução ideal é obter uma ferramenta que receba como entrada um programa MPI qualquer e produza como resultado um GAD que represente esse programa. Porém, devido a dificuldade desse problema, até o presente momento não se sabe da existência de tal ferramenta (como pode ser visto na seção 2.3). Uma abordagem mais simples, é a utilização de linguagens para descrição de GADs [66, 104, 145, 158]. Entretanto, essa descrição é uma tarefa complexa que tem que ser feita pelos usuários da grade e, normalmente, é específica a ferramenta utilizada.

Apesar de não ser um dos objetivos desse trabalho a construção ou proposta de uma ferramenta para geração de GAD a partir de um programa MPI, as características do modelo de execução proposto *um processo por tarefa* podem facilitar a definição de tal ferramenta. Como especificado no modelo alternativo 1PTask, as tarefas se caracterizam por receber dados, computar e enviar dados, o que simplifica a identificação das tarefas e dos arcos entre elas para se gerar o GAD.

Todas as aplicações apresentadas nesse trabalho já possuíam uma representação atra-

vés de um GAD (por exemplo, em [40] é mostrado como representar a eliminação gaussiana através de uma GAD) ou ele foi criado manualmente após uma análise da aplicação. A subseção a seguir apresenta um estudo de caso feito sobre a aplicação paralela N-corpos, onde um algoritmo paralelo alternativo é descrito, assim como um GAD que representa tal aplicação. Em especial, é apresentada uma análise do desempenho do algoritmo tradicional e da nova abordagem proposta para ambientes heterogêneos. Além disso, também são apresentadas duas estratégias para a execução eficiente da aplicação paralela N-corpos em ambientes dinâmicos, heterogêneos e compartilhados, uma para o modelo de execução tradicional 1PProc e outra para o modelo alternativo 1PTask [132].

A aplicação N-corpos *ring* foi escolhida por várias razões, além de sua importância para a comunidade astrofísica: em razão de sua estrutura de comunicação paralela, fortemente acoplada ser similar a estrutura de vários outros problemas de simulação; a sua comprovada eficiência em *cluster* de computadores; e devido a existência de outras tentativas não convicentes de "gridificar"esta aplicação.

5.2.1 Aplicação Paralela N-corpos

O problema clássico N-corpos (N-*body*) simula a evolução de um sistema composto de N corpos, onde existe a força gravitacional exercida em cada corpo devido a sua interação com todos os outros corpos do sistema. Esta subseção apresenta o estudo realizado sobre o desempenho da aplicação N-corpos em ambientes heterogêneos, mostrando as limitações da abordagem tradicional neste tipo de ambiente, assim como uma maneira alternativa de paralelização que possibilita que a aplicação se adapte as características do ambiente (por exemplo, heterogeneidade, compartilhamento e dinamismo).

Algoritmos numéricos para solucionar o problema astrofísico N-corpos podem ser divididos em duas categorias: abordagem direta ou um esquema aproximado. Nos códigos que utilizam a abordagem direta, introduzido por Aarseth [1], o conjunto completo de N^2 forças entre as partículas são calculadas para cada passo da evolução (*time step*). O problema dessa abordagem é o seu alto custo computacional devido a complexidade de $O(N^2)$. A outra abordagem para o problema de N-corpos troca a soma direta de forças exercidas por partículas muito distantes por um esquema aproximado (também chamado de hierárquico). Nesses algoritmos a complexidade do problema é reduzida, porém sua precisão é menor do que a abordagem direta. Dois exemplos bem conhecidos são os algoritmos de Barnes-Hut [10] e Greengard [78] que possuem complexidades de $O(N\log N)$ e $O(N)$, respectivamente.

Por computar as forças exercidas entre todas as partículas do sistema, a abordagem

direta é mais geral, sendo usada para simular tanto regiões de pouca densidade como também de alta densidade [82], *globular star clusters* [72] ou galatic nuclei [112]. Com relação ao número de partículas (N), a complexidade O(N^2) da abordagem direta limita sua escalabilidade. Por exemplo, uma série de simulações com um milhão de estrelas, que é um importante problema astrofísico, é ainda um desafio.

Para simular sistemas com um número grande de partículas (por exemplo, a formação e evolução de grandes galáxias em espiral como a via láctea [85]) uma abordagem distribuída tem que ser utilizada. Existem duas implementações paralelas básicas para calcular a força exercida entre as partículas: algoritmo de cópia e algoritmo *ring*. No algoritmo de cópia todas as partículas são copiadas para cada processador a cada passo da evolução, o que requer uma grande quantidade de memória para um número grande de partículas. O algoritmo *ring* ou também chamado de sistólico [50, 98, 108] iterativamente computa e transmite dados através de uma rede de processadores e, mais importante, somente $N/|P|$ partículas (onde $|P|$ representa a quantidade de processadores disponíveis) são utilizadas em cada processador. Entretanto, estes algoritmos foram projetados para executarem em ambientes dedicados e homogêneos, como *clusters* de computadores.

5.2.1.1 O Algoritmo *ring*

Supondo que existam N corpos ou partículas para se fazer a soma direta e que um conjunto $P = \{p_1, p_2, p_3, ..., p_m\}$ de m processadores se encontra disponível para processar a soma direta. Baseado no modelo MPI tradicional de execução que aloca um processo v_k por processador p_k durante toda execução da aplicação, o algoritmo de *ring* funciona da seguinte maneira, como descrito em [50, 108], assumindo a existência de $|P|$ processadores homogêneos:

Algoritmo 2 : MPI_ring(v_k)

1 Calcula a interação entre suas $N/|P|$ partículas;
2 Envia suas $N/|P|$ partículas para $v_{(k+1)mod|P|}$;
3 Para $i = 1, |P| - 1$
4 Recebe $N/|P|$ partículas enviadas por $v_{(|P|+k-1)mod|P|}$;
5 Calcula a força das partículas recebidas sobre suas próprias partículas;
6 Envia as partículas recebidas para $v_{(k+1)mod|P|}$;

O Algoritmo 2 calcula a força gravitacional exercida em cada partícula (onde cada partícula é composta da sua posição, velocidade e aceleração) para um *time step*. Tipicamente, simulações astrofísica em geral, necessitam a execução de uma série de *time steps*, o que seria basicamente a repetição do código acima. A Figura 5.5 apresenta uma visão

ilustrativa do algoritmo *ring* executando em $|P| = 4$ processadores. Cada processo calcula a força entre suas $N/|P|$ partículas em cada um dos $|P|$ estágios (iterações do laço). Repare que a eficiência do algoritmo *ring* deriva do fato que todos os processos tem a mesma quantidade de partículas e foi originalmente proposto para executar em ambientes homogêneos e dedicados como os supercomputadores e *cluster*. No modelo tradicional de execução MPI existe apenas um processo longo por processador durante toda execução da aplicação.

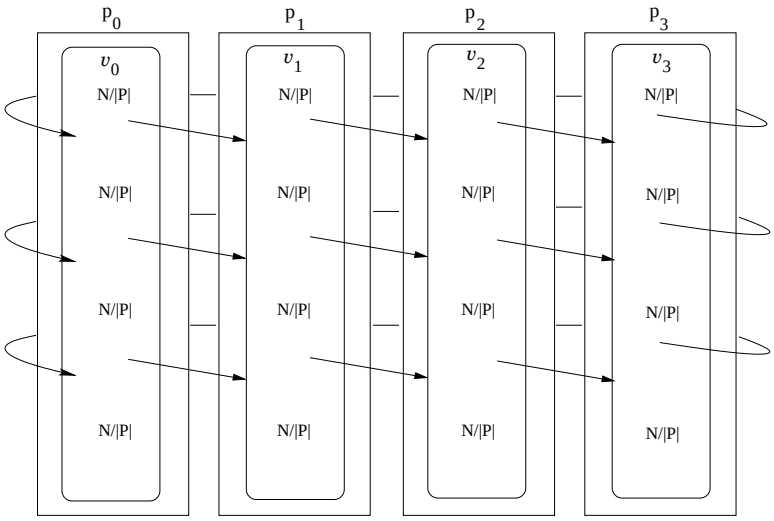

Figura 5.5: Os estágios do algoritmo *ring* em 4 processadores.

5.2.1.2 Ambientes Homogêneos com Comunicação Não Nula

Dada a comunicação e sincronização inerente ao algoritmo, mesmo em um sistema homogêneo pode não ser benéfico utilizar todos os recursos disponíveis. É possível derivar o número ideal de processos independentemente do número de processadores reais.

Seja W o número de processos, onde cada um necessita de W estágios para completar um *time step* (Linha 3 do Algoritmo 2). Cada processo recebe um total de N/W partículas e dessa maneira, a complexidade computacional de cada estágio é $O((N/W)^2)$, denotado por C_{comp}. A complexidade de comunicação para cada estágio, denotada por C_{comm}, é o custo de enviar os dados sobre as partículas para um processo vizinho. Como o algoritmo é composto de W estágios e $W-1$ fases de comunicação, o tempo aproximado de execução para o algoritmo *ring* para o problema de N-corpos executando W processos é $C_{comp} \times W + C_{comm} \times (W-1)$.

Como definido em [82, 108], o custo computacional C_{comp} representa o tempo C_f para executar a interação entre duas partículas multiplicado pela quantidade de partículas que cada processo calcula $(N/W)^2$. Similarmente, o custo de comunicação C_{comm} repre-

senta o tempo C_l para enviar uma partícula de tamanho δ para o processador vizinho multiplicado pelo número de partículas enviadas N/W somadas com a sobrecarga de comunicação C_o. Como resultado, Equação 5.4 apresenta o tempo de execução de um *time step* do MPI_ring() (onde pequenos fatores de correção da ordem de $1/W$ são ignorados):

$$ms_{hom}(W) = C_f \times \frac{N^2}{W} + C_l \times N + C_o \times W \tag{5.4}$$

Repare que esta equação assume o modelo de execução tradicional onde apenas um processo MPI é executado por processador (Modelo 1PProc). Um conhecimento comumente usado, especialmente no contexto de aplicações MPI, é que a execução concorrente de mais de um processo por processador em um ambiente homogêneo é menos eficiente do que apenas um processo por processador devido a comunicações interna a um processador desnecessárias e a troca de contexto [49, 61].

Para um número fixo de partículas N, dado que o custo computacional diminui monotonicamente enquanto o custo de comunicação aumenta monotonicamente, em função do valor de W (Equação 5.4), existe um valor específico para W, denotado por W_{opt}, onde o tempo de execução previsto ou *makespan* é mínimo. O valor W_{opt} é obtido através da derivada da Equação 5.4 e é apresentado na Equação 5.5.

$$W_{opt} = \sqrt{\frac{C_f}{C_o}} \times N \tag{5.5}$$

A partir deste valor W_{opt}, o número ótimo de processos em um ambiente com $|P|$ processadores disponíveis é obtido, onde uma das duas situações seguintes irá ocorrer: $W_{opt} > |P|$ ou $W_{opt} \leq |P|$.

Quando $W_{opt} \leq |P|$, somente W_{opt} processadores são necessários para executar os W_{opt} processos, pois a utilização de mais processos, além de ser um desperdício de recursos, levaria a um pior desempenho. Ao se utilizar o modelo 1PProc, se $W_{opt} > |P|$, então um total de $|P|$ processos devem ser criados e mapeados para os $|P|$ processadores homogêneos. Seja E_p um ambiente com $|P|$ recursos homogêneos.

Proposição 1. *Se $W_{opt} > |P|$, então o makespan do algoritmo ring com $W = |P|$ é ótimo para o ambiente E_p executando sob o modelo* 1PProc.

Demonstração. Seja W_1 e W_2 dois valores não ótimos de W tais que $W_1 < W_2 \leq |P| < W_{opt}$. Comparando os *makespans* $ms_{hom}(W_1)$ e $ms_{hom}(W_2)$ obtidos a partir da Equação 5.4, pode se mostrar que $ms_{hom}(W_1) > ms_{hom}(W_2)$ se $\frac{C_f.N^2}{C_o} > W_1 \times W_2$. Uma vez que $W_{opt}^2 = C_f.N^2/C_o$, esta última condição é verdadeira e a proposição é válida. □

5.2.1.3 Ambientes Heterogêneos

Em um ambiente heterogêneo dedicado, o desempenho da aplicação depende do poder computacional nativo disponível de cada processador p_k (assumindo que o tempo de computação é muito maior que o tempo de comunicação ou que a heterogeneidade dos canais de comunicação é pequena quando comparada com a dos processadores). Adotando a métrica índice de retardo computacional, $csi(p_k)$, é possível computar o tempo de execução para recursos heterogêneos. Como explicado na Seção 3.3.1, esta métrica é inversamente proporcional ao poder computacional de um processador p_k e dessa maneira o tempo de execução previsto ms_{het} do MPI_ring() sob o modelo 1PProc depende do $csi(p_s)$ do processador mais lento p_s, como visto na Equação 5.6.

$$ms_{het}(W) = csi(p_s) \times C_f \times \frac{N^2}{W} + C_l \times N + C_o \times W \tag{5.6}$$

Dado um conjunto de $|P|$ processadores heterogêneos, é possível determinar qual subconjunto de processadores maximiza o desempenho. Baseado na Equação 5.6, o algoritmo a seguir pode ser usado para encontrar tal subconjunto:

Algoritmo 3 : Processors_Choice ()

1 $bestCSI \leftarrow \min_{p_k \in P}\{csi(p_k)\}$;
2 $bestProcs \leftarrow \{p_i \in P / csi(p_i) = bestCSI\}$;
3 $W_{opt} \leftarrow$ derivada da Equação 5.6 em relação a $bestCSI$;
4 $nP \leftarrow W_{opt}$; $W \leftarrow nP$;
5 $bestMS \leftarrow bestCSI \times C_f \times \frac{N^2}{nP} + C_l \times N + C_o \times nP$;
6 se ($W_{opt} > |bestProcs|$) {
7 $p_r \leftarrow$ primeiro($bestProcs$);
8 $nP \leftarrow |bestProcs|$; $W \leftarrow nP$;
9 $ms \leftarrow csi(p_r) \times C_f \times \frac{N^2}{nP} + C_l \times N + C_o \times nP$;
10 $pList \leftarrow \langle p_k \in \{P - bestProcs\} / csi(p_k) \leq csi(p_{k+1}) \rangle$;
11 enquanto $((pList \neq \emptyset) \wedge (\sqrt{\frac{(csi(p_r) \times C_f)}{C_o}} > nP)$ {
12 $p_r \leftarrow$ primeiro($pList$);
13 $nP \leftarrow nP + |\{p_k \in plist / csi(p_k) = csi(p_r)\}|$;
14 $ms \leftarrow csi(p_r) \times C_f \times \frac{N^2}{nP} + C_l \times N + C_o \times nP$;
15 se ($bestMS > ms$) {
16 $bestMS \leftarrow ms; W \leftarrow nP$;
}
17 $plist \leftarrow plist - \{p_k \in plist / csi(p_k) = csi(p_r)\}$;
}
}

O Algoritmo 3 acha um subconjunto ótimo de recursos para execução do algoritmo *ring* em um ambiente heterogêneo dedicado, assumindo o modelo de execução 1PProc. O algoritmo inicialmente avalia se o subconjunto de recursos homogêneos mais rápidos (com $bestCSI$) é suficiente para alcançar o tempo de execução ótimo, de acordo com a Equação 5.6. Caso não seja, o algoritmo inclui o próximo subconjunto homogêneo mais rápido e calcula o novo *makespan* da seguinte forma. Assumindo que $bestProcs$ contém o conjunto de processadores mais rápidos, seja p_r um desses processadores (linha 7). Inicialmente nP contém a quantidade de processadores mais rápidos e de acordo com o valor de nP, o *makespan* correspondente é calculado (linha 9). Neste ponto, $pList$ é especificado como sendo uma lista de todos os processadores excluindo os mais rápidos ($bestProcs$) ordenado não decrescentemente por $csi()$ (linha 10). Considerando que $W_{opt} = \sqrt{\frac{(csi(p_r) \times C_f)}{C_o}}$ é o número ótimo de processadores com o mesmo $csi(p_r)$ corrente (linha 11), enquanto W_{opt} for maior que nP, o processador p_r com o próximo menor $csi()$ é identificado, o número de processadores com $csi()$ igual a $csi(p_r)$ é adicionado a nP e o novo makespan é calculado considerando todos os nP processadores (linhas 12 a 14). Assim, iterativamente, o algoritmo continua verificando se é possível encontrar *makespan* melhores, distribuindo a carga em um número maior de recursos heterogêneos. No pior caso, este processo irá se repetir até que todos os processadores disponíveis tenham sido avaliados.

Sejam E_1 e E_2 dois ambientes separados com número suficiente de recursos heterogêneos. Suponha que $csi_1 = \max_{\forall p_k \in E_1}\{csi(p_k)\}$ e que $csi_2 = \max_{\forall p_k \in E_2}\{csi(p_k)\}$, de maneira que $csi_1 < csi_2$. A seguinte proposição pode ser deduzida.

Proposição 2. *O* makespan *ótimo do algoritmo* ring *para o ambiente* E_1 *é menor do que o* makespan *ótimo para o ambiente* E_2.

Demonstração. O *makespan* ótimo é obtido com $W_{opt} = \sqrt{csi() \times C_f/C_o} \times N$ (a partir da derivação da Equação 5.6). Substituindo este valor na Equação 5.6, é possível mostrar que o *makespan* para E_1 é menor do que para E_2, pois $\sqrt{(csi_1)} < \sqrt{(csi_2)}$. □

5.2.1.4 Gridificando o Algoritmo *ring*

Para maximizar o desempenho do algoritmo *ring* para o problema de N-corpos ao executar não somente em ambientes heterogêneos, mas também dinâmicos e compartilhados como as grades computacionais, é proposta uma nova abordagem baseada no modelo de execução alternativo definido no Capítulo 3. Ao invés de se executar um processo por recurso, um conjunto de processos menores (chamados de tarefas) com relações de precedência entre si, é definido. A aplicação *ring* é modelada como um grafo acíclico direcionado (GAD) com largura e altura iguais a W, como pode ser visto na Figura 5.6. Apesar da

metodologia proposta representar uma significante mudança conceitual, é importante mencionar que as mudanças no código são pequenas e pontuais. Neste modelo, cada processo v_i executa o algoritmo a seguir:

Algoritmo 4 : AMS_ring*(v_i)*

1 Se ($i \geq W$)
2 Recebe N/W partículas enviadas por v_{i-W};
3 Se ($i \mod W = 0$) Recebe N/W partículas enviadas por v_{i-1};
4 Senão Recebe N/W partículas enviadas por v_{i-W-1};
5 Calcula a força das partículas recebidas sobre suas próprias partículas;
6 Se ($i < (W^2 - W)$)
7 Se ($i \mod W = W - 1$) Envia as N/W partículas para v_{i+1};
8 Senão Envia as N/W partículas para v_{i+W+1};
9 Envia suas N/W partículas para v_{i+W};

No Algoritmo 4, as primeiras W tarefas/processos, que não têm nenhuma dependência, somente computam a força exercida entre as suas partículas independentemente. Cada uma dessas tarefas termina após os dados serem enviados para seus dois processos sucessores. Todos os outros processos (com exceção dos W últimos) primeiramente necessitam receber dados de seus dois processos predecessores (linhas 2 a 4), para depois calcular a força entre suas respectivas partículas (linha 5) e então enviar dados para dois processos distintos (linhas 7 a 9). Baseado nessas características, o novo algoritmo N-corpos apresentado pode ser modelado através de um GAD que pode ser visto na Figura 5.6.

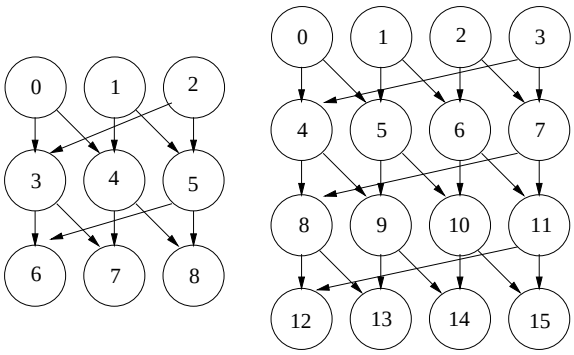

Figura 5.6: representação através de um GAD do novo algoritmo N-corpos *ring* com $W = 3$ and $W = 4$, respectivamente.

De acordo com [108], a complexidade por *time step* do algoritmo *ring* tradicional é $O(N^2/|P|)$ para computação e $O(N+|P|)$ para comunicação. Se $W = |P|$, como mostrado na Figura 5.7, o novo algoritmo tem a mesma complexidade computacional do algoritmo original, porém para a comunicação a complexidade é $O(2 * (N + |P|))$.

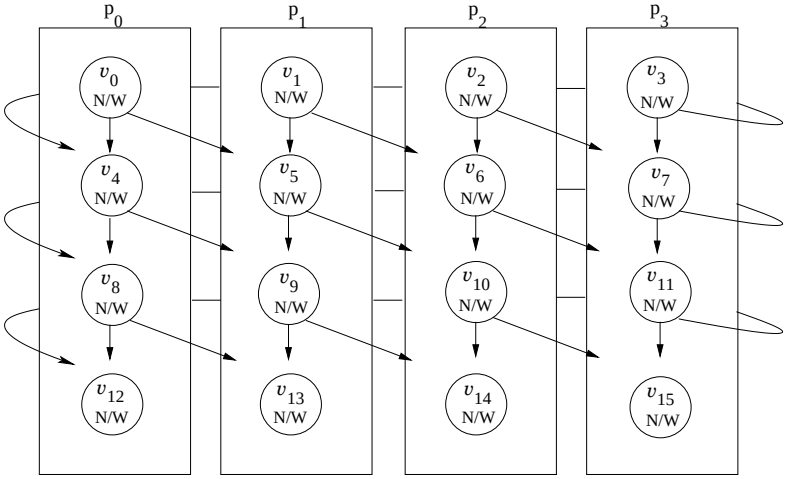

Figura 5.7: Os estágios do novo algoritmo *ring* com largura 4 em 4 processadores homogêneos.

Nesta abordagem para *gridificar* o algoritmo *ring*, o grau de paralelismo continua sendo a largura W. Porém, cada tarefa v_i agora somente realiza uma pequena quantidade de trabalho (quando comparado com a abordagem tradicional 1PProc), que corresponde ao cálculo da força entre N/W partículas, como especificado no Algoritmo 4. Como o algoritmo requer W iterações para processar todas as N partículas, o número total de tarefas por *time step* é W^2.

Um bom desempenho então, continua dependendo de um valor apropriado para W e um subconjunto dos processadores disponíveis nos quais as W^2 tarefas possam ser alocadas. Em ambientes dinâmicos, o reescalonamento de tarefas entre os processadores é fundamental. Um escalonador dinâmico pode tirar proveito de um maior número de tarefas menores para alocá-las de maneira apropriada entre os processadores de acordo com o poder computacional corrente disponível. A Figura 5.8 apresenta um exemplo de como um valor de W diferente do número de processadores disponíveis, pode beneficiar a execução da aplicação, em especial, em ambientes heterogêneos e dinâmicos.

Como pode ser visto na Figura 5.8 cada processo recebe a mesma quantidade de partículas N/W, porém como os processadores têm capacidade de processamento distintas (p_0 e p_1 são duas vezes mais rápidos que p_2 e p_3) foi utilizado uma largura $W = 6$. Com isso, os processadores mais rápidos recebem dois processos, enquanto que os processadores mais lentos apenas um, em cada estágio. Dessa maneira, processadores mais rápidos recebem mais trabalho, enquanto que processadores mais lentos recebem menos, balanceando melhor a carga computacional a ser executada, produzindo um melhor desempenho. Se fosse utilizada a largura $W = 4$, como mostra a Figura 5.7, todos os processadores executariam apenas um processo a cada estágio, com isso o desempenho da aplicação ficaria limitado ao desempenho da máquina mais lenta (neste caso p_2 e p_3).

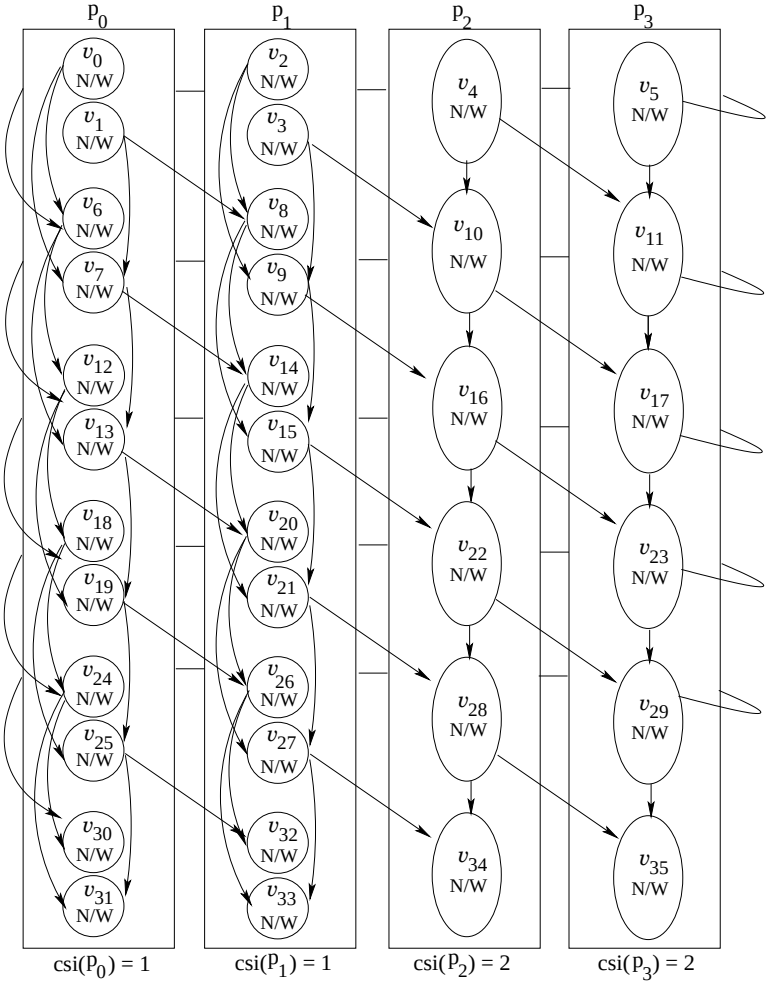

Figura 5.8: Os estágios do novo algoritmo *ring* com W=6 em 4 processadores heterogêneos.

Tipicamente, aplicações MPI criam todos os seus processos no início da execução da aplicação (modelo de execução 1PProc). Ao utilizar o modelo alternativo, o aumento no número total de processos da aplicação pode fazer com que configurações MPI padrão falhem devido a limitações na quantidade de processos que podem ser criados simultaneamente. Este problema pode ser superado se os processos forem criados dinamicamente, apenas no momento em que forem executar (por exemplo, ao receber todas as mensagens necessárias). Além disso, a criação dinâmica de processos evita as sobrecargas causadas pela troca de contexto e a migração de processos. Uma abordagem para se conseguir isto é através do uso de um sistema gerenciador de aplicação (AMS) para controlar a execução da aplicação, como descrito no início do capítulo anterior (Seção 4.1.1).

Com isso, o custo da criação dinâmica C_c de cada processo/tarefa deve ser considerado, juntamente com o custo computacional para executar a iteração entre duas partículas C_f multiplicado pela quantidade de partículas que cada processo/tarefa calcula $(N/W)^2$. Similarmente, o custo de comunicação é o tempo C_l para enviar uma partícula para o

processo vizinho multiplicado pelo número de partículas enviadas N/W mais duas vezes a sobrecarga de comunicação C_o (a sobrecarga é computado duas vezes pois são enviadas duas mensagens e seu valor não é sobreposto). Portanto, o tempo de execução previsto para um *time step* é calculado através da Equação 5.7 (onde pequenos fatores de correção da ordem de $1/W$ são ignorados).

$$ms_{AMSHom} = C_c W + C_f \frac{N^2}{W} + C_l N + 2C_o W \tag{5.7}$$

Enquanto que o tempo de execução para o algoritmo MPI_ring() executando através do modelo 1PProc depende da máquina mais lenta e do número total de processadores escolhidos, o tempo de execução do algoritmo AMS_ring() depende do valor de W e do escalonamento de seus W^2 processos/tarefas nos P processadores disponíveis no ambiente. A estratégia a seguir calcula um valor adequado para W e, baseado nesse valor, um bom escalonamento para suas tarefas:

Algoritmo 5 : Find_Best_W ()

1 $bestCSI \leftarrow \min_{p_k \in P}\{csi(p_k)\}$;
2 $bestProcs \leftarrow \{p_i \in P / csi(p_i) = bestCSI\}$;
3 $P_{ord} \leftarrow \langle p_1, \ldots, p_m / csi(p_i) \leq csi(p_{i+1})\rangle$;
4 $W_{opt} \leftarrow \sqrt{csi(bestCSI) \times C_f / C_o} \times N$
5 if ($W_{opt} \leq |bestProcs|$) {
6 $W \leftarrow W_{opt}$;
7 $bestMS \leftarrow C_c W + C_f \frac{N^2}{W} + C_l N + 2C_o W$;
8 }else {
9 $slowCSI \leftarrow MMC_{\forall p_k \in P}\{csi(p_k)\}$;
10 $W_{min} \leftarrow |bestProcs|$; $bestMS \leftarrow \infty$;
11 $W_{max} \leftarrow \sqrt{csi(slowCSI) \times C_f / C_o} \times N$
12 $i \leftarrow W_{min}$;
13 while ($i \leq W_{max}$){
14 $P_{ord}(i) \leftarrow$ próximos primeiros i processadores em P_{ord};
15 $ms \leftarrow$ TASK_SCHED ($DAG\text{-}N\text{body}(i), P_{ord}(i)$);
16 if ($ms < bestMS$) {
17 $bestMS \leftarrow ms$; $W \leftarrow i$; SAVE_SCHEDULE();
}
18 $i \leftarrow i + 1$;
}
}

O Algoritmo 5 inicialmente acha o valor do melhor $csi()$ ($bestCSI$) entre todos os recursos disponíveis (linha 1). Em seguida, armazena em uma lista todos os processadores

que possuírem o valor de $csi()$ igual a $bestCSI$ (linha 2). Logo após, ordena seus recursos de acordo com o $csi()$ (os recursos mais rápidos primeiro) e calcula o número ótimo de recursos W_{opt} para um ambiente homogêneo composto de máquinas com melhor $csi()$ (assim como no Algoritmo 3).

Se W_{opt} é menor ou igual que a quantidade das máquinas mais rápidas disponíveis ($|bestProcs|$) então, a partir da Proposição 2, não é necessário considerar mais processadores e o algoritmo termina com valor de W igual W_{opt} (ou seja, serão utilizados apenas W_{opt} recursos). Senão, o algoritmo avalia os *makespan* do algoritmo AMS_ring() com valores de W para ambientes construídos com o conjunto de $|bestProcs|$, que é incrementado a cada iteração com os próximos i recursos da lista de processadores ordenado pelo $csi()$, onde esses i processadores tem o mesmo $csi()$ (linha 14), até que o maior valor de W, W_{max} seja alcançado. O maior valor avaliado (W_{max}) é igual a W_{opt} para o menor conjunto composto dos processadores virtuais mais rápidos que puder utilizar todo o poder computacional dos recursos heterogêneos disponíveis (linhas 9 e 11).

Para cada valor entre W_{min} e W_{max}, uma heurística de escalonamento capaz de lidar com a heterogeneidade dos recursos é usada para alocar os i^2 processos nos i processadores mais rápidos disponíveis (linha 15). A política da heurística de escalonamento irá decidir se necessita utilizar ou não todos os processadores disponíveis até o momento.

Baseado nas proposições 1 e 2, a estratégia avalia os melhores valores de W, entretanto, o escalonamento de tarefas em um ambiente heterogêneo é um problema NP-completo [138], logo a solução ótima não é garantida devido a utilização de uma heurística.

Devido as características de um ambiente grade, uma largura W maior do que o número $|P|$ de recursos disponíveis pode levar a uma execução mais eficiente da aplicação pelas seguintes razões:

- uma grande quantidade de tarefas menores permite um escalonador distribuir as tarefas mais adequadamente de acordo com o poder computacional disponível em cada recurso;
- uma largura igual ao número de processadores disponíveis no início da aplicação pode não só limitar o grau de paralelismo, que poderia ser explorado quando novos recursos ficassem disponíveis, mas também reduzir o desempenho significantemente se o número de recursos for reduzido durante a execução;
- como os recursos de uma grade são compartilhados, uma grande quantidade de tarefas menores permite que um escalonador dinâmico reescalone tarefas que ainda não se encontram em execução de acordo com as variações no ambiente, sem a necessidade

de migração de processos que necessita de sincronização global e *checkpointing*;

- como os recursos de uma grade são propensos a falhas, uma grande quantidade de tarefas pequenas permite a utilização de mecanismos de recuperação através de *log* de mensagens que são mais eficientes do que esquemas que utilizam *checkpoint* [45].

A seção a seguir valida o modelo 1PTask para a execução de aplicações paralelas compostas de tarefas que se comunicam. Através de experimentos em ambientes reais e a estratégia apresentada nesta seção para a aplicação N-corpos, os resultados mostrarão a viabilidade de se executar aplicações fortemente acopladas em ambientes heterogêneos, dinâmicos e compartilhados como as grades computacionais.

5.3 Avaliação do Modelo 1PTask para Aplicações Fortemente Acopladas

Para validar o modelo de execução proposto para aplicações paralelas compostas de tarefas dependentes foi escolhido o problema clássico N-corpos (N-*body*) (Subseção 5.2.1). Com o objetivo de avaliar o desempenho das duas abordagens (modelo tradicional e alternativo) apresentadas para o problema de N-corpos, além da sobrecarga de se utilizar um sistema gerenciador de aplicações, os experimentos foram realizados em um ambiente de execução real semi-controlado composto de 24 processadores Pentium IV 2.6 GHz. Enquanto que as máquinas estavam dedicadas aos testes realizados, o mesmo não pode ser dito sobre a rede que interliga os computadores que é compartilhada com outros usuários. Esta subseção também apresenta uma comparação entre os dois algoritmos descritos na Subseção 5.2.1, implementados usando a linguagem C e a biblioteca padrão do LAM/MPI. O algoritmo chamado MPI ring [1, 82, 108], é um algoritmo tradicional para o problema de N-corpos utilizado pelos astrofísicos executando no modelo de execução padrão do MPI (1PProc). O novo algoritmo, executado através do *EasyGrid AMS* que automaticamente transforma a aplicação em uma versão autonômica e adota o modelo de execução alternativo proposto nessa tese (1PTask), é chamado de AMS ring. Como descrito na Seção 4.1.2, o *middleware EasyGrid AMS* adota uma estratégia híbrida de escalonamento. Em todos os experimentos que serão mostrados nesta subseção, a heurística HEFT [147] foi empregada como escalonador estático, que é um algoritmo bastante conhecido na literatura para ambientes heterogêneos. Os principais objetivos dos experimentos que serão apresentados são:

- investigar a sobrecarga da estratégia proposta em um ambiente homogêneo;

- comparação do desempenho dos algoritmos MPI ring e AMS ring em diversos cenários heterogêneos;
- avaliação das estratégias propostas para escolha do valor de W e dos recursos necessários para uma execução eficiente;
- avaliação do comportamento da estratégia proposta em um ambiente compartilhado.

É importante ressaltar que a comparação entre o algoritmo MPI ring tradicional e o algoritmo proposto AMS ring é uma comparação justa, pois o algoritmo MPI ring utiliza a melhor configuração entre os recursos disponíveis de acordo com a análise apresentada na Subseção 5.2.1, o que qualifica os resultados que serão apresentados. Em todos os experimentos, os resultados foram gerados a partir da média aritmética do tempo de, no mínimo, três execuções reais. Além disso, é importante destacar que os tempos obtidos nas três execuções foram sempre muito próximos uns dos outros.

5.3.1 Ambiente Homogêneo

O objetivo deste primeiro experimento é medir a sobrecarga associada ao gerenciamento da execução de processos através do *EasyGrid AMS* que adota o modelo MPI alternativo de execução quando comparado com a abordagem tradicional normalmente utilizada. O desempenho dos dois algoritmos, MPI ring e AMS ring, no mesmo *cluster* homogêneo dedicado, composto de 18 processadores Pentium IV 2.6 GHz, é comparado. Desse modo para não prejudicar o algoritmo MPI ring, o valor $W = 18$ determinado pelo algoritmo Processors_Choice() (Algoritmo 3, descrito na Subseção 5.2.1), foi utilizado para maximizar seu desempenho nesse ambiente.

Com o objetivo de realizar a soma direta para problemas com grande quantidade de corpos (de centenas de milhares a milhões), os resultados apresentados nas colunas onde $W = 18$ da Tabela 5.2 (coluna 2 para MPI ring e coluna 4 para AMS ring) mostram que a estratégia AMS ring é extremamente promissora. Nesta avaliação de ambientes homogêneos, para os dois menores valores de N, os tempos de execução do algoritmo AMS ring foram ligeiramente maiores (6,6% e 1,0%) que os tempos do MPI ring, como era esperado, devido a sobrecarga do *EasyGrid AMS* para gerenciar a criação e execução dos processos, assim como a comunicação extra que surgem em função do modelo alternativo de execução adotado [119, 133]. À medida que o valor de N aumenta, esta sobrecarga diminui até o ponto que AMS ring alcança melhores resultados do que MPI ring. Embora os recursos sejam homogêneos e dedicados ao experimento, existem pequenas variações no tempo de execução de cada estágio devido aos processos estarem executando em processadores distintos. Adicione a isto o fato que implementações MPI tradicionais necessitam

Tabela 5.2: Tempo de execução (em segundos) dos algoritmos MPI ring e AMS ring, à medida que o número de partículas aumenta, em um *cluster* heterogêneo.

	MPI ring (W=)		AMS ring (W=)				
N	18	24	18	24	36	42	48
50.000	**18,3**	26,3	19,5	21,8	21,6	19,6	25,2
100.000	73,3	103,6	74,0	78,0	68,0	**64,2**	68,7
150.000	165,4	237,5	164,9	165,6	143,0	**141,6**	145,5
200.000	293,4	431,1	291,1	290,8	250,4	**248,9**	251,7
250.000	457,8	654,5	456,6	450,8	387,9	**385,8**	389,5

de uma forma de sincronização entre cada fase de comunicação (chamada de estágio neste trabalho) para evitar *deadlocks*. O gerenciamento das comunicações através do *EasyGrid AMS* automaticamente evita os *deadlocks* ao custo de latências mais longas. Entretanto, por evitar o custo da sincronização entre cada estágio, além de utilizar *over-provisioning* e técnicas para esconder a latência através de um escalonamento efetivo, essa sobrecarga pode ser diluída, especialmente para granularidades e uma quantidade de tarefas maiores.

Ao adotar o modelo de execução alternativo, o *middleware EasyGrid AMS* simplifica a implementação das funcionalidades de uma aplicação autonômica ao custo de gerenciar $O(W^2)$ processos e duas vezes mais comunicações do que o MPI ring. Essa troca de modelo é especialmente vantajosa para ambientes heterogêneos, compartilhados e dinâmicos (por exemplo, as grades computacionais) como será mostrado nos experimentos a seguir.

5.3.2 Ambiente Heterogêneo

Para destacar o desempenho dos algoritmos ao lidarem com recursos heterogêneos, este experimento utiliza recursos dedicados que oferecem poder computacional distintos para a aplicação. Desse modo, 18 máquinas fornecem 100% do seu poder computacional enquanto que 6 máquinas fornecem apenas 50%. Sete configurações distintas da aplicação N-corpos foram avaliadas considerando os dois modelos:

(1) MPI ring com $W = 18$, executando nos 18 recursos mais rápidos;

(2) MPI ring com $W = 24$, executando em todos os 24 recursos;

(3) AMS ring com $W = 18$, onde o novo algoritmo *ring*, gerenciado pelo *EasyGrid AMS*, executa a aplicação representada por um GAD com largura de $W = 18$ tarefas;

(4) AMS ring com $W = 24$;

(5) AMS ring com $W = 36$;

(6) AMS ring com $W = 42$;

(7) AMS ring com $W = 48$.

Os resultados das duas primeiras colunas da Tabela 5.2 mostram que o algoritmo MPI ring é incapaz de aproveitar os 16,7% a mais de poder computacional (os 6 recursos mais lentos) uma vez que os tempos de execução para $W = 24$ foram todos piores do que para $W = 18$ (que executa apenas nos 18 processadores homogêneos). Este comportamento destaca o fato que pode ser melhor executar essa classe de aplicação em um subconjunto de processadores do que em todos os processadores disponíveis, confirmando a análise apresentada na Subseção 5.2.1. Dependendo do número de recursos lentos disponíveis e seu poder computacional, pode ser melhor executar a aplicação apenas nos recursos mais rápidos, como será analisado com mais detalhes mais adiante.

Como mencionado anteriormente, para um número pequeno de partículas o tempo computacional de cada processo pode não ser grande o suficiente para fornecer uma melhora significativa devido a sobrecarga de executar um programa MPI controlado por um sistema gerenciador de aplicações (*AMS*). Entretanto, a aplicação AMS ring com um valor apropriado de W para $N > 50.000$ obteve melhoras significativas (variando de 12,4% a 15,7%) em comparação com os tempos de execução produzidos pelo MPI ring em um ambiente homogêneo. Neste experimento, o melhor valor de W, W_{opt}, encontrado pelo algoritmo Find_Best_W () foi $W_{opt} = 42$. Repare que, para um dado N, os *makespan* não necessariamente diminuem monotonicamente com o aumento do valor de W (para $W < W_{opt}$) em um ambiente heterogêneo. Os experimentos a seguir investigam a habilidade dos algoritmos propostos, Processors_Choice () e Find_Best_W () (Subseção 5.2.1), acharem um valor de W apropriado para cada algoritmo *ring*.

5.3.3 Escolhendo o valor de W e os Recursos Apropriados

No experimento anterior, quando o algoritmo *MPI ring* foi executado em todos os recursos disponíveis houve perda de desempenho, como pode ser visto na Tabela 5.2, pois alguns dos recursos tinham menos poder computacional. Este experimento investiga a habilidade dos respectivos algoritmos Processors_Choice() e Find_Best_W () (Algoritmos 3 e 5), dado um conjunto de processadores heterogêneos, identificar o melhor valor de W a ser utilizado. Achar W_{opt} em cada caso depende de uma estimativa precisa do *makespan* para cada implementação.

Para estimar o *makespan* utilizando a Equação 5.6, é necessário informar os valores de C_f, C_o e C_l para a máquina mais lenta que está sendo utilizada. Este processo pode ser muito trabalhoso, ou até impraticável, devido a variedade de recursos que podem estar disponível em uma grade. Além disso, na prática, estes valores não são constantes com

relação a N e são afetados pela granularidade do processo. Para tratar esses problemas, foi adotado uma forma mais abstrata onde C_{comp} e C_{comm} são calculados em função de N e W tanto para *MPI ring* quanto para *AMS ring*, baseados no tempo de execução de um único *time step* para valores pequenos de N em um ambiente controlado (neste caso processadores Pentium IV 2.6 GHz interconectados por *switches Gigabit* e *fast Ethernet*). A partir dos valores do tempo de execução e de comunicação obtidos neste ambiente controlado, foram geradas duas equações para representar C_{comp} e C_{comm} para cada uma das implementações, como mostram as equações 5.8, 5.9, 5.10 e 5.11. O fato que o tempo de execução para cada implementação é diferente é refletido nos seus respectivos valores médios para C_f, C_o e C_l. Para generalizar esses valores para recursos heterogêneos é utilizado o índice de retardo computacional $csi()$.

$$C_{comp}^{MPI} = 34.192 - 0.023438 \times \frac{N}{W} + 0.00013315 \times (\frac{N}{W})^2 \tag{5.8}$$

$$C_{comm}^{MPI} = 0.35 + 8.5 \times 10^{-5} \times \frac{N}{W}\delta - 6.6 \times 10^{-13} \times (\frac{N}{W}\delta)^2 \tag{5.9}$$

$$C_{comp}^{AMS} = 44.094 - 0.016733 \times \frac{N}{W} + 0.00012812 \times (\frac{N}{W})^2 \tag{5.10}$$

$$C_{comm}^{AMS} = 11.059 + 0.011876 \times \frac{N}{W}\delta - 7.18 \times 10^{-7} \times (\frac{N}{W}\delta)^2 \tag{5.11}$$

Os processadores utilizados para executar a aplicação estavam dedicados ao experimento, de modo que 6 processadores forneceram 100% do seu poder computacional, 12 processadores forneceram 50% e 6 recursos apenas 33%. Três cenários distintos foram avaliados pelo algoritmo Processors_Choice(), tendo como entrada:

(1) apenas os 6 recursos mais rápidos;

(2) os 6 recursos mais rápidos juntamente com os 12 recursos com 50% do poder computacional;

(3) todos os 24 recursos.

A Tabela 5.3 apresenta o *makespan* estimado e o tempo de execução real para o algoritmo MPI ring com $N = 200.000$ partículas. Nos cenários (1) e (2), Processors_Choice() decidiu utilizar todos os recursos disponíveis ($W = 6$ e $W = 18$, respectivamente). Para o cenário (3) o algoritmo decidiu não utilizar todos os recursos disponíveis, mas somente os

Tabela 5.3: Estimativa e tempo de execução real (em segundos) para MPI ring em um ambiente heterogêneo dedicado.

Cenários	W	tempo previsto	tempo MPI ring	erro(%)
(1)	6	883,4	887,1	0,4
(2) and (3)	18	**583,9**	**587,6**	0,6
all resources	24	654,4	655,8	0,2

Tabela 5.4: Estimativa e tempo de execução (em segundos) real para AMS ring em um ambiente heterogêneo dedicado.

Cenários	W	tempo previsto	tempo AMS ring	erro(%)
(1)	6	884,0	889,5	0,6
(2)	24	431,7	434,9	0,7
(3)	50	**380,1**	**381,7**	0,4

18 mais rápidos. De acordo com o resultado apresentado na terceira linha da Tabela 5.3, Processors_Choice() fez a melhor escolha, uma vez que o tempo de execução teria sido pior se todos os 24 recursos tivessem sido utilizados. Devido as pequenas variações entre o *makespan* previsto e o tempo de execução real, como mostra a coluna erro da Tabela 5.3, é possível concluir que a modelagem é bastante precisa. Entretanto, a principal informação fornecida pelo algoritmo Processors_Choice() para uma execução otimizada da aplicação MPI ring é o subconjunto de processadores que deve ser utilizado.

A aplicação AMS ring também foi executada considerando esses mesmos cenários, com o objetivo de avaliar seu desempenho de acordo com a variação de W. O Algoritmo 5 (Find_Best_W ()) foi utilizado para encontrar o melhor valor de W e definir o escalonamento estático para as W^2 tarefas no conjunto P de processadores disponíveis.

Os tempos de execução do AMS ring para os cenários (2) e (3), apresentados na Tabela 5.4, foram bem melhores do que os do MPI ring (Tabela 5.3). No cenário (3), o tempo de execução da aplicação MPI ring foi 53,9% pior do que o AMS ring. No cenário (1), apesar de ser um ambiente homogêneo, a aplicação AMS ring foi apenas ligeiramente pior do que MPI ring devido as sobrecargas do sistema gerenciador (AMS). Essas sobrecargas também aumentam ligeiramente o erro entre o tempo previsto e a execução real.

A Figura 5.9 apresenta o desempenho da aplicação AMS ring com $N = 200.000$ partículas utilizando o cenário (3) e duas estimativas do tempo de execução, para valores crescentes de W. Repare que o rótulo **makespan** se refere ao *makespan* previsto pelo algoritmo HEFT baseado no modelo latência de comunicação [125], que considera que não existe custo para processos que se comunicam em um mesmo processador.

Por outro lado, o rótulo makespan usando sobrecarga se refere ao resultado utilizando

uma versão modificada do HEFT que adota o modelo de comunicação proposto neste trabalho de tese (Seção 3.3.2) que considera um custo computacional ou sobrecarga para enviar e receber mensagens, mesmo que os processos ou tarefas estejam localizados no mesmo processador. Além de ser um modelo de comunicação mais preciso, a razão para sua utilização é que a comunicação tem um impacto maior no desempenho da aplicação AMS ring (principalmente quando $W > |P|$) do que na implementação tradicional MPI ring. Como explicado na Seção 3.3.2, toda comunicação tem que passar por pelo menos um processo gerenciador do *EasyGrid AMS* antes de ser entregue ao processo destino, o que aumenta a sobrecarga de comunicação dos processos da aplicação AMS ring. Essas sobrecargas são modeladas como custos de processamento ao invés de latência de comunicação.

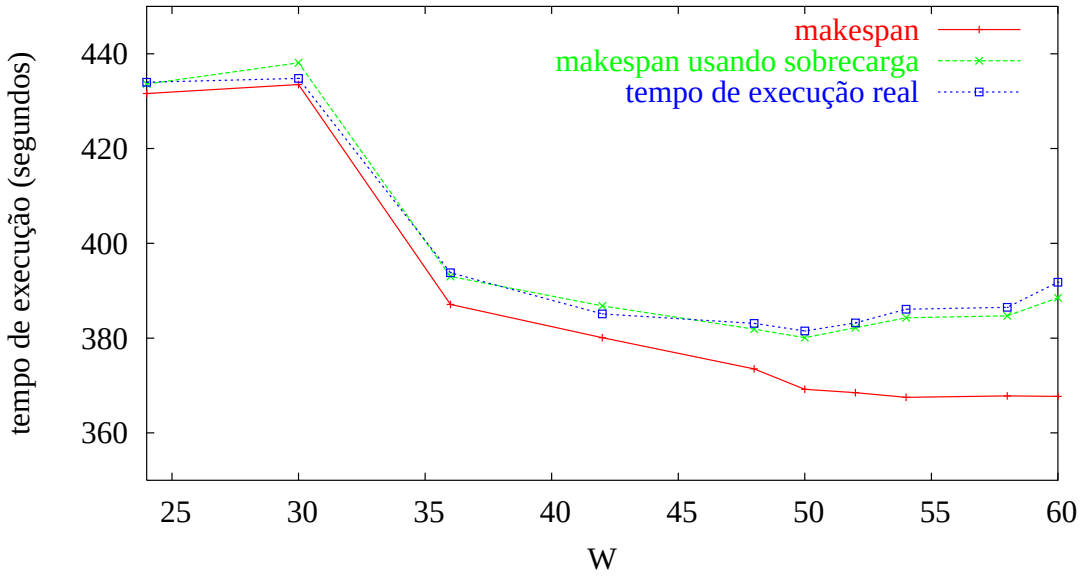

Figura 5.9: Makespan estimados e tempo de execução real para o AMS ring para diferentes valores de W.

Desse modo, Find_Best_W () foi executado utilizando dois modelos de comunicação: (a) o algoritmo HEFT original;(b) o algoritmo HEFT modificado que considera sobrecarga e comunicação interna a um processador. O valor retornado por Find_Best_W () para o modelo (b) foi $W = 50$ enquanto para o modelo (a) foi $W = 54$. Observando a Figura 5.9, o tempo de execução mínimo foi realmente obtido com $W = 50$. A modelagem da sobrecarga de comunicação, assim como a comunicação interna a um processador permitiu estimativas muito mais precisas e com comportamento consistente ao da execução real do que o modelo de comunicação padrão, que considera apenas latência. Por outro lado, a diferença entre os tempos de execução para $W = 50$ e $W = 54$ foi de apenas 1%. Enquanto ambos os modelos foram razoavelmente precisos para valores pequenos de W, à medida que W aumenta o modelo (a) se torna menos preciso em função da diminuição da granularidade do processo e, consequentemente, o aumento no custo de comunicação.

5.3.4 Distribuição Heterogênea

O experimento a seguir investiga o comportamento das aplicações *ring* ao se variar a distribuição de recursos rápidos e lentos. Um total de 24 recursos estavam disponíveis através de três cenários distintos e seis variações do algoritmo N-corpos *ring* foram executadas. Os cenários foram os seguintes:

(1) 18 máquinas com 100% de seu poder computacional enquanto 6 máquinas com apenas 50%;

(2) 12 máquinas com 100% de seu poder computacional enquanto 12 máquinas com apenas 50%;

(3) 6 máquinas com 100% de seu poder computacional enquanto 18 máquinas com apenas 50%;

Tabela 5.5: Tempo de execução (em segundos) do MPI ring e AMS ring com 200.000 partículas em três cenários.

Cenários	MPI ring	AMS ring com $W =$				
		24	30	36	42	48
(1)	291,7 (18)	291,2	260,5	251,1	**250,0**	251,8
(2)	433,2 (12)	360,0	300,6	**291,1**	292,0	293,5
(3)	440,3 (24)	432,6	**347,2**	349,3	348,5	353,7

Os resultados apresentados na Tabela 5.5 e Figura 5.10 são baseados na aplicação *ring* executando com seus melhores valores de W, obtidos a partir dos algoritmos Processors_Choice() e Find_Best_W (), respectivamente. No caso do AMS ring, além de executar a aplicação com os melhores valores de W, foram utilizados valores de W próximos do melhor valor encontrado em cada cenário para observar o desempenho da aplicação.

Os resultados mostram que o MPI ring produz resultados ruins quando comparados com a melhor execução do AMS ring, apesar de usar a melhor configuração de recursos fornecida pelo Processors_Choice(): somente os 18 recursos mais rápidos no cenário (1); nos 12 recursos mais rápidos no cenário (2); em todos os recursos no cenário (3). Comparando o desempenho da aplicação MPI ring nos cenários (1) e (2), a redução de 14,2% do poder computacional disponível fez com que o tempo de execução aumentasse em 48,6%, enquanto que entre os cenários (2) e (3), uma redução adicional no poder computacional disponível de 16,7% causou um aumento de apenas 1,6% no tempo de execução, pois, para o cenário (3), o algoritmo utilizou os recursos mais lentos.

Diferentemente, os benefícios do escalonamento estático fornecidos para a aplicação AMS ring produziram um desempenho muito bom para todos os cenários onde $W > |P|$.

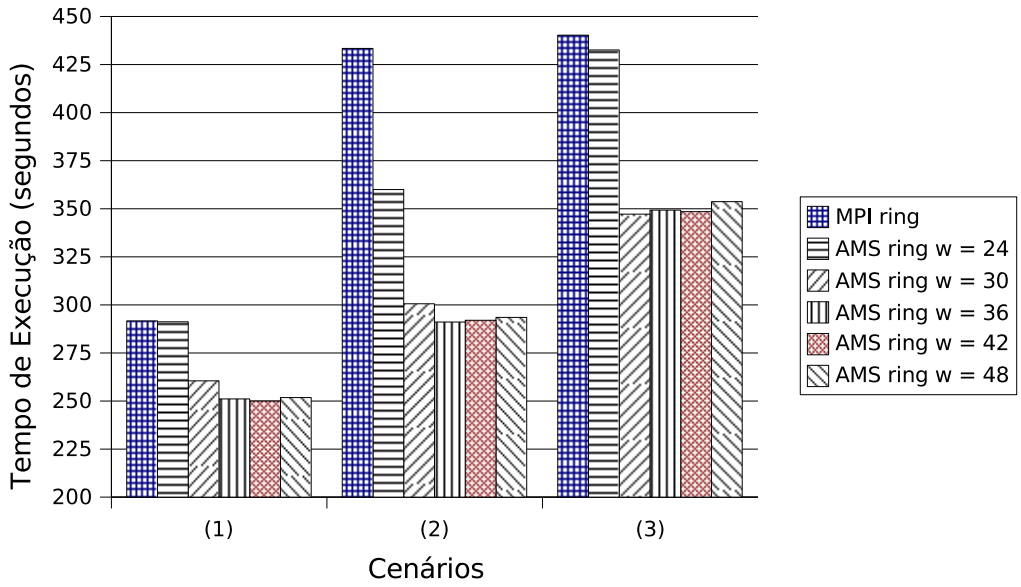

Figura 5.10: Desempenho dos algoritmos MPI ring e AMS ring com 200.000 partículas em três cenários distintos.

De acordo com o algoritmo Find_Best_W(), os melhores *makespan* são alcançados com $W = 42$ para o cenário (1), $W = 36$ para o cenário (2) e $W = 30$ para o cenário (3), resultado este validado através dos tempos de execuções reais, como pode ser visto na Tabela 5.5 e na Figura 5.10. Para cada cenário, os tempos de execução obtidos pelo AMS ring foram 14,3%, 32,8% e 21,1% melhores do que o MPI ring, respectivamente. AMS ring faz um melhor uso dos recursos disponíveis, comparando o desempenho da aplicação nos cenários (1) e (2), o tempo de execução aumentou apenas 16,8%, e nos cenários (2) e (3), 19,3%, valores esses bem próximos da redução do poder computacional entre cada cenário.

Uma outra característica importante que pode ser destacada nesse experimento é que execuções de instâncias do AMS ring com valores de W próximos ao valor que produziu o melhor *makespan* também obtiveram bons resultados. Como pode ser visto na Tabela 5.5, a diferença entre os tempos de execução com os melhores valores de W para cada cenário e os valores próximos foi menor do que 1%. A razão para este comportamento é que o número total de processos para valores de W próximo ao valor com melhor *makespan* foi suficiente para se conseguir um bom escalonamento. Isso é possível, pois a comunicação do AMS ring é assíncrona e com isso o escalonador consegue dividir as tarefas entre os processadores existentes sem grandes perdas de desempenho, mesmo quando a quantidade de tarefas não é totalmente proporcional ao poder computacional disponível. Este experimento também mostra que, para valores superiores ao valor ideal de W, fornecido pelo algoritmo Find_Best_W(), o tempo de execução piora apesar de uma maior quantidade de processos para distribuir entre os recursos. A explicação para tal comportamento é que,

Tabela 5.6: Tempo de execução (em segundos) do MPI ring e AMS ring com 200.000 partículas e grau de heterogeneidade distintos.

Cenários	MPI ring	AMS ring com $W =$							
		24	30	36	42	48	54	56	60
(1)	433,2	360,0	300,6	**291,1**	292,0	293,5	296,8	298,3	301,0
(2)	433,2	434,1	427,9	379,2	339,0	**333,3**	336,7	338,5	342,1
(3)	433,2	434,4	427,1	427,6	438,1	396,8	366,1	**365,3**	368,7

como pode ser visto na Equação 5.7, com o aumento do número de processos aumenta-se também o custo de comunicação e com isso a sobrecarga de comunicação para mensagens entre processos localizados no mesmo processador.

Quando $W = |P|$, AMS ring ainda conseguiu tempos razoáveis, porém o número pequeno de processos paralelos limita uma boa distribuição das tarefas. Um exemplo deste comportamento pode ser visto no cenário (1) onde, apesar dos 24 recursos disponíveis, somente os 18 mais rápidos foram utilizados, uma vez que o número de tarefas prontas e o poder computacional dos recursos não eram suficientes para que os processadores mais lentos fossem utilizados.

5.3.5 Grau de Heterogeneidade

Com o objetivo de estudar o comportamento da nova abordagem e da abordagem tradicional do algoritmo *ring* para ambientes compostos de recursos com diferente níveis de heterogeneidade entre seus recursos, o experimento a seguir, com $N = 200.000$ partículas, foi realizado. Três cenários distintos foram definidos:

(1) 12 recursos com 100% do seu poder computacional e 12 recursos com apenas 50%;

(2) 12 recursos com 100% do seu poder computacional e 12 recursos com apenas 33%;

(3) 12 recursos com 100% do seu poder computacional e 12 recursos com apenas 25%.

Assim como nos experimentos anteriores, o programa MPI ring foi executado nos recursos especificados pelo algoritmo Processors_Choice(). Por sua vez, a aplicação AMS ring foi executada nos processadores designados pelo algoritmo Find_Best_W(), seguindo o escalonamento de tarefas fornecido pelo HEFT considerando a sobrecarga, inclusive, entre processos localizados em um mesmo processador.

A Figura 5.11 e a Tabela 5.6 mostram que o desempenho da aplicação MPI ring para os três cenários foi o mesmo, pois Processors_Choice() escolheu apenas os 12 recursos mais rápidos em cada um dos cenários. Por sua vez, com o Find_Best_W() escolhendo

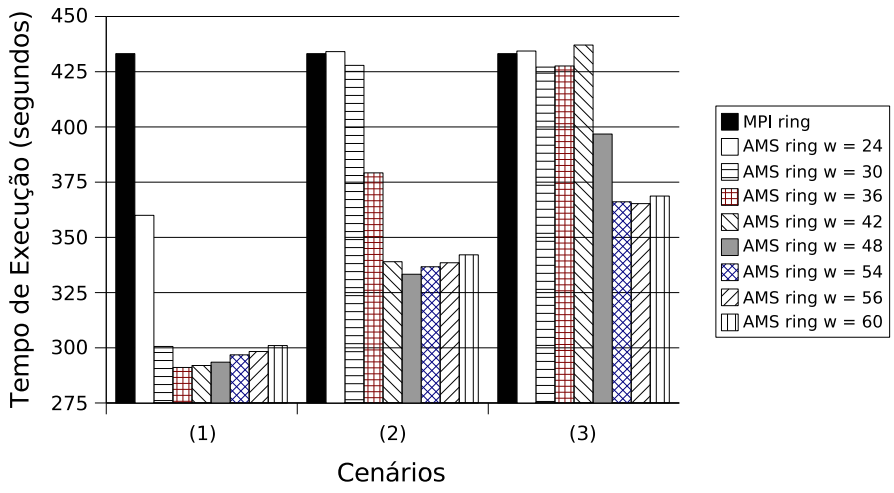

Figura 5.11: Desempenho do MPI ring e AMS ring com 200.000 partículas e grau de heterogeneidade distintos.

$W = 36$, $W = 48$ e $W = 56$ para os cenários (1), (2) e (3), respectivamente, a aplicação AMS ring obteve resultados significantemente melhores do que MPI ring. Para cada um dos três cenários, AMS ring utilizou os 24 recursos disponíveis com os respectivos valores apropriados de W para cada cenário, calculado através do algoritmo Find_Best_W().

Dado um número fixo de recursos, à medida que o grau de heterogeneidade aumenta, o poder computacional disponível diminui e, como esperado, o desempenho do AMS ring diminui proporcionalmente. Valores pequenos de W limitam o grau de paralelismo, de modo que o escalonador estático é forçado a limitar a escolha de recurso apenas aos mais rápidos. Deste modo, para maiores níveis de heterogeneidade, um valor maior de W é necessário para atingir melhor desempenho uma vez que um número maior de tarefas paralelas permite ao escalonador distribuir as tarefas de cada estágio proporcionalmente ao poder computacional de cada recurso. A Tabela 5.6 também mostra que o tempo de execução para valores de W próximo ao valor ideal produzem resultados apenas ligeiramente piores. Por exemplo, o tempo de execução com $W = 56$ para o cenário (3) foi apenas 0,3% melhor do que com $W = 54$ e 0,8% melhor do que com $W = 60$.

5.3.6 Ambiente Compartilhado

Os experimentos anteriores mostraram que a nova abordagem para o algoritmo *ring* pode explorar ambientes estáticos heterogêneos eficientemente. Por outro lado, a abordagem tradicional, eficiente em ambientes homogêneos, sofre uma degradação significante no seu desempenho à medida que a heterogeneidade aumenta.

Tabela 5.7: Tempo de execução (em segundos) dos algoritmos MPI ring e AMS ring $W = 48$ com 200.000 partículas em ambientes dinâmicos compartilhados.

TS	MPI ring	AMS ring	melhora(%)	limite inferior
1	314,1	224,1	28,6	217,8
2	712,6	440,6	38,2	435,6
3	1136,3	659,1	41,9	653,4
4	1555,3	890,5	42,8	871,2
5	1935,6	1138,1	41,2	1089,0

O experimento a seguir avalia o desempenho dos dois algoritmos ao serem executados em ambientes dinâmicos compartilhados. Diferentemente dos experimentos anteriores onde o algoritmo N-corpos *ring* foi executado por apenas um *time step*, uma série de *time steps* são executados com o objetivo de avaliar o desempenho dos algoritmos. A solução de problemas astrofísicos reais pode requerer milhares de *time steps*.

Os 24 recursos foram divididos em três *sites* compostos de 8 recursos cada, com uma carga extra (um programa *cpu bound*) executando em um recurso, escolhido aleatoriamente, do primeiro *site*, de maneira que apenas 50% do poder computacional fica disponível para a aplicação. Após, aproximadamente, 1/3 do tempo de execução total da aplicação, a carga extra migra para um recurso do segundo *site*, também escolhido aleatoriamente. O mesmo comportamento é repetido após 2/3 do tempo de execução total quando a carga extra migra para um recurso do terceiro *site* aleatoriamente.

Uma vez que grades computacionais podem ser compostas de recursos que podem estar compartilhados com outras aplicações, este experimento investiga o comportamento dos dois algoritmos N-corpos *ring* em ambientes compartilhados. Os algoritmos não tem conhecimento sobre a carga extra antes de iniciar a execução. Porém, mesmo que soubessem, determinar o W_{opt} seria extremamente difícil. Os modelos matemáticos e os algoritmos para achar o W_{opt} apresentados na Seção 5.2.1 não estão preparados para lidar com o dinamismo. Apenas se o comportamento das cargas externas fossem previsíveis, poderia ser possível investigar um algoritmo mais sofisticado. Este experimento tem o objetivo de mostrar que isto não é necessário.

Experimentos anteriores mostraram que valores de W próximos ao W_{opt} produzem normalmente bons resultados para a aplicação AMS ring. Nesse experimento, em especial, o escalonador dinâmico é capaz de modificar a alocação inicial das tarefas que ainda não se encontram em execução de acordo com as variações do ambiente e da execução da aplicação.

Os tempos de execução dos dois algoritmos *ring* são apresentados na Figura 5.12 e na Tabela 5.7 juntamente com um valor teórico de limite inferior para o tempo de execução

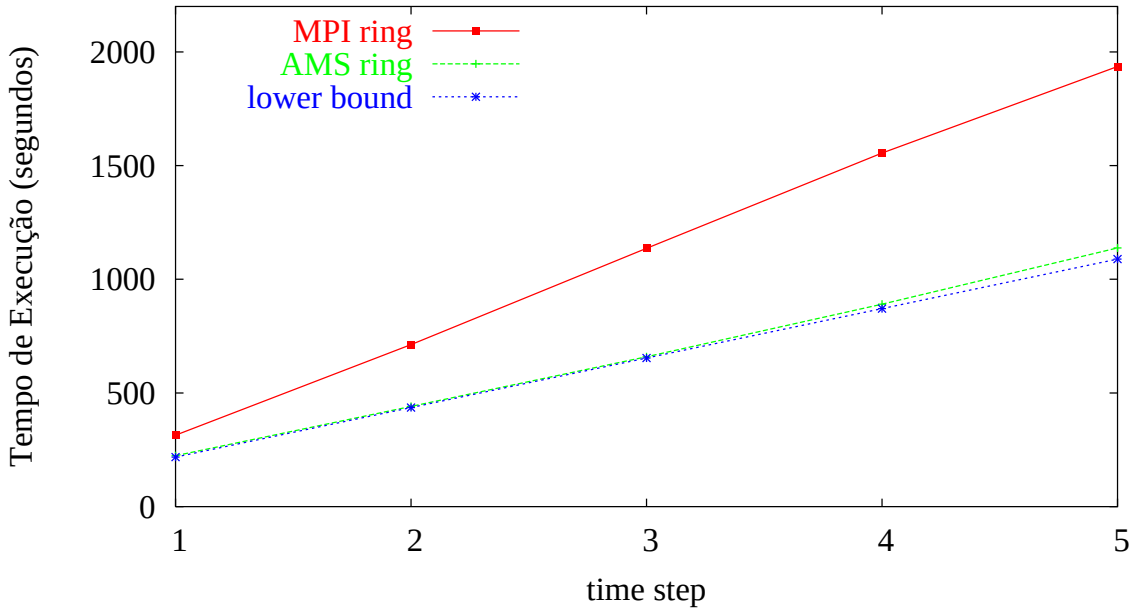

Figura 5.12: Desempenho do MPI ring e AMS ring $W = 48$ com 200.000 partículas em ambientes dinâmicos compartilhados.

considerando a quantidade de *time steps* crescentes. O limite inferior (*lower bound*) foi calculado baseado no poder computacional total disponível, balanceamento de carga perfeito e sem considerar o custo de comunicação. Como esperado, o fraco desempenho da aplicação MPI ring é causado pela sua incapacidade de modificar sua alocação de processos e seu grau de paralelismo. Enquanto não existe suporte para modificar o grau de paralelismo, a modificação da alocação inicial iria requerer um ambiente de execução sofisticado para monitorar e tomar decisões durante a execução da aplicação. Uma das razões que faz com que essa classe de aplicações não seja considerada apta a ser executada em ambientes não dedicados é que uma única carga extra (programa) pode acarretar uma piora no desempenho de até 75% em relação a versão alternativa proposta e de até 79% em relação ao limite inferior, como mostra a Tabela 5.7.

A abordagem adotada pelo AMS ring que utiliza um sistema gerenciador de aplicações e que está baseado no modelo alternativo de execução possibilita um desempenho próximo do ótimo, onde os tempos de execução se encontram 4,5% do limite inferior, valor este que não inclui as sobrecargas do sistema operacional e do *AMS*, assim como os custos de comunicação e as sincronizações ao final de cada *time step*.

5.3.7 Ambiente Grade

O último experimento a ser apresentado considera um ambiente grade geograficamente distribuído, compartilhado e heterogêneo composto de dois *sites* interconectados

através da RedeGiga. Todos os processadores disponíveis executam Linux Fedora Core 2, Globus Toolkit 2.4 e LAM/MPI 7.0.6 localmente interconectados através de *switches Gigabit* e *fast Ethernet*. O *site* 1 é composto de 22 processadores Pentium IV 2.6 GHz localizados na Universidade Federal Fluminense (UFF) enquanto o *site* 2 tem 21 processadores Pentium II 400 MHz localizados na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Para avaliar um cenário compartilhado real, o programa que simula a carga das máquinas através de *traces* reais, PlayLoad [47], foi usado no *site* 1 onde os recursos estavam dedicados ao experimento. Esta ferramenta foi utilizada para gerar uma carga nos processadores do *site* 1 que simulam um computador de mesa com carga moderada (o *trace* utilizado está disponível em [47]). As máquinas do *site* 2 não estavam dedicadas ao experimento, logo cargas externas podiam entrar ou sair das máquinas a qualquer momento. O desempenho relativo de uma máquina do *site* 2 é, na média, metade do desempenho de uma máquina do *site* 1 executando o PlayLoad com carga moderada. Além disso, a latência de comunicação entre os *sites* 1 e 2 é de uma ordem de magnitude maior do que a comunicação interna a cada *site*. Uma visão geral do ambiente disponível para as aplicações, pode ser vista na Figura A.4 do Apêndice.

Quatro cenários foram avaliados para um *time step* do algoritmo N-body com 200.000 partículas, executando no ambiente descrito anteriormente:

(1) execução da aplicação MPI ring utilizando todos os recursos disponíveis;

(2) execução da aplicação MPI ring de acordo com o algoritmo Processors_Choice() (apenas nos recursos do *site* 1);

(3) execução da aplicação AMS ring de acordo com o algoritmo Find_Best_W() sem utilizar o escalonador dinâmico;

(4) execução da aplicação AMS ring de acordo com o algoritmo Find_Best_W() com o escalonador dinâmico.

Devido ao dinamismo e compartilhamento desse ambiente, todos os resultados desse experimento foram gerados a partir da média aritmética do tempo de 10 execuções.

Como pode ser visto na Tabela 5.8, o desempenho da aplicação MPI ring quando forçada a utilizar todos os recursos é desastroso. O poder computacional baixo e compartilhado oferecido pelas máquinas do *site* 2 degradam o desempenho da aplicação, de maneira que (como escolhido pelo algoritmo Processors_Choice()) apenas os recursos do *site* 1 devem ser utilizados pela aplicação MPI ring. Embora o tempo de execução obtido no cenário (2) seja praticamente a metade em relação ao cenário (1) devido ao poder computacional

Tabela 5.8: Tempo de execução (em segundos) do MPI ring e AMS ring com $W = 60$ para 200.000 partículas em um ambiente grade composto de dois *sites* geograficamente distribuídos.

	MPI ring		AMS ring	
	(1)	(2)	(3)	(4)
valor médio	4.292,3	2.285,5	790,7	533,3
desvio padrão	855,7	315,9	79,5	66,4
intervalo de	5.042,4	2.481,4	860,3	582,5
confiança de 95%	3.542,3	2.089,7	721,0	484,0

dos recursos do *site* 1, mesmo assim ele pode ser considerado fraco devido a incapacidade da aplicação MPI ring de se adaptar a heterogeneidade e compartilhamento dos recursos.

Ao contrário, a aplicação AMS ring estática e mais ainda, a dinâmica alcançaram desempenhos bem superiores aproveitando os recursos dos *sites* disponíveis, de acordo com o poder computacional de cada um deles. No cenário (3), como o escalonador dinâmico do *EasyGrid AMS* estava desabilitado, a melhora do desempenho em relação ao cenário (2) se deve a habilidade do escalonador estático de alocar as tarefas apropriadamente (foi informado ao escalonador HEFT a diferença de capacidade de processamento entre os sites) e aos benefícios do modelo de execução alternativo. Como o ambiente grade estava compartilhado com outros usuários (e suas aplicações), a aplicação AMS ring com o escalonador híbrido é capaz de adaptar sua execução as variações do ambiente alcançando um desempenho ainda melhor. Devido as dependências de comunicação da aplicação AMS ring, estratégias de balanceamento de carga tradicionais não puderam ser utilizadas, desse modo o *EasyGrid AMS* utiliza um escalonador dinâmico inovador para aplicações fortemente acopladas, mais detalhes sobre o escalonador dinâmico podem ser encontrados em [116].

A principal conclusão desses experimentos é que a aplicação MPI ring tradicional é incapaz de lidar com um ambiente grade típico e de aproveitar efetivamente recursos que são naturalmente heterogêneos e compartilhados. Todavia, o mesmo algoritmo adaptado para o modelo de execução alternativo pode ser usado em conjunto com um sistema gerenciador de aplicação como o *EasyGrid AMS* para executar eficientemente em tais ambientes.

É importante destacar que nos experimentos apresentados, o valor de W não se modifica durante toda execução da aplicação. Entretanto, ao executar uma aplicação N-corpos com centenas (milhares) de *time steps* é possível ajustar o valor de W dinamicamente, quando ocorrerem mudanças no ambiente que necessitem isso, no início de cada *time step*. É importante destacar que não é necessário realizar nenhuma modificação ou ajuste no modelo de execução alternativo, uma vez que ele já foi proposto para suportar a maleabilidade. Porém, do ponto de vista de implementação seria necessário adicionar maleabilidade

a aplicação N-corpos, através de uma adaptação específica do middleware *EasyGrid AMS* para aplicações *ring*, o que não faz parte do escopo deste trabalho de tese.

5.4 Resumo

Este capítulo apresentou a validação do modelo alternativo de execução proposto neste trabalho de tese para execução de aplicações compostas por tarefas que se comunicam entre si. Para isso foi utilizado o sistema gerenciador de aplicações MPI, *EasyGrid AMS*, que adota o modelo de execução proposto nesta tese, 1PTask, e que tem como função controlar a execução de aplicações MPI nas grades computacionais.

Inicialmente, foi definido o modelo de comunicação para representar a troca de mensagens entre processos. Em especial, essa seção mostra que existe uma sobrecarga de comunicação para se enviar e receber mensagens, inclusive entre tarefas/processos localizados no mesmo processador. Como os modelos arquiteturais anteriores eram baseados na utilização do modelo 1PProc, que executa apenas um processo por processador, eles consideravam a comunicação interna ao processador desprezível.

Em seguida, foi apresentada uma análise detalhada da aplicação N-corpos projetada para ambientes de alto desempenho tradicionais (*cluster* e supercomputadores). A partir deste estudo, foi proposta uma versão alternativa para ambientes heterogêneos, juntamente com uma estratégia inovadora para otimizar o seu desempenho. Por último, foi apresentada a validação do modelo 1PTask para aplicações paralelas compostas de tarefas com dependências (foi utilizada a aplicação N-corpos). Os resultados apresentados mostraram que o modelo proposto permite que a aplicação se adapte às variações do ambiente, utilizando de maneira mais apropriada os recursos, aumentando o seu desempenho. Por outro lado, as aplicações que utilizam o modelo tradicional 1PProc têm seu desempenho limitado pela máquina mais lenta, prejudicando sua execução em ambientes heterogêneos, não dedicados e dinâmicos. Uma grande contribuição deste capítulo foi mostrar a viabilidade de se executar aplicações fortemente acopladas nas grades computacionais, através do modelo de execução 1PTask.

Capítulo 6

Sensibilidade da Estratégia Proposta em Relação à Precisão dos Modelos da Aplicação e da Arquitetura

Como apresentado na Seção 2.1, devido à complexidade de se executar as aplicações nos ambientes grade, o controle da execução das aplicações deve ser feito através de um sistema gerenciador. Mais precisamente, se o objetivo principal é a execução eficiente de uma única aplicação, então os sistemas gerenciadores da aplicação (AMS) devem ser utilizados. Nesses sistemas, as informações sobre a aplicação e sobre a grade, necessárias para que a aplicação seja executada eficientemente, são passadas através do modelo da aplicação e da arquitetura, respectivamente. Consequentemente, o desempenho dos sistemas gerenciadores está diretamente ligado as informações fornecidas por esses modelos. Entretanto, nem sempre as estimativas fornecidas por esses modelos são 100% precisas. Considerando a natureza das aplicações e dos ambientes grades, estimativas sobre o peso das tarefas da aplicação, custo das mensagens e poder computacional dos recursos são difíceis de serem estimadas com precisão.

Desse modo, o principal objetivo deste capítulo é investigar a sensibilidade da estratégia de escalonamento híbrida a (falta de) precisão das estimativas dos modelos da aplicação e arquitetural. No escalonamento híbrido avaliado, primeiramente é realizado um escalonamento estático, que pode utilizar heurísticas mais elaboradas visto que seus respectivos tempos de execução não afetam o desempenho da aplicação. Em seguida, é utilizado um escalonamento dinâmico, que emprega heurísticas mais simples que se baseiam nas decisões do escalonamento estático e nas mudanças do ambiente para reescalonar eficientemente as tarefas da aplicação. É importante destacar que tanto o escalonamento estático, como também o dinâmico são baseados no modelo alternativo de execução 1PTask. Além disso, cabe ressaltar que a estrutura de escalonamento dinâmico avaliada neste trabalho de tese,

assim como as heurísticas dinâmicas utilizadas foram propostas em [116].

Ao longo deste capítulo serão descritas e analisadas algumas heurísticas de escalonamento estática, bem como a sua influência no desempenho da estratégia híbrida. Entretanto, a principal contribuição deste capítulo é avaliar a sensibilidade da estratégia proposta (escalonamento híbrido baseado no modelo de execução 1PTask) especialmente quando os valores estimados sobre as tarefas da aplicação forem imprecisos. Ou seja, identificar o impacto de estimativas erradas ou imprecisas no desempenho da aplicação.

Inicialmente foram avaliadas as aplicações do tipo *bag-of-tasks*, por se tratarem de aplicações mais simples, compostas de tarefas que não se comunicam entre si. Em seguida, são então analisadas aplicações paralelas com relações de precedências entre suas tarefas, o que aumenta a complexidade do problema.

6.1 Escalonamento Híbrido para Aplicações com Tarefas Independentes

O problema de escalonar tarefas independentes é na sua forma geral NP-completo [87], já tendo sido bastante estudado, inclusive para ambientes heterogêneos [23, 87, 90]. Ao definir o problema, uma ou mais funções objetivos podem ser utilizadas. Neste trabalho de tese, a função objetivo especificada corresponde à minimização do tempo de fim da execução da aplicação de entrada, também chamado de *makespan*.

Como discutido na Seção 3.3.1, dependendo do modelo de computação adotado, diferentes características são representadas. As heurísticas de escalonamento são construídas em função do modelo escolhido. A seguir, como exemplo, são apresentadas três heurísticas que utilizam o modelo de processadores uniformes (Seção 3.3.1) que é adotado neste trabalho: LPT [69, 73], BESTFIT [95] e MULTIFIT [38, 68]. Essas heurísticas foram escolhidas pelo fato de produzirem escalonamentos eficientes para o modelo de processadores uniformes (consistentes e heterogêneos). É importante destacar que heurísticas mais recentes (como por exemplo, MinMin, MaxMin e Sufferage) foram implementadas baseadas no modelo de Matriz (ao invés do modelo de processadores uniformes), sendo várias dessas heurísticas apenas adaptações das heurísticas originais (LPT, por exemplo), para o modelo de Matriz [30].

A seguir, cada um desses algoritmos é explicado detalhadamente. Como definido na Seção 3.3.1, m corresponde a quantidade de processadores em P da grade, n ao total de tarefas da aplicação em V e $csi(p_i)$ ao índice de retardo computacional do processador $p_i \in P$. Além disso, LT_k é a lista de tarefas alocadas ao processador p_k e W_k corresponde

a soma do peso de todas as tarefas de LT_k.

LPT

O algoritmo LPT (*Largest Processing Time*) escalona tarefas independentes em processadores homogêneos e foi estudado por Graham em [73] que demonstrou que um escalonamento gerado é 33% maior que o ótimo, no pior caso. Já Friesen [69] concluiu que para processadores uniformes (heterogêneo e consistente), modelo adotado nesse trabalho, qualquer escalonamento gerado pelo LPT possui *makespan* entre 52% e 67% maior do que o ótimo, no pior caso.

O pseudo-código dessa heurística é apresentado no Algoritmo 6. Inicialmente, as tarefas são ordenadas decrescentemente de acordo com o seu peso (linha 2). Seguindo essa ordem, é determinado o processador que minimiza o tempo de fim da tarefa considerando a carga computacional W_k relativa às tarefas já alocadas ao processador (linhas 4 a 7). Em caso de empate, o processador é escolhido arbitrariamente. Seja p_k o processador selecionado então, a tarefa v_i é alocada na sua lista de tarefas LT_k (linha 8) e o peso dessa tarefa é acrescentado ao acumulador de pesos W_k (linha 9).

Algoritmo 6 : LPT*()*

1 $\forall\, p_k \in P, W_k \leftarrow 0; LT_k \leftarrow \langle\rangle$;
2 $listaTarefas \leftarrow$ tarefas em ordem decrescente de peso $\varepsilon(\ )$;
3 enquanto ($listaTarefas \neq 0$)
4 $\quad v_i \leftarrow$ primeiro($listaTarefa$);
5 $\quad$ para j = 1 até m faça
6 $\quad\quad tempo(v_i, p_j) \leftarrow \varepsilon(v_i) \times csi(p_j)$;
$\quad$ fim para
7 $\quad$ escolha a máquina p_k onde $W_k \times csi(p_k) + tempo(v_i, p_k)$ for o mínimo;
8 $\quad LT_k \leftarrow LT_k + \langle v_i \rangle$;
9 $\quad W_k \leftarrow W_k + \varepsilon(v_i)$;
10 $\quad listaTarefas \leftarrow listaTarefas$ - $\langle v_i \rangle$;
fim enquanto

BESTFIT

O algoritmo chamado de BESTFIT [95], tenta alocar as tarefas no processador que for deixar menos espaço livre, em relação a um limite pré-estabelecido, que indica o total de tempo que cada processador pode vir a receber (onde o tempo de cada tarefa v_i em um processador p_j é igual a $\varepsilon(v_i) \times csi(p_j)$). Esse limite é calculado da seguinte maneira:

$$lInferior = \frac{\sum_{\forall v_i \in V} \varepsilon(v_i)}{\sum_{\forall p_j \in P} \frac{1}{csi(p_j)}} \tag{6.1}$$

Mais especificamente, o limite $lInferior$ (Equação 6.1) indica a quantidade de tempo máxima que cada processador pode receber, tendo em vista os índices de retardo $csi(p_j)$ $\forall p_j \in P$. O pseudo-código da heurística BESTFIT pode ser visto no Algoritmo 7, que recebe como parâmetro o limite $lInferior$. Seja $listaTarefas$ a lista de tarefas ordenadas não crescentemente pelo peso de computação $\varepsilon(\ )$ (linha 2). Seguindo essa ordem, é encontrado o processador onde a primeira tarefa da lista encaixa melhor. O "melhor encaixe" é avaliado da seguinte forma. A carga $carga(p_j)$ de cada processador p_j incluindo o peso $\varepsilon(v_i)$ da tarefa v_i é calculada (linha 6). Em seguida, para cada um dos processadores p_j, é calculado a diferença entre o $lInferior$ e $carga(p_j)$. O processador p_k selecionado é aquele que possuir a menor diferença positiva (linhas 7 e 8). Caso não exista tal processador, a tarefa é alocada no processador p_k que minimize o *makespan* (linha 9). A tarefa v_i é então alocada na lista de tarefas LT_k (linha 10) e o peso dessa tarefa é acrescentado ao acumulador de pesos W_k (linha 11). Os passos acima são repetidos até que todas as tarefas da aplicação sejam alocadas.

Algoritmo 7 : BESTFIT*(lInferior)*

1 $\forall p_k \in P, W_k \leftarrow 0; LT_k \leftarrow \langle\rangle$; $\forall p_k, carga(p_k) \leftarrow 0$;
2 $listaTarefas \leftarrow$ tarefas em ordem decrescente de peso $\varepsilon(\ )$;
3 enquanto ($listaTarefas \neq 0$)
4 $\quad v_i \leftarrow$ primeiro($listaTarefa$);
5 $\quad$ para j = 1 até m faça
6 $\quad\quad carga(p_j) \leftarrow W_j \times csi(p_j) + \varepsilon(v_i) \times csi(p_j)$;
$\quad$ fim para
7 $\quad P' \leftarrow \langle\rangle; \forall p_j \in P$ tal que $lInferior - carga(p_j) \geq 0$ então $P' \leftarrow P' + \langle p_j \rangle$;
8 $\quad$ se ($P' \neq \langle\rangle$) então $p_k \leftarrow \min_{\forall p_j \in P'}\{lInferior - carga(p_j)\}$;
9 $\quad$ senão $p_k \leftarrow \min_{\forall p_j \in P}\{W_j \times csi(p_j) + \varepsilon(v_i) \times csi(p_j)\}$;
10 $\quad LT_k \leftarrow LT_k + \langle v_i \rangle$;
11 $\quad W_k \leftarrow W_k + \varepsilon(v_i)$;
12 $\quad listaTarefas \leftarrow listaTarefas$ - $\langle v_i \rangle$;
fim enquanto

MULTIFIT

A heurística MULTIFIT [38] foi adaptada a partir de técnicas aplicadas ao problema de *bin-packing* [109], tendo como objetivo escalonar tarefas independentes em processadores homogêneos. Essa heurística produz soluções que são, no pior caso, 20% maior do que a solução ótima. Já para processadores uniformes, Friesen [68] provou que a solução

encontrada pelo MULTIFIT é, no pior caso, 40% maior do que a solução ótima.

MULTIFIT é um pouco mais complexa que as duas heurísticas anteriormente apresentadas, pois executa $\log n$ vezes o algoritmo *First Fit Decreasing* (FFD) para conseguir encontrar uma boa solução.

Para uma melhor compreensão da heurística MULTIFIT, primeiramente será explicado o algoritmo FFD, que tem como entrada o parâmetro C calculado por MULTIFIT(). Esse parâmetro representa a carga máxima que cada processador pode receber. Na heurística FFD (Algoritmo 9), as tarefas também são ordenadas decrescentemente de acordo com o seu peso (linha 2), sendo que os processadores são ordenados decrescentemente de acordo com seu índice de retardo computacional (linha 3). Seguindo essa ordem, a tarefa é alocada no primeiro processador que ela couber. Para saber se uma tarefa cabe ou não no processador, a diferença entre a capacidade máxima que cada processador pode receber C (parâmetro passado pelo MULTIFIT) e a carga corrente $carga(p_k)$ é calculada (linhas 8 a 12). Se existir tal processador, a tarefa é alocada e as listas de tarefas e carga desse processador são atualizadas (linhas 13 a 16). O algoritmo termina caso alguma tarefa não caiba em nenhum processador, retornado uma falha (linha 17), ou todas as tarefas sejam alocadas, retornando sucesso (linha 18).

Devido a essa influência da capacidade C na heurística FFD, o algoritmo MULTIFIT executa várias vezes o algoritmo FFD, variando-se C até que uma boa solução seja encontrada. A dificuldade está em encontrar um bom valor C. Uma forma de se chegar a este valor é executar a heurística FFD testando todos os valores entre $lInferior$ (Equação 6.1), que pode ser considerado o limite inferior, e o *makespan* encontrado pelo algoritmo LPT, considerado um limite superior e denotado $lSuperior$. Como esse intervalo pode ser muito grande, a solução encontrada foi usar uma pesquisa binária para achar uma boa solução neste intervalo. O algoritmo MULTIFIT, cujo o pseudo-código é apresentado no Algoritmo 8, inicia C como sendo a média aritmética entre $lInferior$ e $lSuperior$ (linhas 1 e 2). Executa-se a seguir o algoritmo FFD (linha 4) e avalia-se o resultado retornado que pode ser um valor menor do que o $lSuperior$ ou falhar (não consegue alocar pelo menos uma tarefa). Em caso de falha, um $lInferior$ recebe C (linha 6), caso contrário $lSuperior$ é que recebe C (linha 7). O novo C é calculado conforme a linha 8 e esse processo se repete enquanto $lSuperior$ for diferente de $lInferior$.

Algoritmo 8 : MULTIFIT*(lInferior)*

1 $lSuperior \leftarrow LPT()$;
2 $C \leftarrow (lSuperior + lInferior)/2$;
3 enquanto $lSuperior \neq lInferior$ faça
4 $\quad achou \leftarrow FFD(C)$;
5 $\quad$ se $(\backsim achou)$
6 $\quad\quad lInferior \leftarrow C$;
$\quad$ senão
7 $\quad\quad lSuperior \leftarrow C$;
$\quad$ fim se
8 $\quad C \leftarrow (lSuperior + lInferior)/2$;
fim enquanto

Algoritmo 9 : FFD*(C)*

1 $\forall p_k \in P, W_k \leftarrow 0; LT_k \leftarrow \langle\rangle; \forall p_k, carga(p_k) \leftarrow 0$;
2 $listaTarefas \leftarrow$ tarefas em ordem decrescente de peso $\varepsilon(\ )$;
3 $listaProc \leftarrow$ processadores em ordem crescente de acordo com $csi(\ )$;
4 enquanto $(listaTarefas \neq 0)$
5 $\quad v_i \leftarrow$ primeiro$(listaTarefa)$;
6 $\quad achou \leftarrow$ falso;
7 $\quad p_k \leftarrow$ inicio$(listaProc)$;
8 $\quad$ enquanto $((p_k \leftarrow$ proximo$(listaProc) \neq$ NULL $) \wedge (\backsim achou))$
9 $\quad\quad carga(p_k) \leftarrow W_k \times csi(p_k) + \varepsilon(v_i) \times csi(p_k)$;
10 $\quad\quad$ se $(C - carga(p_k) \geq 0)$
11 $\quad\quad\quad achou \leftarrow$ verdadeiro;
12 $\quad\quad\quad proc \leftarrow k$;
$\quad\quad$ fim se
$\quad$ fim enquanto
13 $\quad$ se $(achou)$
14 $\quad\quad LT_{proc} \leftarrow LT_{proc} + \langle v_i \rangle$;
15 $\quad\quad W_{proc} \leftarrow W_{proc} + \varepsilon(v_i)$;
16 $\quad\quad$ listaTarefas $\leftarrow$ listaTarefas - $\langle v_i \rangle$;
$\quad$ senão
17 $\quad\quad$ retorne falso;
$\quad$ fim se
fim enquanto
18 retorne verdadeiro;

6.1.1 Análise Experimental - Simulação

Os três algoritmos apresentados (LPT, BESTFIT e MULTIFIT) são heurísticas de escalonamento que não garantem a solução ótima. Uma grande quantidade de simulações utilizando esses três algoritmos de escalonamento para 28 ambientes heterogêneos consistentes foram realizadas (total de 16.800 experimentos) com o objetivo de analisar os respectivos comportamentos. Através dos resultados obtidos não foi possível identificar

situações específicas onde cada um desses algoritmos obteve melhores resultados. Na maioria dos casos (15781 experimentos) eles produziram resultados iguais (empate) entre pelo menos dois desses algoritmos.

Tabela 6.1: Comparação dos três algoritmos: LPT, BESTFIT e MULTIFIT

Algoritmos	#Vitórias	Qualidade
LPT	507	1.0009
BESTFIT	7	1.0011
MULTIFIT	505	1.0003

A Tabela 6.1 mostra o desempenho dos três algoritmos nas 5.600 execuções realizadas para cada algoritmo, onde a coluna #Vitórias indica o número de vitórias de cada algoritmo em relação aos outros dois. Uma vitória representa que o algoritmo obteve um escalonamento com *makespan* menor do que os dois outros algoritmos. Já a coluna Qualidade representa a média do grau de degradação do algoritmo em relação ao melhor *makespan* encontrado. Para saber a qualidade do resultado produzido por um algoritmo de escalonamento basta dividir o valor encontrado por esse algoritmo pelo valor encontrado pelo melhor algoritmo para uma dada instância. Quanto mais próxima de 1 (um) for a qualidade, melhor o resultado (menor a degradação). Os resultados mostram que o algoritmo LPT obteve duas vitórias a mais do que o algoritmo MULTIFIT mostrando que os dois algoritmos são equivalentes considerando este critério. Os valores apresentados na coluna Qualidade mostram que os algoritmos apresentam um grau de degradação bem pequeno, sendo que MULTIFIT apresenta uma ligeira vantagem. Outro ponto importante a destacar é que, embora os resultados apresentados para as heurísticas LPT e MULTIFIT sejam equivalentes, a heurística MULTIFIT é mais complexa do que a heurística LPT.

6.1.2 Prioridades para o Escalonamento Híbrido

Independentemente dos resultados obtidos na análise anterior, algumas observações importantes devem ser analisadas quando se trata de uma heurística de escalonamento estático que auxilia o escalonador dinâmico a obter um bom desempenho na execução da aplicação paralela em uma grade computacional. Por ser um ambiente compartilhado, mudanças na carga do sistema provavelmente ocorrerão durante a execução da aplicação, mudanças estas a serem tratadas pelo escalonador dinâmico. Duas características que influenciam o desempenho do escalonador dinâmico, quando se tratando de aplicações com tarefas independentes, foram identificadas através de experimentos: ordenação e distribuição das tarefas nos recursos. É importante lembrar que os experimentos foram feitos utilizando o *EasyGrid AMS*, de maneira que apenas tarefas que não se encontram em execução podem ser reescalonadas pelo escalonador dinâmico (Seção 4.1.2).

Com relação à ordenação estática de tarefas dentro dos recursos, três opções poderiam ser utilizadas: ordem decrescente em relação ao peso da tarefa, crescente e aleatória. Intuitivamente, a melhor opção parece ser a ordenação decrescente, pois ela permite que as tarefas de maior peso executem primeiro. Logo, se durante a execução houver uma mudança na grade (por exemplo, a entrada de uma outra aplicação disputando recursos da grade), as tarefas maiores têm mais chance de terem terminado sua execução, sobrando apenas tarefas menores que serão redistribuídas pelo escalonador dinâmico. A ordenação crescente, por outro lado, executa primeiro as tarefas de menor peso de computação. Logo, caso haja uma perda de oferta computacional na grade, o escalonador dinâmico pode não conseguir uma boa redistribuição para as tarefas remanescentes de pesos maiores (o escalonador dinâmico cede tarefas de apenas um recurso sobrecarregado para os recursos subcarregados [18]). Além disso, essas tarefas maiores podem ter que executar em recursos já sobrecarregados provocando uma piora no tempo de execução da aplicação. A última opção seria uma ordenação aleatória de modo que o desempenho do escalonamento híbrido estaria sempre relacionado a um fator sorte (aleatório).

A distribuição estática dos pesos nos recursos também pode ser feita de três maneiras: alternando as tarefas entre os processadores por ordem de peso, concentrando as tarefas mais pesadas em poucos recursos ou fazendo uma distribuição dos pesos aleatoriamente.

Originalmente, os três algoritmos estáticos apresentados ordenam as tarefas decrescentemente por ordem de peso. Com relação a distribuição das tarefas, o algoritmo LPT, por colocar a tarefa sempre onde esta termina mais cedo, normalmente alterna as tarefas por ordem de peso entre os recursos disponíveis. O algoritmo MULTIFIT, por utilizar o algoritmo FFD que aloca a tarefa no primeiro recurso que couber, concentra as tarefas com maiores tempos computacionais nos primeiros recursos disponíveis. O algoritmo BESTFIT, produz efeito semelhante, pois aloca as tarefas onde se encaixam melhor. O experimento a seguir mostra a influência dessas três heurísticas de escalonamento estático no escalonamento híbrido utilizado no middleware *EasyGrid*.

Efeito Sobre o Escalonamento Dinâmico

Este experimento foi realizado em uma grade computacional com três *sites* distintos, interconectados por um *switch fast Ethernet* (Figura A.1 no Apêndice). Cada *site* contém um conjunto de processadores Pentium IV 2.6 GHz com 512Mb de memória RAM, sob o sistema operacional Linux Fedora Core 2, Globus toolkit 2.4 and LAM/MPI 7.0.6. Os *sites* 1 e 3 contém 8 máquinas cada, enquanto o *site* 2, 13 máquinas. Com o objetivo de avaliar as diferentes características de algumas políticas de escalonamento, os resultados apresentados na Tabela 6.2 foram produzidos utilizando a grade computacional acima,

dedicada a este experimento.

Uma aplicação PSwp com 1.000 tarefas (aplicação *parameter sweep* apresentada na Seção 4.2) foi executada em um ambiente heterogêneo e consistente, onde cargas externas foram executadas em 5 diferentes recursos ao longo do tempo de execução. A aplicação PSwp para esse experimento tem processos heterogêneos, onde as tarefas possuem tempo de processamento não uniforme (metade dos processos requer 1 segundo de tempo de computação, 1/4 5 s e 1/4, 10 s). O objetivo deste experimento é avaliar o impacto que diferentes políticas de escalonamento estático podem causar sobre o escalonamento dinâmico. Além disso, um outro objetivo é também avaliar a influência da ordenação das tarefas especificada estaticamente no esquema de escalonamento híbrido. Para isso foram utilizadas seis heurísticas de escalonamento estáticas: LPT, BESTFIT e MULTIFIT, em suas formas originais, e essas mesmas três heurísticas com as tarefas sendo selecionadas para escalonamento por ordem crescentemente de peso de computação LPT_C, BESTFIT_C e MULTIFIT_C. A aplicação PSwp foi executada utilizando como entrada para o escalonador dinâmico, um escalonamento estático gerado por cada uma das seis heurísticas mencionadas, assumindo que todos os recursos da grade computacional estivessem disponíveis e dedicados. Além disso, são criados dois processos da aplicação PSwp por vez em cada recurso para sobrepor o custo da criação de processos. Para avaliar o desempenho da estratégia híbrida, durante a execução foram colocadas cargas externas em cinco dessas máquinas (de um total de 25 máquinas) fazendo com que a capacidade computacional nesses recursos ficasse reduzida à metade, considerando cinco cenários distintos:

(a) as cinco máquinas perdem metade de sua capacidade desde o início da execução;

(b) após 40 segundos do início da execução;

(c) após 80 segundos;

(d) após 120 segundos;

(e) após 160 segundos.

Os valores na Tabela 6.2 correspondem a média aritmética de três execuções para cada um desses cenários (em segundos).

Os resultados apresentados na Tabela 6.2 mostram que o escalonamento estático realmente influencia no desempenho do escalonador híbrido. Neste experimento, as heurísticas com ordenação estática decrescente proporcionaram melhores resultados que as heurísticas com ordenação estática crescente. A explicação para esse comportamento é que o escalonador dinâmico cede as tarefas do final da fila de um recurso sobrecarregado para

Tabela 6.2: Influência do escalonador estático no escalonamento híbrido do EasyGrid

cenário	LPT	LPT_C	BESTFIT	BESTFIT_C	MULTFIT	MULTFIT_C
(a)	200.23	212.61	203.14	214.02	203.55	214.56
(b)	189.14	213.36	198.46	200.28	201.32	203.96
(c)	186.63	213.70	196.72	197.40	197.44	197.00
(d)	184.07	213.52	203.31	211.26	200.95	210.51
(e)	175.34	196.67	194.26	206.94	194.40	205.59

os recursos subcarregados [18] e, como na ordenação decrescente as tarefas do final da fila são aquelas de menor peso de computação, o escalonador dinâmico consegue uma distribuição melhor das tarefas. Esse tipo de comportamento fica bem evidente na heurística LPT, onde os resultados com a ordenação decrescente foram sempre muito melhores do que com a ordenação crescente. Isto se deve ao fato de que nessa heurística as tarefas são distribuídas alternadamente entre os recursos (as tarefas mais pesadas não ficam concentradas nos primeiros recursos disponíveis, pois a função objetivo do LPT é alocar no processador que minimiza o tempo de fim da tarefa selecionada) assim, as tarefas de peso de computação menor serão espalhadas nos diferentes recursos da grade para serem executadas no final. Já, nas heurísticas BESTFIT e MULTIFIT, as tarefas mais pesadas ficam concentradas nos primeiros recursos. Deste modo, se um desses recursos ficar sobrecarregado, ele cederá tarefas do final da fila, supostamente mais longas, dificultando um bom balanceamento quando há necessidade de reescalonamento.

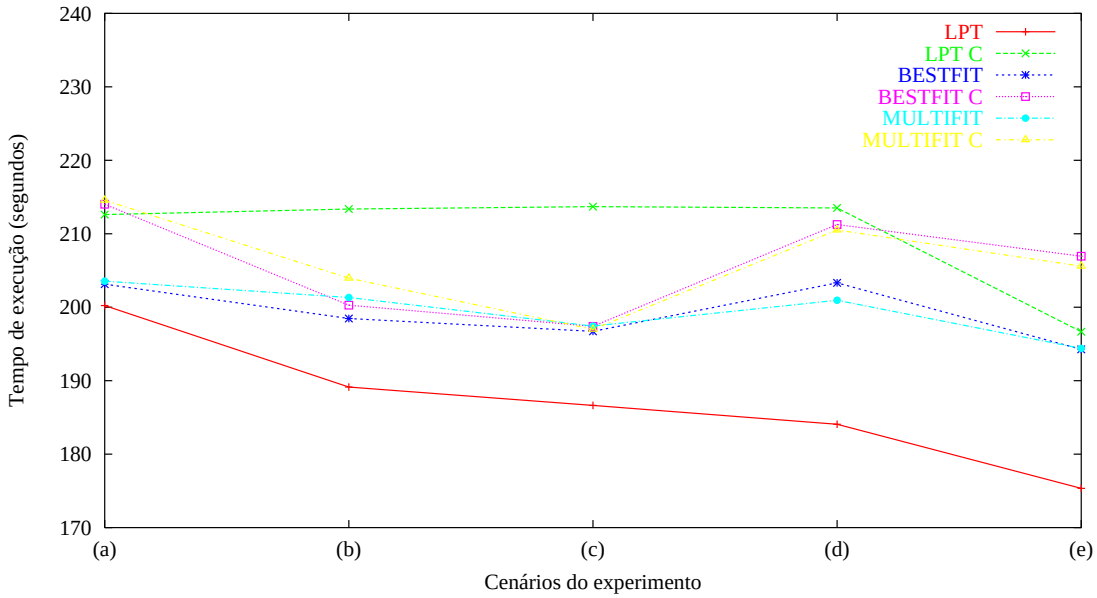

Figura 6.1: Influência do escalonamento estático no desempenho do escalonador híbrido.

A Figura 6.1 mostra o desempenho do escalonador híbrido com seis políticas estáticas

diferentes em cinco cenários (a), (b), (c), (d) e (e). Os cenários aparecem no gráfico de tal forma que a capacidade computacional total oferecida desde o início da execução da aplicação em (a) < (b) < (c) < (d) < (e). De acordo com essa ordenação, o comportamento esperado é que o tempo de execução da aplicação diminua ao longo dos cenários (pois a capacidade computacional disponível aumenta). Porém, somente o LPT apresentou tal comportamento constante e uniforme ao longo dos cinco cenários, pois nessa heurística, além das tarefas estarem ordenadas decrescentemente, elas são distribuídas alternadamente de acordo com a capacidade de cada processador. Logo, quando algum processador perde capacidade de processamento (devido às cargas externas inseridas), este cederá tarefas menores que serão divididas entre os processadores subcarregados. Os algoritmos BESTFIT e MULTIFIT tiveram um comportamento parecido, pois as tarefas também são ordenados pelo decrescentemente de acordo com seus pesos, mas tendem concentrar as tarefas de maior peso computacional em poucos processadores, conseguindo diminuir o tempo de execução da aplicação ao longo dos cenários. Porém, no cenário (d) (após 120 segundos, cargas externas são inseridas) o escalonador dinâmico não conseguiu balancear as tarefas adequadamente entre os recursos prejudicando o tempo de execução da aplicação. Os algoritmos com ordenação crescente apresentaram resultados bem inferiores aos algoritmos com ordenação decrescente. O pior desempenho foi do LPT_C pois, ao contrário do LPT, as tarefas de maior peso são reescalonadas no final da execução o que afeta negativamente o desempenho do escalonador dinâmico.

6.1.3 Impacto da Ausência de Informação sobre o Ambiente no Escalonamento de Tarefas de Aplicações *bag-of-tasks*

Esta subseção reporta, através de experimentos, como a falta de informação sobre o ambiente grade, disponível para a execução, afeta o desempenho da aplicação. Em especial, é analisado os benefícios da estratégia híbrida de escalonamento quando os dados sobre o estado dos recursos do ambiente forem imprecisos. Para isso, foi utilizado o mesmo ambiente grade do experimento anterior, Figura A.1 no Apêndice.

Com o objetivo de também avaliar o desempenho do escalonamento de acordo com o aumento no número de tarefas, foi utilizada a mesma aplicação PSwp do experimento anterior com 1.000, 5.000 e 10.000 processos cada. Foram utilizadas quatro políticas de escalonamento distintas em um ambiente heterogêneo, porém estável. Vale lembrar que a aplicação PSwp para esse experimento tem processos heterogêneos, onde as tarefas tem tempo de processamento não uniformes (metade dos processos requerem 1 s (segundo) de tempo de computação, 1/4 requer 5 s e 1/4, 10 s). As políticas de escalonamento utilizadas foram:

Tabela 6.3: Comparação de diferentes políticas de escalonamento através do EasyGrid

N		(1)	(2)	(3)	(4)
1000	mspan	392.81	211.86	191.92	**191.26**
	stdev	68.65	8.37	0.44	0.26
5000	mspan	1926.15	969.07	960.51	**954.48**
	stdev	367.01	7.30	1.96	0.75
10000	mspan	3522.50	1926.84	1911.79	**1908.97**
	stdev	669.53	6.98	2.22	0.99

(1) escalonamento round-robin utilizado pelo MPI, sem escalonamento dinâmico

(2) escalonamento dinâmico considerando o escalonamento round-robin em (1)

(3) escalonamento LPT, sem escalonamento dinâmico

(4) escalonamento dinâmico utilizando o escalonamento estático em (3)

Enquanto a estratégia utilizada em (1) não utiliza nenhuma informação sobre a aplicação e sobre o ambiente, o escalonador dinâmico da política (2) é capaz de, pelo menos, identificar diferenças na capacidade de processamento dos recursos a partir do tempo de fim das tarefas executadas, pois considera que todas as tarefas têm o mesmo peso. As políticas (3) e (4) se beneficiam das informações conhecidas sobre a aplicação. Por exemplo, para aplicações *bag-of-tasks*, que não têm dependências, utilizam a estimativa do tempo de execução das tarefas da aplicação.

A Tabela 6.3 apresenta os resultados obtidos por cada política, com mspan and stdev indicando o tempo de execução real e o desvio padrão, respectivamente. O desvio padrão do tempo de término dos processadores é utilizado para analisar como as tarefas foram distribuídas entre os diferentes recursos heterogêneos. Apesar da coluna (1) mostrar os resultados de uma alocação estática ruim, os resultados da coluna (2) mostram a capacidade do escalonador dinâmico de melhorar essa distribuição. Porém, os resultados da coluna (2) não são tão bons quanto os da coluna (4), pois o escalonador dinâmico da política (2) não possui informações precisas sobre as tarefas. Os tempos de execução obtidos em (3) indicam que para aplicações PSwp, escalonamentos estáticos eficientes podem ser gerados quando as características das aplicações são consideradas e o ambiente grade não sofre modificações durante a execução (o que, na prática, é improvável). Os melhores resultados (em negrito) foram alcançados pelo escalonador híbrido (4), onde até a boa alocação inicial fornecida pelo escalonador estático foi levemente melhorada. Isto ocorreu, devido ao fato de que nesse ambiente semi-controlado ocorrem pequenas variações na rede e na capacidade de processamento durante a execução.

Ambiente Dinâmico

O próximo experimento investiga a limitação de um escalonador estático em um ambiente que sofre alguma mudança após o início da execução do programa. Para isso foram utilizadas a aplicação sintética PSwp e a aplicação *Thermions*, apresentada na Seção 4.4. Para esse experimentos foram utilizadas 30 máquinas disponíveis na UFF, onde uma máquina executava o gerenciador global *GM*, outra executava o gerenciador local *SM* e as outras 28 máquinas executavam um gerenciador da máquina *HM*. O tempo de duração das tarefas da aplicação PSwp é de 5 segundos e da aplicação *Thermions* varia entre 4 e 4,5 segundos, considerando um processador Pentium IV 2.6 GHz dedicado. Além disso, foram testadas aplicações PSwp com 1.000 e 10.000 tarefas e a aplicação *Thermions* com 1.000 tarefas sendo que em todas as execuções, foram criados dois processos simultaneamente em um mesmo processador.

Cinco variações no ambiente foram avaliadas:

(1) sem variação (todos os recursos estavam dedicados a execução da aplicação);

(2) 25% das máquinas executam uma carga extra (1 programa *CPU bound*) após o início da execução;

(3) 25% das máquinas executam duas cargas extras (2 programas *CPU bound*) após o início da execução;

(4) 50% das máquinas executam uma carga extra após o início da execução;

(5) 75% das máquinas executam uma carga extra após o início da execução;

Inicialmente, o escalonador estático percebe (através de um programa modelador) que todas as máquinas estão dedicadas e com isso, distribui igualmente as tarefas entre as 28 máquinas disponíveis para executar tarefas da aplicação através do algoritmo LPT. Cada um dos 5 cenários foram avaliados e os resultados foram resumidos na Tabela 6.4. A máquina mais carregada, cenários de (2) a (5), determina o tempo total de execução nas situações em que somente o escalonamento estático é considerado, não sendo o dinâmico utilizado (coluna "Estático"). No cenário (1), os tempos de execuções das aplicações utilizando o escalonador estático e o escalonador híbrido foram praticamente os mesmos, devido a homogeneidade do ambiente. No caso da aplicação PSwp com 10.000 tarefas, a versão com escalonamento dinâmico produziu um resultado melhor pois houve reescalonamento. Mesmo em ambientes compostos de processadores idênticos pequenas variações de processamento podem causar pequenos desbalanceamentos. No cenário (2), as tarefas alocadas aos processadores com uma carga extra perdem apenas 1/3 do poder computacional

disponível, pois duas tarefas da aplicação são executadas concorrentemente. Entretanto, o escalonador dinâmico compensa essa perda remapeando tarefas de máquinas sobrecarregadas para máquinas subcarregadas, balanceando a distribuição das tarefas, quando a abordagem híbrida é utilizada. O tempo de execução é apenas 1,0% (1,3%) acima do ótimo para o caso com 10.000 (1.000) tarefas. Os cenários (3), (4) e (5) mostram que os respectivos tempos de execução são proporcionais as mudanças na capacidade de processamento das máquinas.

Tabela 6.4: Comparação entre o escalonador estático e híbrido em um ambiente que sofre mudanças após o início da execução da aplicação (em segundos)

S	PSwp				Thermions	
	10.000 tarefas		1.000 Tarefas		1.000 Tarefas	
	Estático	Híbrido	Estático	Híbrido	Estático	Híbrido
(1)	1809,9	1802,9	182.4	182,3	161,2	161,3
(2)	2719,9	1970,4	273.7	197,7	241,0	181,8
(3)	3629,4	2066,7	365.3	217,6	321,7	191,7
(4)	2719,4	2170,5	273.7	222,8	241,5	196,8
(5)	2719,5	2412,0	273.5	253,4	241,0	223,0

6.1.4 Impacto de Estimativas Imprecisas no Escalonamento de Aplicações *bag-of-tasks*

Tendo visto que o desempenho da aplicação depende das informações passadas para o escalonador de tarefas, seja ele estático ou dinâmico, através do modelo de aplicação, esta subseção avalia o desempenho da estratégia híbrida de escalonamento quando as informações sobre as tarefas da aplicação forem imprecisas ou erradas.

6.1.4.1 Sem a execução do escalonador dinâmico

O experimento a seguir utiliza apenas 8 processadores homogêneos (Figura A.2 no Apêndice), onde um processador executa o GM, outro o SM e os outros seis restantes um HM cada. Apenas as máquinas com o HM executam tarefas da aplicação (apesar do GM executar a tarefa 0). Utilizando essa configuração, a aplicação *Sched* (descrita na Seção 4.2) contendo 8.820 tarefas foi executada através do *EasyGrid AMS*, tendo como base quatro formas distintas de estimar os pesos das tarefas da aplicação Sched:

(1) Sem conhecimento sobre a aplicação (todas as tarefas foram consideradas com peso igual a 1 unidade)

(2) Tarefas são selecionadas em ordem decrescente do peso da tarefa. O peso da tarefa é dado pela dimensão da entrada D_i associada a tarefa $v_i \in Sched$

(3) Tarefas são selecionadas em ordem decrescente de acordo com a complexidade da heurística de cada experimento de *Sched*

(4) Tarefas são selecionadas em ordem decrescente do peso da tarefa. O peso da tarefa é dado pela metodologia apresentada na Seção 4.2

A escolha desses quatro cenários foi baseada na facilidade de se conseguir informações sobre uma aplicação paralela. O cenário (1) representa a ausência de informação. Por outro lado, os cenários (2) e (3) apresentam formas simples e direta de se conseguir informações sobre as tarefas de uma aplicação, que é a quantidade de dados ou tamanho relativo de cada tarefa, respectivamente. Por último, o cenário (4) mostra uma maneira mais elaborada de se obter informações sobre tarefas de uma aplicação, sendo necessário uma análise do comportamento da aplicação, assim como dos dados que ela recebe.

Todos os escalonamentos de tarefas foram gerados utilizando o algoritmo LPT, tendo como entrada os arquivos de pesos seguindo cada situação listada anteriormente. A Tabela 6.5 apresenta o tempo de execução da aplicação em cada uma das máquinas (média aritmética de três execuções), para cada uma das situações descritas de (1) a (4), sendo que os valores em negrito representam o tempo de execução da aplicação. A última linha da Tabela, mostra também uma previsão do tempo de execução (*makespan*), fornecida pelo LPT considerando a ordenação da situação (4), em cada máquina, utilizando os pesos das tarefas e o índice de retardo computacional de cada máquina como mostrado na Equação 4.2 da Seção 3.3.1.

Tabela 6.5: Variação do tempo de execução de acordo com a precisão da informação (segundos)

	Tempo de execução de cada recursos utilizado (seg)						
cenário	maq1	maq2	maq3	maq4	maq5	maq6	Desvio Padrão
(1)	63.5	**1102.5**	128.5	67.2	1049.1	141.6	461.06
(2)	66.5	**1215.0**	136.8	64.4	938.6	138.1	467.58
(3)	410.6	537.4	**612.8**	369.5	270.3	352.2	115.77
(4)	436.5	436.2	**437.6**	437.4	436.1	435.9	0.65
Previsão	417.9	417.9	417.9	417.9	417.9	417.9	0

Observando a Tabela 6.5, a ausência de informação, cenário (1), sobre as tarefas da aplicação fez com que a distribuição das tarefas entre os recursos disponíveis não fosse feita de maneira adequada, de modo que, enquanto o recurso *maq*1 terminou de executar em 63.5 segundos, o recurso *maq*2 demorou 1.102,5 segundos para executar todas as suas tarefas. Apesar do algoritmo LPT ter distribuído exatamente 1.470 tarefas para cada máquina, a ausência de conhecimento sobre o peso das tarefas fez com que o escalonador estático dividisse as tarefas entre os processadores de forma completamente desbalanceada, considerando os pesos reais. Entretanto, como mostrado no cenário (2), apesar do

escalonador de tarefas ordenar as tarefas da aplicação levando em conta a dimensão da entrada D_i associada a tarefa, o desempenho foi ainda pior, fazendo com que o recurso $maq2$ demorasse 1.215,0 segundos para terminar a sua execução. O motivo da piora do desempenho foi a utilização de uma informação imprecisa (dimensão da entrada D_i), que não representa adequadamente o peso relativo da tarefa.

De acordo com as linhas (3) e (4) da Tabela 6.5, uma informação estática adequada sobre cada tarefa da aplicação leva a um bom desempenho. Na linha (3), a informação utilizada foi a complexidade da heurística de escalonamento executada em cada uma das tarefas da aplicação *Sched*. Como a complexidade da heurística tem uma influência grande no tempo de execução das tarefas, o desempenho da aplicação melhorou, de modo que o tempo de execução da aplicação caiu para 612,8 segundos. Por último, como mostra a linha (4), foi realizado o escalonamento de tarefas estático com o conhecimento preciso sobre a aplicação (conforme os resultados da metodologia apresentada na Seção 4.2). Logo, o desempenho da aplicação foi excelente, pois o escalonador estático (LPT) conseguiu distribuir as tarefas de forma adequada de modo que o tempo de execução da aplicação foi de 437,6 segundos e a diferença entre a máquina que executou as tarefas mais rapidamente e a que executou mais lentamente foi de apenas 1,7 segundos. Além disso, comparando o tempo de execução obtido em cada máquina com o *makespan* previsto por LPT, verifica-se que o escalonamento em (4) está próximo (apenas 4.7% pior) da previsão.

Na coluna Desvio Padrão da Tabela 6.5, o desvio padrão entre os tempos totais de execução de cada máquina é apresentado, confirmando o grau de desbalanceamento entre os recursos, que no caso do cenário (4) é mínimo.

6.1.4.2 Execução Considerando a Estratégia Híbrida

Utilizando a grade Sinergia composta de três *sites* (Figura A.1 no Apêndice) em um ambiente heterogêneo com um total de 29 processadores, porém estável (onde cargas externas foram colocadas em quatro recursos distintos) e a aplicação PSwp com tarefas com pesos não uniformes (com 1000 tarefas), foi realizado um segundo experimento para investigar o impacto de uma estimativa errada nos pesos das tarefas sobre o desempenho da aplicação. Dessa forma, a partir dos pesos originais das tarefas (metade dos processos requer 1 segundo de tempo de computação, 1/4 requer 5 s e 1/4, 10 s) foram geradas aleatoriamente estimativas com até 20, 40, 60, 80 e 100% de erro para mais ou para menos. Por exemplo, a estimativa de uma tarefa de peso 10 com 60% de erro pode variar entre 4 e 16. Para cada um desses percentuais de erro foram gerados aleatoriamente cinco estimativas distintas. Depois, utilizando cada uma dessas cinco estimativas, foi executada a aplicação somente com o escalonador estático e, em seguida, a estratégia híbrida e calculada a média

aritmética dos tempos totais de três execuções.

Para efeito de comparação, representando um escalonamento sem qualquer conhecimento sobre a aplicação, a mesma aplicação foi executada recebendo como entrada uma lista com os pesos das tarefas (1, 5 e 10 segundos) ordenados aleatoriamente, porém para o escalonador estático e dinâmico foi informado que todas as tarefas possuíam o mesmo peso (1 segundo). Foram geradas cinco ordenações aleatórias diferentes, onde cada uma foi executado três vezes, somente com o escalonador estático e, em seguida, com a estratégia híbrida. A média aritmética das cinco execuções aleatórias foi considerado o tempo de execução da aplicação sem conhecimento sobre as tarefas da aplicação.

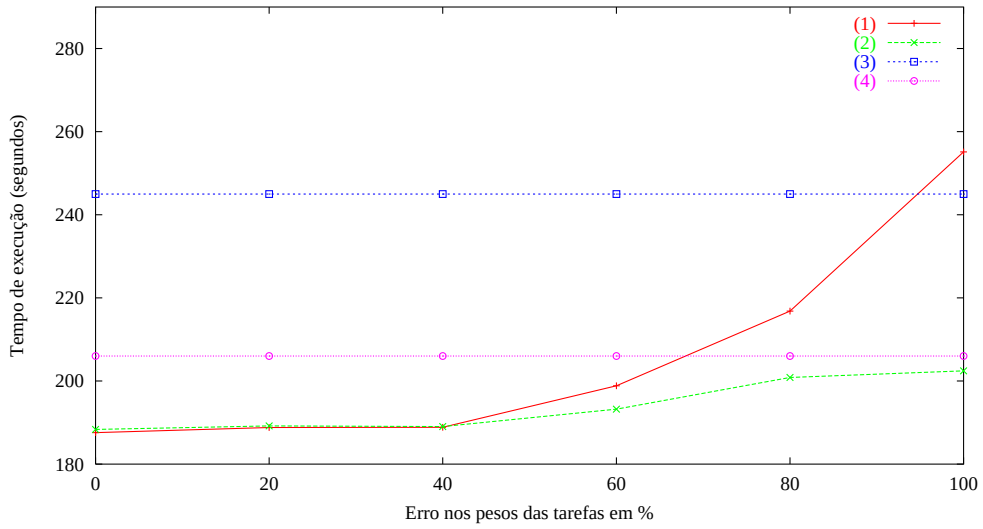

Figura 6.2: Evolução do tempo de execução das aplicações de acordo com o aumento do erro na estimativa dos pesos (tarefas com pesos 1, 5 e 10 segundos)

Quatro cenários são analisados nesse experimento, cuja a evolução do desempenho é apresentada na Figura 6.2:

(1) Aplicação executada com as estimativas com até 0, 20, 40, 60, 80 e 100% de erro utilizando apenas o escalonamento LPT (estático);

(2) Aplicação executada com as estimativas com até 0, 20, 40, 60, 80 e 100% de erro utilizando a estratégia híbrida;

(3) Aplicação executada apenas com o escalonamento LPT (estático) sem informação sobre a aplicação;

(4) Aplicação executada com a estratégia híbrida sem informação sobre a aplicação.

Conforme visto na Figura 6.2, os resultados obtidos para os cenários (1) e (2) comprovam que quanto menos conhecimento (quanto maior o erro) pior o desempenho da apli-

cação, pois os escalonadores não distribuem as tarefas adequadamente entre os recursos disponíveis. Mesmo assim, no cenário (2), o escalonador dinâmico, apesar das informações erradas, consegue perceber as diferenças de capacidade computacional entre os recursos e com isso, o desempenho melhora bastante apesar do escalonamento estático ruim. Outro ponto a destacar nos cenários (1) e (2) é que os escalonamentos com até 20% e 40% de erro produziram resultados quase tão bons quanto os que utilizaram estimativas precisas (0% de erro). Isso aconteceu devido ao fato dos pesos das tarefas utilizadas (1, 5 e 10 segundos) serem percentualmente distantes uns dos outros, de modo que mesmo com até 40% de erro as estimativas geradas praticamente não causaram interferência no escalonamento.

Por outro lado, os cenários (3) e (4) mostram o desempenho da aplicação sem conhecimento sobre as tarefas da aplicação executando apenas com escalonamento estático e com a abordagem híbrida, respectivamente. Os valores apresentados na Figura 6.2 mostram que, na média, a execução da aplicação sem conhecimento sobre os pesos das tarefas é bastante inferior do que quando se tem algum conhecimento. A versão estática sem informação só conseguiu ser um pouco melhor do que a versão estática com até 100% de erro. Diferentemente, o desempenho da versão híbrida sem informação foi sempre pior do que o desempenho da versão com algum conhecimento sobre os pesos das tarefas da aplicação. Mesmo uma informação com até 100% de erro sobre os pesos das tarefas permitiu uma maior eficiência do que a ausência de informação.

Esse mesmo experimento foi refeito, usando exatamente a mesma configuração da grade, para uma aplicação também com 1.000 tarefas, porém os tempos de processamento das tarefas foram modificados para a seguinte configuração: 1/4 dos processos requerem 5s de tempo de computação, 1/4 com 10s, 1/4 com 15s e 1/4 com 20s. Logo, nesse experimento, os pesos das tarefas são (percentualmente) mais próximos uns dos outros. Como observado no gráfico da Figura 6.3, o comportamento das execuções neste segundo experimento é similar ao observado no experimento anterior, entretanto algumas diferenças podem ser notadas. Neste experimento, a execução com até 20% e 40% de erro nos cenários (1) e (2) apresentaram resultados piores do que a aplicação sem erro. Esse comportamento era esperado pois os pesos das tarefas nesse experimento são mais próximos uns dos outros (por exemplo, a estimativa de uma tarefa de peso 20 com 20% de erro pode variar entre 16 e 24 e uma de peso 15 entre 12 e 18), de modo que um erro de 20% pode prejudicar o escalonamento estático ou híbrido. Em outras palavras, dada duas tarefas de pesos X e Y, respectivamente, com apenas 20% de erro no pesos dessas tarefas para o escalonador $X > Y$ (de acordo com as estimativas) e na verdade $X < Y$ (valor real).

Uma segunda diferença foi que com 20% de erro, na média, o escalonamento estático foi melhor que o escalonador dinâmico. Isso ocorre devido ao fato que o escalonador dinâ-

mico detectou que a aplicação estava desbalanceada e assim tarefas foram reescalonadas, porém como a estimativa não era precisa resultados ligeiramente piores foram obtidos. Entretanto, é importante observar que conforme o erro aumenta (prejudicando o escalonador estático) o escalonador dinâmico, percebendo o desbalanceamento, consegue melhorar bastante o tempo de execução da aplicação.

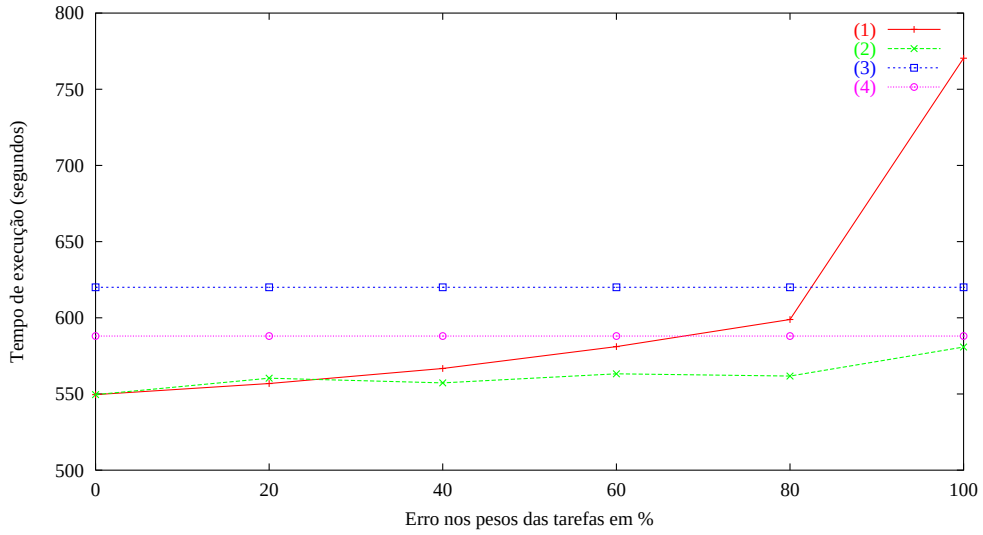

Figura 6.3: Evolução do tempo de execução das aplicações de acordo com o aumento no erro na estimativa dos pesos (tarefas com pesos 5, 10, 15 e 20 segundos)

Na comparação com um escalonamento sem qualquer conhecimento sobre a aplicação, cenários (3) e (4), o desempenho foi similar ao do experimento anterior. Ou seja, o escalonamento estático sem informação só conseguiu ser melhor do que o escalonamento estático com até 100% de erro. Por outro lado, embora o escalonador dinâmico da versão sem conhecimento, (4), tenha melhorado o escalonamento estático inicial ruim, na média o tempo de execução foi sempre pior do que a estratégia híbrida com algum tipo de informação, (2), inclusive com 100% de erro, como pode ser observado na Figura 6.3.

Através dos resultados apresentados nesta seção, é possível concluir que o conhecimento prévio sobre as tarefas das aplicações do tipo *bag-of-tasks* influencia diretamente na eficiência do escalonamento de tarefas desse tipo de aplicação, que por sua vez tem um papel fundamental no desempenho da aplicação. Além disso, os resultados revelam que quanto mais imprecisa for a informação sobre as tarefas, pior será o desempenho da aplicação. Entretanto, esses experimentos mostraram que até mesmo uma informação (peso da tarefa) com até 100% de erro é melhor do que a ausência de informação.

6.2 Escalonamento Híbrido para uma Aplicação Paralela com Dependências entre Tarefas

O escalonamento de aplicações paralelas representadas por GADs quaisquer acrescenta uma dificuldade ainda maior, que é a relação de precedências entre tarefas, com custo de comunicação associado. Esse trabalho de tese propõe o uso de um modelo de comunicação que considera sobrecargas de envio e recebimento nos processadores e a latência de comunicação entre eles.

Para avaliar o desempenho da estratégia de escalonamento híbrida nas aplicações paralelas com dependências foram utilizada apenas as máquinas disponíveis na UFF, utilizando um ambiente distribuído, dedicado e controlado para analisar o desempenho das aplicações paralelas. A configuração virtual do *EasyGrid AMS* pode ser vista na Figura A.5 no Apêndice. São definidos dois *sites* distintos, compostos de 13 e 12 máquinas cada *site*. Do total de 25 máquinas, 22 são dedicadas a execução de tarefas da aplicação, enquanto que duas executam os gerenciadores de cada *site* (SM) e uma executa o gerenciador global (GM).

Para uma análise mais precisa da eficiência da estratégia de escalonamento híbrida, nesta seção são utilizadas aplicações sintéticas cuja a relação de precedências é baseada em aplicações reais existentes. Porém, os tempo de execução das tarefas são controlados, assim como o tamanho das mensagens, que são de 128 bytes. Foram utilizadas aplicações *InTree*, *OutTree* e Diamantes (Seção 3.2).

6.2.1 Impacto da Ausência de Informação sobre o Ambiente no Escalonamento de Tarefas de Aplicações Paralelas com Dependência

O objetivo desta subseção é mostrar a importância do conhecimento sobre o ambiente de execução para estratégia híbrida de escalonamento utilizada neste trabalho. No primeiro experimento as 12 máquinas do primeiro *site* oferecem apenas 50% do seu poder computacional, enquanto que as 12 máquinas do segundo *site* estão dedicadas a execução da aplicação (oferecem 100% do seu poder computacional), o que caracteriza um ambiente heterogêneo, porém dedicado. As aplicações foram executadas através do *EasyGrid AMS* considerando os 4 cenários a seguir:

(1) Escalonamento estático utilizando a heurística HEFT sem conhecimento sobre os recursos (considerou que todos os recursos possuíam a mesma capacidade)

(2) Escalonamento estático utilizando a heurística HEFT considerando que as máquinas do primeiro *site* têm metade do poder das máquinas do segundo *site*

(3) Escalonamento estático definido em (2) e o escalonamento dinâmico é ativado durante a execução da aplicação

(4) Escalonamento estático definido em (1) e o escalonamento dinâmico é ativado durante a execução da aplicação

De acordo com os resultados apresentados na Tabela 6.6 é possível observar que a ausência total de conhecimento sobre os recursos e a ausência de escalonamento dinâmico prejudica muito o desempenho da aplicação, produzindo os piores resultados. Ao contrário, em um ambiente heterogêneo mas dedicado, um escalonador estático com conhecimento sobre os recursos produz excelentes resultados, neste caso servindo como base de comparação para os outros cenários uma vez que não existem custos associados a política de escalonamento dinâmica e o poder computacional dos recursos não varia durante a execução da aplicação. Por sua vez, os resultados apresentados na abordagem (3), que utiliza o conhecimento do escalonamento estático juntamente com o escalonador dinâmico, foram muito bons, bem próximo dos valores conseguidos pela abordagem (2), porém com uma pequena sobrecarga que variou de 0% a 4%, para árvores binárias e de 5% a 12%, para grafos diamantes. É importante destacar que, além da sobrecarga inerente ao escalonamento dinâmico, os tempos também são influenciados pela própria heurística dinâmica. Neste caso, a heurística possui uma visão parcial do grafo (pois, uma visão total pode aumentar muito a sobrecarga) o que pode acarretar decisões de escalonamento eficientes em um dado momento, mas que não necessariamente contribuem para um tempo de execução menor. Este tipo de problema aconteceu principalmente nos grafos diamantes que apresenta um número grande de dependências entre as tarefas. Por último, a estratégia adotada em (4) mostra que um escalonamento estático inicial ruim juntamente com o escalonamento dinâmico prejudica o desempenho da aplicação produzindo resultados piores do que os alcançados em (3). Por outro lado, o os resultados também mostram que o escalonador dinâmico foi capaz de corrigir os erros do escalonamento estático inicial e com isso produziu resultados bem melhores do que a estratégia (1).

É importante notar que quanto menor o tempo de duração da tarefa e quanto maior o número de tarefas, um melhor desempenho pode ser esperado, uma vez que as tarefas são unidades de computação indivisíveis e para um número maior de tarefas o escalonador possui mais opções de reescalonamento a cada passo.

6.2.1.1 Ambiente Dinâmico

Um segundo experimento utilizando as mesmas estratégias do experimento anterior e o mesmo ambiente virtual foi efetuado. A principal diferença é que, além de heterogêneo,

Tabela 6.6: Impacto do Conhecimento no Escalonamento de Tarefas de Aplicações Paralelas com Dependência em um ambiente heterogêneo e dedicado

tamanho da tarefa	aplicação	(1)	(2)	(3)	(4)
$1s$	$OutTree_{511}$	56,46	37,12	37,59	43,02
	$InTree_{511}$	58,61	37,26	37,53	44,55
	Di_{400}	82,67	48,88	51,56	81,50
	$OutTree_{4095}$	397,59	266,81	267,26	275,39
	$InTree_{4095}$	406,44	269,23	266,79	281,86
	Di_{4096}	481,18	315,88	330,29	411,91
$5s$	$OutTree_{511}$	275,60	178,64	183,57	206,93
	$InTree_{511}$	285,32	178,87	183,16	218,68
	Di_{400}	397,88	221,68	248,28	397,24
	$OutTree_{4095}$	1931,87	1281,99	1288,12	1310,21
	$InTree_{4095}$	1940,34	1287,44	1290,88	1343,79
	Di_{4096}	2315,60	1430,41	1558,75	1832,96

o ambiente também é dinâmico, de maneira que o poder computacional dos recursos varia durante a execução da aplicação. Para isto, a partir dos tempos obtidos no experimento anterior para a abordagem (2) são definidos dois instantes de tempo $t1$ e $t2$ para modificar o poder computacional dos recursos durante a execução. O tempo $t1$ indica 1/4 do tempo total de execução obtido e o tempo $t2$ indica 3/4. Inicialmente, todos os recursos do primeiro *site* disponibilizam apenas 50% do seu poder computacional, enquanto que todos os recursos do segundo *site* oferecem 100%, exatamente como no experimento anterior. No instante $t1$, todos os recursos do primeiro *site* passam a oferecer 100% e os recursos do segundo *site* apenas 33%. No instante $t2$, todos os recursos dos dois *sites* passam a oferecer 100% de sua capacidade de processamento. Para este experimento apenas as aplicações com uma quantidade grande de tarefas foram executadas.

A Tabela 6.7 apresenta os resultados obtidos para o cenário dinâmico, utilizando as quatro estratégias de escalonamento do experimento anterior. Como esperado, as duas abordagens estáticas (1) e (2) produziram sempre resultados piores que as abordagens híbridas (3) e (4) (estático + dinâmico). Devido a utilização de um escalonador dinâmico, as abordagens (3) e (4) foram capaz de perceber as variações no ambiente e com isso adaptar a execução da aplicação, produzindo bons resultados.

Dois aspectos interessantes podem ser destacados a partir dos resultados da Tabela 6.7. Em primeiro lugar, em um ambiente dinâmico a importância de um conhecimento prévio sobre o ambiente (escalonamento estático preciso) não é tão relevante quanto em um ambiente heterogêneo e dedicado. Isto acontece pois como o ambiente é dinâmico as decisões inicialmente tomadas pelo escalonador estático podem ter sido completamente erradas. Entretanto, mesmo em um ambiente dinâmico sujeito a variações a abordagem (3) que utiliza o escalonador com conhecimento prévio sobre o ambiente produziu sempre

Tabela 6.7: Impacto do Conhecimento no Escalonamento de Tarefas de Aplicações Paralelas com Dependência em um ambiente dinâmico

tamanho da tarefa	aplicação	(1)	(2)	(3)	(4)
1s	$OutTree_{4095}$	300,23	357,53	268,22	270,02
	$InTree_{4095}$	291,80	357,19	281,33	290,46
	Di_{4096}	419,00	415,52	373,42	375,72
5s	$OutTree_{4095}$	1419,36	1708,44	1262,94	1280,50
	$InTree_{4095}$	1393,37	1710,37	1300,34	1300,46
	Di_{4096}	1892,52	1960,64	1546,09	1581,41

melhores resultados que a abordagem (4) que utiliza um escalonador estático sem conhecimento sobre ambiente. Em segundo lugar, os resultados das abordagens (1) e (2) mostram que em um ambiente dinâmico um escalonador estático sem um conhecimento inicial pode produzir resultados tão bons ou até melhores do que um escalonador estático que conheça o estado inicial dos recursos. Desse modo, as duas principais conclusões que podem ser tiradas a partir deste experimento são que a utilização de apenas escalonadores estáticos nos ambientes dinâmicos não traz muitos benefícios ao desempenho da aplicação e que a abordagem híbrida produz escalonamentos eficientes, até mesmo quando não existe um conhecimento preciso sobre os estados dos recursos no início da aplicação.

6.2.2 Impacto de Estimativas Imprecisas no Escalonamento de Aplicações Paralelas com Dependências

O experimento a seguir avalia o desempenho da aplicação quando as estimativas sobre os pesos das tarefas não forem precisas. Para isso foi utilizado o ambiente heterogêneo e estático do primeiro experimento da subseção anterior, onde as 12 máquinas do primeiro *site* oferecem apenas 50% do seu poder computacional, enquanto que as 12 máquinas do segundo *site* estão dedicadas a execução da aplicação (oferecem 100% do seu poder computacional). Porém, neste experimento, diferentemente da subseção anterior, as aplicações paralelas são compostas de processos heterogêneos, onde as tarefas tem tempo de processamento não uniformes (1/4 dos processos requerem 20s (segundo) de tempo de computação, 1/4 requer 15s, 1/4 requer 10s e 1/4, 5s).

Para simular estimativas imprecisas, a partir dos pesos originais das tarefas (20, 15, 10 ou 5 segundos) foram geradas aleatoriamente estimativas com até 20, 40, 60, 80 e 100% de erro para mais ou para menos. Usando a mesma metodologia aplicada para as aplicações *bag-of-tasks*, para cada uma dessas estimativas, foi executada a aplicação considerando somente o escalonador estático e também a estratégia híbrida. A média do tempos de três execuções das aplicações paralelas analisadas pode ser visto nas Tabelas 6.8 e 6.9.

Tabela 6.8: Evolução do tempo de execução (em segundos) das aplicações de acordo com o aumento no erro na estimativa sobre pesos das tarefas ($OutTree_{511}$, $InTree_{511}$ e Di_{400})

	$OutTree_{511}$		$InTree_{511}$		Di_{400}	
Erro	Est	Hib	Est	Hib	Est	Hib
0%	459,9	463,9	411,5	418,7	635,6	649,3
20%	492,4	473,5	427,7	427,6	669,2	667,8
40%	536,5	468,8	460,6	427,1	722,3	680,7
60%	605,0	487,0	534,0	435,5	814,4	694,4
80%	664,0	501,6	616,2	448,2	951,3	701,1
100%	720,2	493,3	718,5	466,5	1078,4	742,9

Tabela 6.9: Evolução do tempo de execução (em segundos) das aplicações de acordo com o aumento no erro na estimativa sobre pesos das tarefas ($OutTree_{4095}$, $InTree_{4095}$ e Di_{4096})

	$OutTree_{4095}$		$InTree_{4095}$		Di_{4096}	
Erro	Est	Hib	Est	Hib	Est	Hib
0%	3202,6	3208,3	3154,2	3154,5	4653,2	4637,3
20%	3263,5	3244,3	3237,8	3219,0	4809,8	4701,3
40%	3481,2	3220,1	3484,2	3232,3	5219,1	4716,4
60%	3821,0	3210,6	3831,6	3256,9	5726,2	4828,9
80%	4187,3	3245,2	4296,3	3331,9	6306,5	4868,7
100%	4923,4	3344,4	5069,1	3341,2	6823,0	4915,9

À medida que o erro aumenta, o desempenho da aplicação piora, especialmente, quando apenas o escalonar estático é utilizado (colunas **Est** das Tabelas 6.8 e 6.9). Devido aos erros nas estimativas, o escalonador estático não produz um escalonamento eficiente, o que degrada o desempenho da aplicação. Porém, a abordagem híbrida permite que o escalonador dinâmico perceba os erros do escalonador estático melhorando consideravelmente o desempenho da aplicação (colunas **Hib**). Entretanto, é importante ressaltar que o escalonador dinâmico reescalona as tarefas baseado também nas estimativas imprecisas, o que interfere no seu desempenho.

Baseado nos dados da Tabela 6.8, as Figuras 6.4, 6.5 e 6.6 apresentam o desempenho do escalonador estático e da abordagem híbrida a medida que o erro sobre a estimativa dos pesos das tarefas aumenta para as aplicações $OutTree_{511}$, $Intree_{511}$ e Di_{400}, respectivamente. O comportamento é bem similar para as três aplicações, o escalonador estático é ligeiramente melhor que a abordagem híbrida para ambientes sem erro e sem mudanças de carga (devido a pequena sobrecarga do escalonador dinâmico), porém seu desempenho degrada drasticamente a medida que o erro aumenta. Por outro lado, a utilização da abordagem híbrida permite que o escalonador dinâmico perceba o fraco desempenho do escalonamento estático, modificando-o durante a execução da aplicação conseguindo

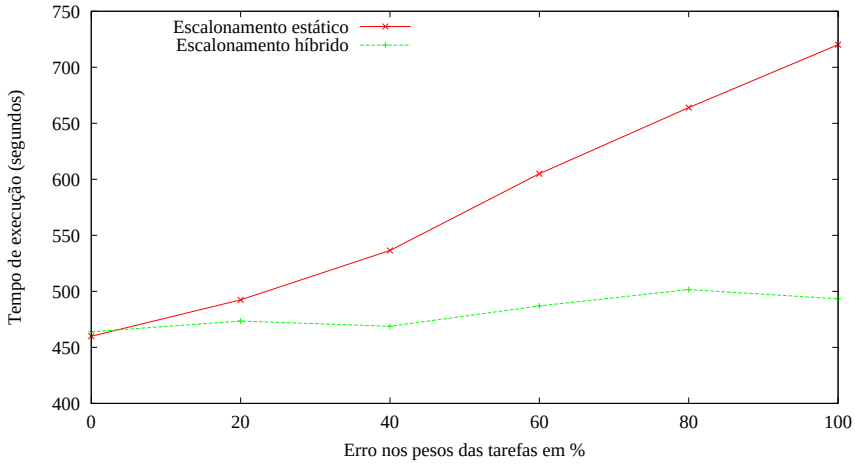

Figura 6.4: Evolução do tempo de execução das aplicações de acordo com o aumento no erro na estimativa dos pesos (Aplicação $OutTree_{511}$)

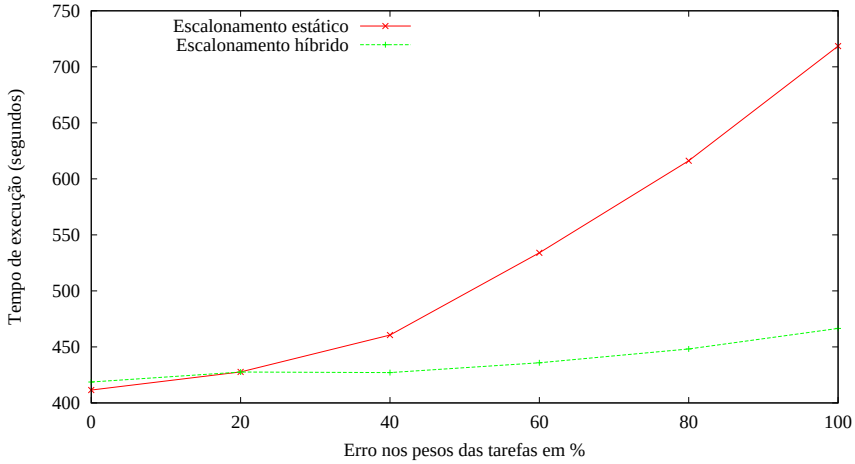

Figura 6.5: Evolução do tempo de execução das aplicações de acordo com o aumento no erro na estimativa dos pesos (Aplicação $InTree_{511}$)

melhorar bastante o seu desempenho. A medida que o erro aumenta o tempo de execução piora, mas mesmo quando a estimativa de erro é de até 100%, os tempos de execução são apenas 6,3%, 11,4% e 14,4% piores do que os tempos com 0% de erro nas estimativas para $OutTree_{511}$, $Intree_{511}$ e Di_{400}, respectivamente. O erro é um pouco maior para a aplicação diamante (Di_{400}), onde o número de dependências entre as tarefas é maior, de modo que poucas tarefas paralelas ficam disponíveis para que o escalonador dinâmico corrigir os erros do escalonamento estático. Ao contrário, devido à topologia das árvores binárias, um número maior de tarefas independentes pode ser detectado a cada evento de escalonamento.

As Figuras 6.7, 6.8 e 6.9 apresentam o desempenho do escalonador estático e da abordagem híbrida a medida que o erro sobre a estimativa dos pesos das tarefas aumenta para as aplicações $OutTree_{4095}$, $Intree_{4095}$ e Di_{4096}, respectivamente. Apesar do número

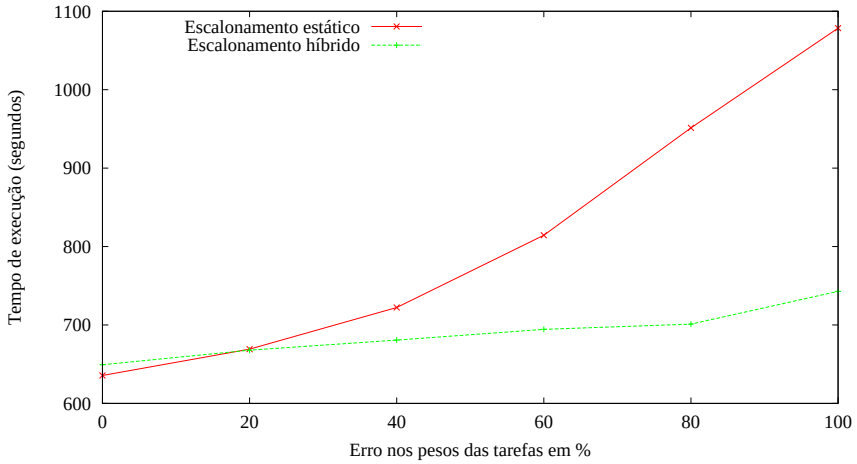

Figura 6.6: Evolução do tempo de xecução das aplicações de acordo com o aumento no erro na estimativa dos pesos (Aplicação Di_{511})

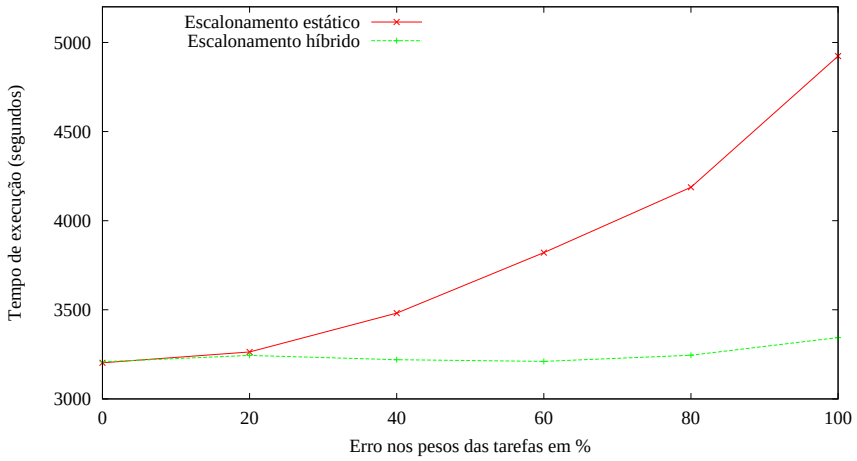

Figura 6.7: Evolução do tempo de execução das aplicações de acordo com o aumento no erro na estimativa dos pesos (Aplicação $OutTree_{4095}$)

bem maior de tarefas, os resultados são bastante similares aos obtidos com as mesmas aplicações com um número menor de tarefas (Tabela 6.8). Entretanto, a quantidade maior de tarefas destas aplicações reflete no número de tarefas paralelas disponíveis a cada evento de escalonamento para que o escalonador dinâmico avalie, e melhore o desempenho da aplicação. Analisando o experimento quando as estimativas tem até 100% de erro em relação ao peso real das tarefas, a piora no tempo de execução foi de apenas 4,2%, 5,9% e 6,0% em relação aos tempos sem erro nas estimativas para as aplicações $OutTree_{4095}$, $Intree_{4095}$ e Di_{4096}, respectivamente.

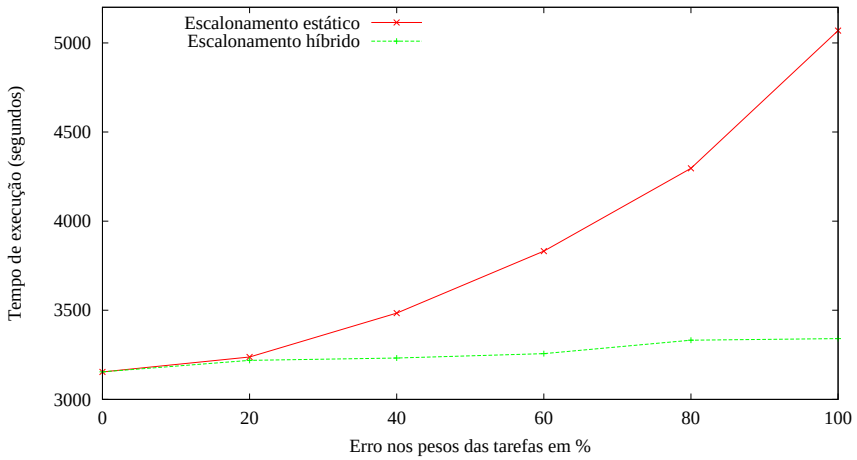

Figura 6.8: Evolução do tempo de execução das aplicações de acordo com o aumento no erro na estimativa dos pesos (Aplicação $InTree_{4095}$)

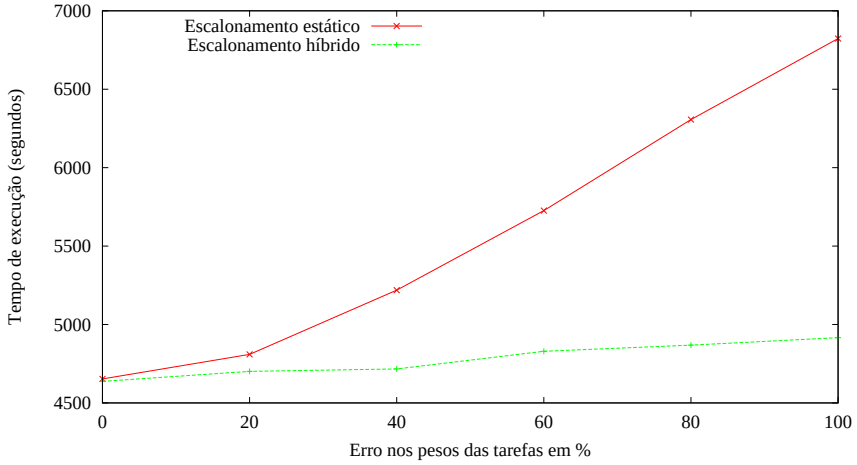

Figura 6.9: Evolução do tempo de execução das aplicações de acordo com o aumento no erro na estimativa dos pesos (Aplicação Di_{4096})

6.3 Resumo

Informações como o peso das tarefas de uma aplicação ou o custo de transmissão de uma mensagem, assim como o poder computacional dos recursos disponíveis no ambiente de execução nem sempre são fáceis de serem obtidas com precisão. Ainda mais, em ambientes heterogêneos, compartilhados e dinâmicos, como as grades computacionais. Logo, esse capítulo avaliou a sensibilidade da estratégia proposta a (falta de) precisão dos modelos da aplicação e da arquitetura.

Primeiramente foi constatado que fatores como a ordenação das tarefas e a forma da distribuição das tarefas entre os processadores influenciam no desempenho das aplicações *bag-of-tasks*. Em seguida, a análise indica que a ausência de informações sobre o ambiente e sobre a aplicação prejudica o desempenho. Entretanto, a estratégia híbrida adotada

neste trabalho minimiza essa perda de desempenho quando estimativas precisas sobre a aplicação e o ambiente não são coletadas, através da utilização do escalonador dinâmico que, ao perceber as mudanças no ambiente e os erros do escalonamento estático, reescalona as tarefas da aplicação proporcionando uma execução eficiente.

A última parte deste capítulo apresenta os benefícios da estratégia híbrida para aplicações paralelas composta de tarefas com relações de precedência. Em especial, é mostrado que, assim como nas aplicações *bag-of-tasks*, a falta de informação sobre o ambiente e sobre a aplicação prejudica bastante o desempenho da aplicação. No entanto, assim como para aplicações *bag-of-tasks*, a estratégia híbrida de escalonamento para aplicações com dependências consegue suprir o erro de informação através de um escalonamento dinâmico eficaz, produzindo bons resultados. Desse modo, mesmo com a ausência de informação (ou informações imprecisas) sobre o ambiente grade e a aplicação, o *EasyGrid AMS* executa a aplicação eficientemente, proporcionando excelentes resultados.

A principal conclusão deste capítulo é que a estratégia proposta ao longo deste trabalho de tese é pouco sensível aos erros das estimativas fornecidas pelos modelos da aplicação e da arquitetura. Ou seja, mesmo que as informações desses modelos estejam muito erradas (imprecisas) o desempenho da aplicação é pouco afetado, mantendo o bom desempenho. No cerne de tudo o que foi apresentado encontra-se o modelo alternativo de execução 1PTask que, de acordo com os resultados apresentados ao longo deste trabalho de tese, possibilita a execução eficiente de aplicações MPI nas grades computacionais.

Capítulo 7

Conclusões e Trabalhos Futuros

As grades computacionais fornecem um enorme poder computacional a um baixo custo e por formarem um ambiente dinâmico, compartilhado, heterogêneo e descentralizado, o acesso aos seus recursos é normalmente feito através de sistemas gerenciadores de recursos ou aplicações que servem de interface entre o usuário e a infra-estrutura. Pesquisadores vem mostrando que programas compostos por tarefas independentes podem ser executados eficientemente nesses ambientes. Porém, o uso desse ambiente para execução de programas paralelos ainda é um desafio.

Atualmente, a biblioteca MPI é uma das ferramentas mais utilizadas para a implementação de programas paralelos que se comunicam através de trocas de mensagens [49, 61]. Porém, esses programas normalmente foram projetados para serem executados em ambientes dedicados, homogêneos e estáveis. Desse modo, o modelo padrão adotado pelos programadores para suas aplicações MPI é a execução de apenas um processo por recurso durante toda duração do programa, onde todos os processos são criados estaticamente quando o programa MPI é iniciado.

O objetivo central deste trabalho é a execução eficiente de aplicações MPI nas grades computacionais. Para atingir tal objetivo, esta tese propõe a adoção de um modelo alternativo de execução, "um processo por tarefa", ao invés do tradicional "um processo por recurso". Sua principal característica é ser independente da arquitetura e do ambiente disponível e com isso, permitir uma utilização eficaz dos recursos da grade.

O modelo alternativo proposto é validado utilizando o *middleware EasyGrid*, que adota o modelo de execução proposto nesta tese, através de duas classes de aplicações distintas: aplicações compostas por tarefas independentes e aplicações paralelas com dependências. Através de experimentos em ambientes grades reais, a execução de aplicações paralelas (com e sem relações de precedências) utilizando o modelo de execução alternativo proporcionou uma melhor adaptação da aplicação às condições do ambiente, produzindo

bons resultados.

Mais especificamente, com relação as aplicações paralelas com dependências, este trabalho apresenta a viabilidade da execução de aplicações fortemente acopladas através do modelo alternativo proposto nas grades computacionais. Inicialmente, foi apresentado um estudo sobre o algoritmo *ring* para a aplicação N-corpos e suas limitações para os ambientes heterogêneos, dinâmicos e compartilhados. Em seguida, foi proposta uma adaptação para esse algoritmo onde, apesar de mudanças superficiais no código, a principal diferença em relação ao algoritmo anterior é conceitual. Em outras palavras, o novo algoritmo se baseia no modelo alternativo para conseguir um melhor desempenho nos ambientes grades. Além disso, uma análise de desempenho do novo algoritmo mostra que este varia em função da largura W utilizada e do escalonamento de suas tarefas. Baseado nessa análise, foi apresentada uma estratégia para se conseguir uma execução eficaz do novo algoritmo que não é especifica para uma determinada granularidade. Ou seja, o algoritmo fornece um valor de W, valor este que ao ser calculado ajusta a granularidade da aplicação de maneira a conseguir maximizar seu desempenho.

Os resultados dos experimentos, comparando as aplicações executadas sob o modelo tradicional 1PProc e sob o modelo alternativo 1PTask, nos ambientes grades indicaram claramente a vantagem da estratégia proposta. Por exemplo, a execução de uma aplicação fortemente acoplada em ambientes grades utilizando o modelo proposto nesta tese foi 4,3 vezes melhor do que a abordagem tradicional.

Finalizando, uma vez que pode ser difícil obter informações precisas sobre as tarefas de uma aplicação parela e sobre os recursos disponíveis para execução da aplicação, este trabalho de tese avaliou a sensibilidade da estratégia proposta a (falta de) precisão sobre as informações fornecidas pelos modelos da aplicação e da arquitetura. Através de experimentos, foi concluído que mesmo que as estimativas sobre o tempo de execução das tarefas e sobre o poder computacional dos recursos conseguidas estaticamente sejam imprecisas, o escalonador dinâmico consegue adaptar a execução da aplicação, produzindo bons resultados. Mais do que isso, os experimentos mostraram que mesmo que as informações sobre as tarefas da aplicação estejam até 100% imprecisas (erradas), o escalonador dinâmico é capaz de adaptar a execução da aplicação, conseguindo tempos de execução no máximo 14% pior do que os tempos obtidos com as informações corretas. Isso comprova a pouca sensibilidade da estratégia proposta aos possíveis erros nas estimativas sobre a aplicação e o ambiente de execução, de maneira que quanto mais precisa forem as informações melhores resultados serão alcançados. Por outro lado, o aumento na falta de precisão sobre a informação é compensado pela estratégia adotada, prejudicando pouco o desempenho da aplicação.

7.1 Trabalhos Futuros

A solução apresentada nesta tese para aplicação N-corpos *ring* é do tipo **evolutiva**, conforme a definição apresentada no Capítulo 1. Uma proposta de continuação do estudo apresentado nesta tese sobre a aplicação N-corpos é torná-la também uma aplicação **maleável**, sem a necessidade de um sistema gerenciador de recursos específico e sem que o programador tenha que codificar em seu programa paralelo o mecanismo para permitir a maleabilidade. Nos experimentos apresentados neste trabalho, após o início da aplicação, o valor de W permanece fixo. Porém, em ambientes dinâmicos e compartilhados como as grades, onde recursos podem entrar e sair a qualquer momento e processos de usuários locais podem ter prioridade maior do que tarefas remotas, o valor de W pode necessitar ser ajustado durante a execução da aplicação, visto que esta poderá executar por um longo tempo (milhares de *time steps*). Desse modo, o *EasyGrid AMS*, a partir dos dados fornecidos pela camada de monitoramento, iria recalcular o novo valor de W a cada *time step* utilizando a estratégia apresentada na Subseção 5.2.1 (para a aplicação N-corpos) e, caso necessário, adaptaria a aplicação para um novo valor de W e consequentemente, uma melhor utilização dos recursos disponíveis. Ou seja, a aplicação N-corpos *ring* além de evolutiva (toma decisões própria para alterar a alocação de processos) se tornaria também maleável (capaz de alterar a granularidade de seus processos durante a execução).

Para uma melhor eficiência do modelo alternativo apresentado, é interessante procurar diminuir a sobrecarga do sistema gerenciador da aplicação, no caso, o *EasyGrid AMS*. Atualmente, otimizações nas estruturas de dados interna do *middleware* já estão sendo realizadas, assim como um controle no envio de mensagens.

Todos os modelos apresentados nesta tese se referem a execução de uma aplicação paralela em um ambiente grade. Assumindo que várias aplicações gerenciadas através do *EasyGrid AMS* podem executar ao mesmo tempo em um ambiente grade compartilhando recursos, um trabalho interessante seria conseguir que essas aplicações utilizassem os recursos disponíveis da melhor maneira possível. Uma solução poderia ser a criação de um escalonador centralizado que controlasse todas as aplicações. Porém, para uma maior escalabilidade, esse controle deveria ser distribuído, de maneira que cada sistema gerenciador conseguisse se comunicar com os outros e através de algum mecanismo autonômico (ou protocolo pré-definido) as aplicações se ajustassem aos recursos disponíveis otimizando seu desempenho. Testes iniciais já foram feitos e mostraram que as aplicações, executando através do *EasyGrid AMS*, conseguem se adaptar as políticas existentes e ao compartilhamento de recursos [18, 118] alcançando ótimos resultados. O desafio agora é ver como se comporta a grade como um todo, ou seja, se a utilização dos recursos está sendo feita da

melhor maneira possível ou pode ser melhorada.

Com relação ao modelo da aplicação adotado neste trabalho de tese, GAD, um trabalho interessante é a criação de uma ferramenta para gerar os GADs que representam as aplicações paralelas a serem executadas. Como foi descrito na Seção 2.3, várias soluções alternativas já foram propostas, porém ainda não se conhece uma ferramenta que a partir do programa paralelo gere uma representação através de GAD, de maneira que todas as relações de precedências entre as tarefas sejam definidas, assim como os pesos das tarefas e os custos das mensagens. O modelo alternativo proposto nesta tese facilita a geração das relações de precedência entre as tarefas, uma vez que as tarefas são unidades básicas que recebem dados, executam e enviam dados e, mais importante, é criado um processo para cada tarefa. A princípio, a complexidade maior reside na geração dos pesos das tarefas e dos custos das mensagens. Seria necessária uma análise dos índices utilizados para medir a capacidade de processamento dos recursos e a partir do índice escolhido, conseguir uma maneira efetiva para gerar os pesos das tarefas e custos das mensagens. É importante que os pesos/custos gerados sejam independentes do recurso em que a tarefa for ser executada ou a mensagem for ser transmitida.

Um outro possível estudo seria a adoção de um modelo da aplicação que permitisse a representação de ciclos, de maneira que mais aplicações paralelas pudessem ser representadas. Por exemplo, o modelo GAD poderia ser estendido para permitir a co-dependência entre tarefas. Desse modo, esse novo modelo teria todas as características de um GAD, além de uma nova característica chamada de co-dependência. Duas ou mais tarefas co-dependentes tem que ser executadas simultaneamente, pois elas trocam dados entre si (por exemplo, fazem parte de um laço). É importante destacar que a maior parte das tarefas nessa nova proposta constituem um GAD, uma pequena parte que se comunica entre si possui a co-dependência. Porém, apesar da pequena modificação na representação dos GADs, essa nova característica acrescenta uma complexidade maior ao problema de escalonamento. A princípio é necessário uma adaptação nas heurísticas existentes para GADs, de modo que elas possam também ser usadas para o escalonamento de grafos com co-dependência. Além disso, o *EasyGrid AMS* também teria que ser adaptado para tratar tarefas co-dependentes.

APÊNDICE A – Configurações dos Recursos Utilizados nos Experimentos

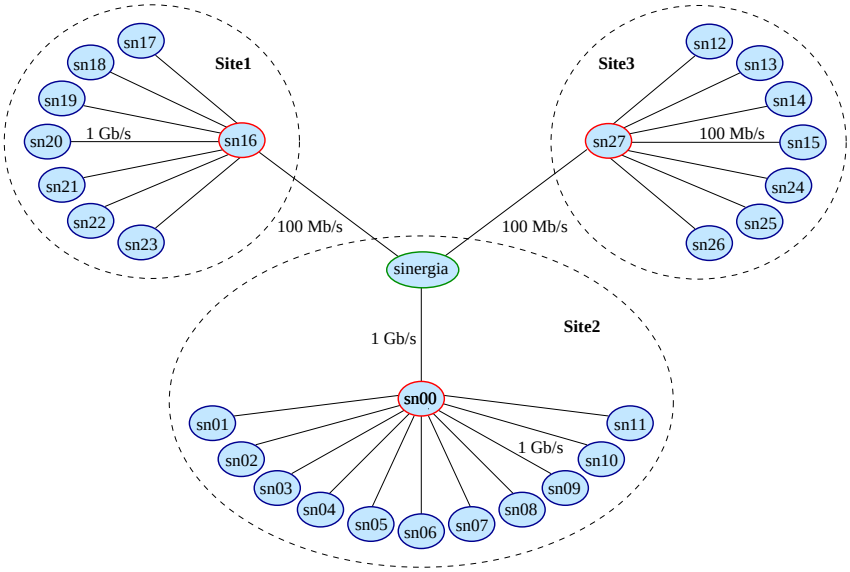

Figura A.1: Grade sinergia composta de 3 sites distintos

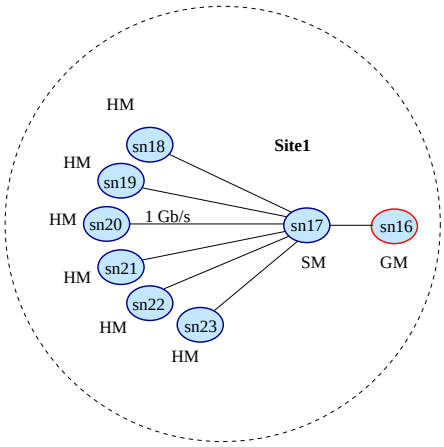

Figura A.2: Configuração utilizando 8 recursos

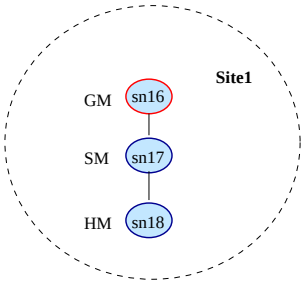

Figura A.3: Configuração utilizando apenas 3 recursos

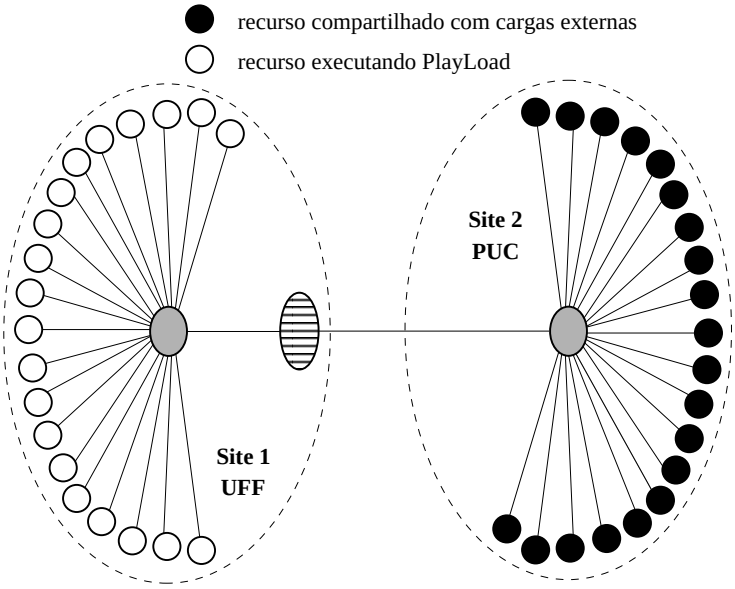

Figura A.4: Configuração disponível para MPI ring e AMS ring com $W = 60$ para 200.000 partículas em um ambiente grade composto de 2 *sites*.

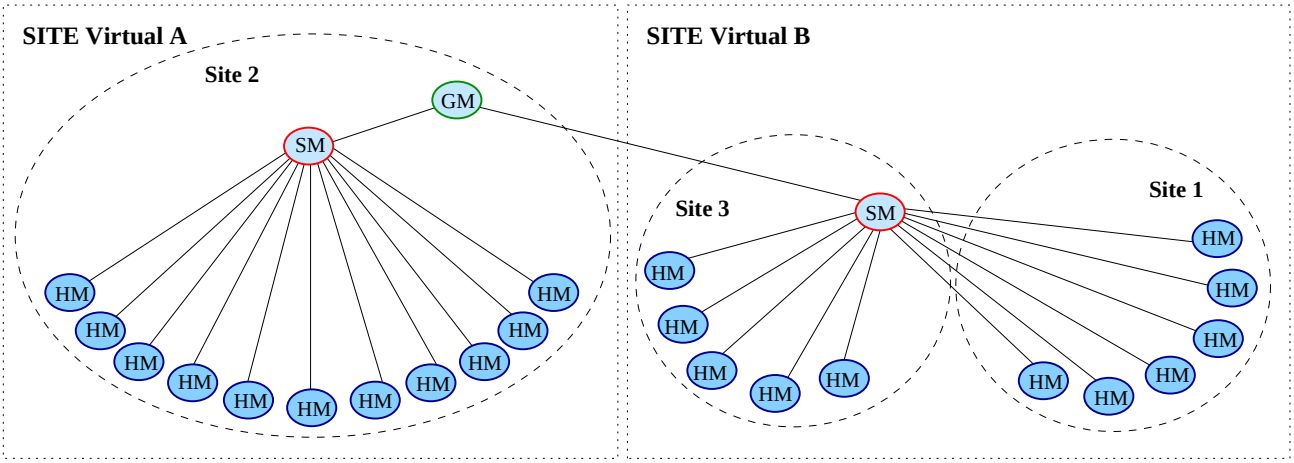

Figura A.5: Configuração virtual da grade para avaliação do escalonamento híbrido

Referências

[1] Aarseth, S. J. From NBODY1 to NBODY6: The growth of an industry. *Publications of the Astronomical Society of The Pacific 111* (Novembro 1999), 1333–1346.

[2] Abramson, D., Giddy, J., and Kotler, L. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)* (Cancun, Mexico, Maio 2000), IEEE Computer Society, pp. 520–528.

[3] Ahmad, I., Kwok, Y.-K., Wu, M.-Y., and Shu, W. CASCH: A tool for computer-aided scheduling. *IEEE Concurrency 8*, 4 (2000), 21–33.

[4] Al-Tawil, K., and Moritz, C. A. LogGP performance evaluation of MPI. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing* (Washington, D.C., EUA, 1998), IEEE Computer Society, pp. 366–373.

[5] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing 44*, 1 (1997), 71–79.

[6] Ali, S., Siegel, H. J., Maheswaran, M., Ali, S., and Hensgen, D. Task execution time modeling for heterogeneous computing systems. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop* (Cancun, Mexico, 2000), IEEE Computer Society, pp. 185–199.

[7] Alves, C. E. R., Caceres, E. N., Dehne, F., and Song, S. W. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proc. 14th Symposium on Parallel Algorithms and Architectures (ACM-SPAA)* (2002), ACM Press, pp. 275–281.

[8] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. Seti@home: an experiment in public-resource computing. *Communications of the ACM 45*, 11 (2002), 56–61.

[9] Anderson, T. E., Culler, D. E., and Patterson, D. A. A case for NOW (networks of workstations). *IEEE Micro 15*, 1 (1995), 54–64.

[10] Barnes, J., and Hut, P. A hierarchical O(N log N) force-calculation algorithm. *Nature 324* (1986), 446–449.

[11] Beaumont, O., Legrand, A., and Robert, Y. Static scheduling strategies for heterogeneous systems. *Computing and Informatics 21* (2002), 413–430.

[12] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed,

D., Torczon, L., and Wolski, R. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications 15*, 4 (2001), 327–344.

[13] Berman, F., Fox, G., and Hey, T. *Grid Computing.* John Wiley and Sons Ltd, 2003.

[14] Berman, F., and Wolski, R. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium* (Berlin, Germany, Maio 1997), Springer-Verlag.

[15] Boeres, C., Fonseca, A. A., Mendes, H. A., Menezes, L. T., Moura, N. T., Silva, J. A., Vianna, B. A., and Rebello, V. E. F. An EasyGrid Portal for scheduling system-aware applications on computational grids. *Concurrency and Computation: Practice and Experience 18*, 6 (2005), 553–566.

[16] Boeres, C., Lima, A., and Rebello, V. Hybrid task scheduling: Integrating static and dynamic heuristics. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)* (São Paulo, Brasil, Novembro 2003), IEEE Computer Society, pp. 199–206.

[17] Boeres, C., Nascimento, A., and Rebello, V. Cluster-based task scheduling for LogP model. *International Journal of Foundations of Computer Science 10*, 4 (1999), 405–424.

[18] Boeres, C., Nascimento, A. P., Rebello, V. E. F., and Sena, A. C. Efficient hierarchical self-scheduling for MPI applications executing in computational grids. In *Proceedings of the 3rd International workshop on Middleware for Grid Computing* (Grenoble, France, Novembro 2005), ACM Press, pp. 1–6.

[19] Boeres, C., and Rebello, V. E. F. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience 16*, 5 (Abril 2004), 425–432.

[20] Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lantéri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.-G., and Iréa, T. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications 20*, 4 (Novembro 2006), 481–494.

[21] Branco, K., and Santana, M. A novel simulator for evaluating performance indices on heterogeneous distributed systems environments. *International Symposium on Industrial Electronics 4* (Julho 2006), 3316–3321.

[22] Branco, K., Santana, M., Santana, R., and Bruschi, S. PIV and WPIV: Performance index for heterogeneous systems evaluation. *International Symposium on Industrial Electronics 1* (Julho 2006), 323–328.

[23] Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing 61*, 6 (2001), 810–837.

[24] Brooks, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*, 1 (Março 1986), 14–23.

[25] Buyya, R., Ed. *High Performance Cluster Computing: Architectures and Systems*. Vol. 1. Prentice Hall, EUA, 1999.

[26] Buyya, R., Abramson, D., and Giddy, J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region (HPC ASIA'2000)* (Beijing, China, Maio 2000), vol. 1, IEEE Computer Society, pp. 283–289.

[27] Buyya, R., Abramson, D., and Venugopal, S. The grid economy. *Proceedings of the IEEE, Special Issue on Grid Computing 93*, 3 (Março 2005), 698–714.

[28] Cao, J., Jarvis, S., Saini, S., and Nudd, G. Gridflow: Workflow management for grid computing. In *Proceedings of 3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003)* (Tóquio, Japão, Maio 2003), IEEE Computer Society, pp. 198–205.

[29] Casanova, H. Network modeling issues for grid application scheduling. *International Journal of Foundation on Computer Science 16*, 2 (2005), 145–162.

[30] Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings 9th Heterogeneous Computing Workshop (HCW '00)* (Cancun, Mexico, Maio 2000), IEEE Computer Society, pp. 349–363.

[31] Castain, R. H., Woodall, T. S., Daniel, D. J., Squyres, J. M., Barrett, B., and Fagg, G. E. The open run-time environment (openrte): A transparent multicluster environment for high-performance computing. *Future Generation Computer Systems 24*, 2 (2008), 153–157.

[32] Cera, M. C., Pezzi, G. P., Mathias, E. N., Maillard, N., and Navaux, P. O. A. Improving the dynamic creation of processes in MPI-2. In *Proceedings of the 13th European PVM/MPI User's Group Meeting* (2006), vol. 4192 of *Lecture Notes in Computer Science*, Springer, pp. 247–255.

[33] Cesar, E., Mesa, J., Sorribes, J., and Luque, E. Modeling master-worker applications in poetries. In *Proceedings ot the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (2004), pp. 22–30.

[34] Chao, L.-F., and Sha, E. H.-M. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems 8*, 12 (1997), 1259–1267.

[35] Chao, L.-F., and Sha, E.-M. Efficient retiming and unfolding. *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on 1* (Apr 1993), 421–424.

[36] Chapin, S. J. Distributed and multiprocessor scheduling. *ACM Comput. Surv. 28*, 1 (1996), 233–235.

[37] Cirne, W., Paranhos, D., Costa, L.and Santos-Neto, E., Brasileiro, F., Sauve, J., Silva, F., Barros, C., and Silveira, C. Running bag-of-tasks applications on computational grids: The MyGrid approach. In *Proceedings of the 32nd International Conference on Parallel Processing (ICPP03)* (Kaohsiung, Taiwan, 2003), IEEE Computer Society, pp. 407–416.

[38] Coffman, E. G., Garey, M. R., and Johnson, D. S. An application of bin-packing to multiprocessor scheduling. *SIAM Jounal on Computing 7*, 1 (1978), 1–17.

[39] Cooper, K., Dasgupta, A., Kennedy, K., Koelbel, C., Mandal, A., Marin, G., Mazina, M., Mellor-Crummey, J., Berman, F., Casanova, H., Chien, A., Dail, H., Liu, X., Olugbile, A., Sievert, O., Xia, H., Johnsson, L., Liu, B., Patel, M., Reed, D., Deng, W., Mendes, C., Shi, Z., YarKhan, A., and Dongarra, J. New grid scheduling and rescheduling methods in the GrADS project. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)* (New Mexico, EUA, Abril 2004), vol. 1, IEEE Computer Society, pp. 199–206.

[40] Cosnard, M., and Trystram, D. *Parallel Algorithms and Architectures.* PWS Publishing Co., Boston, EUA, 1995.

[41] Costa, L. B., Feitosa, L., Araújo, Mendes, G. W. D., Coelho, R., Cirne, W., and Fireman, D. MyGrid: A complete solution for running bag-of-tasks applications. In *Salão de Ferramentas do Simpósio Brasileiro de Redes de Computadores* (Gramado, Brasil, 2004).

[42] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken., T. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP'93)* (San Diego, CA, EUA, Julho 1993), ACM Press, pp. 1–12.

[43] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken., T. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP'93)* (San Diego, EUA, Julho 1993), ACM Press, pp. 1–12.

[44] da Silva, D. P., Cirne, W., and Brasileiro, F. V. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proceedings of the 9th International Euro-Par Conference (Euro-Par '03)* (2003), pp. 169–180.

[45] da Silva, J. A., and Rebello, V. E. F. Low cost self-healing in MPI applications. In *Proceedings of the 14th European PVM/MPI User's Group Meeting, Paris, France* (Outubro 2007), Springer, pp. 144–152.

[46] de Araújo, A. P. F. *Paralelização Autonômica de Metaheurísticas em Ambientes de Grid.* PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, RJ, Brasil, Março 2008.

[47] Dinda, P. A., and O'Hallaron, D. R. Realistic CPU workloads through host load trace playback. In *LCR '00: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (Londres, Reino Unido, 2000), Springer-Verlag, pp. 246–259.

[48] Dogan, A., and Özgüner, F. LDBS: A duplication based scheduling algorithm for heterogeneous computing systems. In *Proceedings of the International Conference on Parallel Processing (ICPP'02)* (Vancouver, Canada, Agosto 2002), IEEE Computer Society, pp. 352–359.

[49] Dongarra, J., Foster, I., Fox, G., Gropp, W., kennedy, K., torczon, L., and White, A. *Sourcebook of Parallel Computing.* Morgan Kaufmann Publishers Inc., 2003.

[50] Dorband, E. N., Hemsendorf, M., and Merritt, D. Systolic and hypersystolic algorithms for the gravitational N-body problem, with an application to brownian motion. *Journal of Computational Physics 185* (2003), 484.

[51] EGEE - Enabling Grid for E-sciencE. `http://www.eu-egee.org/`, Último acesso 10/10/2008.

[52] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (2002), 375–408.

[53] Ernemann, C., Hamscher, V., Schwiegelshohn, U., Yahyapour, R., and Streit, A. On advantages of grid computing for parallel job scheduling. *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid* (Maio 2002), 39–39.

[54] Fagg, G. E., and Dongarra, J. Ft-mpi: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Londres, Reino Unido, 2000), Springer-Verlag, pp. 346–353.

[55] Feitelson, D. G., and Rudolph, L. Towards convergence in job schedulers for parallel supercomputers. In *Proceedings of the JSSPP* (1996), D. G. Feitelson and L. Rudolph, Eds., Lecture Notes in Computer Science, Springer, pp. 1–26.

[56] Fitzgerald, S. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC'01)* (Washington, D.C., EUA, 2001), IEEE Computer Society, p. 181.

[57] Fortune, S., and Wyllie, J. Parallelism in Random Access Machines. In *In Proceedings of the 10th ACM Symposium on Theory of Computing* (1978), pp. 114–118.

[58] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, Julho 1997. `http://www.mpi-forum.org/`, Último acesso 10/10/2008.

[59] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, junho 1995. `http://www.mpi-forum.org/`, Último acesso 10/10/2008.

[60] Fosdick, L. D., Jessup, E. R., Schauble, C. J. C., and Domik, G. *An introduction to high-performance scientific computing.* MIT Press, Cambridge, EUA, 1996.

[61] Foster, I., Ed. *Designing and Building Parallel Programs.* An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation, 1995.

[62] Foster, I. The grid: A new infrastructure for 21st century science. *Physics Today 55(22):42*, 8 (2002).

[63] Foster, I., and Kesselman, C. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (Summer 1997), 115–128.

[64] Foster, I., and Kesselman, C., Eds. *The GRID: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1999.

[65] Foster, I., and Kesselman, C., Eds. *The GRID 2: Blueprint for a New Computing Infrastructure.* 2ª edição. Morgan Kaufmann, 2004.

[66] Foster, I., Vockler, J., Wilde, M., and Zhao, Y. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Proceedings of 14th International Conference on Scientific and Statistical Database Management.* (2002), pp. 37–46.

[67] Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. Condor-G: A computational management agent for multi-institutional grids. *Cluster Computing 3*, 5 (2002), 237–246.

[68] friesen, D., and langston, M. Bounds for multifit scheduling on uniform processors. *SIAM Journal on Computing 12*, 1 (1983), 60–70.

[69] Friesen, D. K. Tighter bounds for LPT scheduling on uniform processors. *SIAM Journal on Computing 16*, 3 (1987), 554–560.

[70] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapeste, Hungria, Setembro 2004), pp. 97–104.

[71] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., Nova Iorque, EUA, 1979.

[72] Giersz, M., and Heggie, D. C. Statistics of N-body simulations I. Equal masses before core collapse. *Royal Astronomical Society MONTHLY NOTICES 268* (1994), 257–275.

[73] Graham, R. L. Bounds on multiprocessing time anomalies. *Journal of the ACM 17*, 2 (1969), 416–429.

[74] Graham, R. L., Choi, S.-E., Daniel, D. J., Desai, N. N., Minnich, R. G., Rasmussen, C. E., Risinger, L. D., and Sukalski, M. W. A network-failure-tolerant message-passing system for terascale clusters. *Int. J. Parallel Program. 31*, 4 (2003), 285–303.

[75] Graham, R. L., Shipman, G. M., Barrett, B. W., Castain, R. H., Bosilca, G., and Lumsdaine, A. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks* (Barcelona, Spain, Setembro 2006).

[76] Grand Challenges for Engineering. `http://www.engineeringchallenges.org/`, Último acesso 10/10/2008.

[77] Gray, J. What next?: A dozen information-technology research goals. *Communications of the ACM 50*, 1 (2003), 41–57.

[78] Greengard, L. *The Rapid Evaluation of Potential Fields in Particle Systems.* The MIT Press, 1988.

[79] Gregoretti, F., Laccetti, G., Murli, A., Oliva, G., and Scafuri, U. Mgf: A grid-enabled MPI library. *Future Generation Computer Systems 24*, 2 (2008), 158–165.

[80] Grid'5000. `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`, Último acesso 10/10/2008.

[81] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI second edition.* The MIT Press, 1999.

[82] Gualandris, A., Zwart, S. P., and Tirado-Ramos, A. Performance analysis of direct N-body algorithms for astrophysical simulations on distributed systems. *Parallel Computing 33*, 3 (2007), 159–173.

[83] Hagerup, T. Allocating independent tasks to parallel processors: an experimental study. *Journal of Parallel Distributed Computing 47*, 2 (1997), 185–197.

[84] Han, J.-J., and Li, Q.-H. A novel static task scheduling algorithm in distributed computing environments. *ipdps 1* (2004), 3a.

[85] Hoekstra, A. G., Zwart, S. P., Bubak, M., and Sloot, P. M. *Towards Distributed Petascale Computing.* Chapman and Hall/CRC Computational Science Series, 2007.

[86] Hong, B., and Prasanna, V. Adaptive allocation of independent tasks to maximize throughput. *IEEE Transactions on Parallel and Distributed Systems 18*, 10 (2007), 1420–1435.

[87] Horowitz, E., and Sahni, S. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM 23*, 2 (1976), 317–327.

[88] Huang, C., Lawlor, O., and Kale, L. V. Adaptive MPI. In *Proc. 16th International Workshop on Languages and Compilers for Parallel Computing* (2003), Springer, pp. 306–322.

[89] Huedo, E., Montero, R. S., and Llorente, I. M. The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing: Practice and Experience 6*, 3 (Setembro 2005), 1–8.

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

[91] Ino, F., Fujimoto, N., and Hagihara, K. LogGPS: a parallel computational model for synchronization analysis. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming (PPoPP)* (Nova Iorque, EUA, 2001), ACM Press, pp. 133–142.

[92] Kale, L. V., and Krishnan, S. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In *Proc. Conference on Object Oriented Programming Systems, Languages and Applications* (Setembro 1993), A. Paepcke, Ed., ACM Press, pp. 91–108.

[93] Kalinowski, T., Kort, I., and Trystram, D. List scheduling of general task graphs under logp. *Parallel Computing 26*, 9 (2000), 1109–1128.

[94] Karniadakis, G., and II, R. K. *Parallel Scientific Computing in C++ and MPI.* Cambridge University Press, 2003.

[95] Kenyon, C. Best-fit bin-packing with random order. In *In SODA: ACM-SIAM Symposium on Discrete Algorithms* (1996), pp. 359–364.

[96] Kim, S. J., and Browne, J. C. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proceedings of the 3rd International Conference on Parallel Processing* (1988), vol. 3, pp. 1–8.

[97] Krauter, K., Buyya, R., and Maheswaran, M. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience 32*, 2 (2002), 135–164.

[98] Kung, H. Why systolic architectures? *Computer 15*, 1 (1982), 37–46.

[99] Kwok, Y.-K., and Ahmad, I. Dynamic critical-path scheduling: An effective technique for allocating tasks graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 7*, 5 (Maio 1996), 506–521.

[100] Kwok, Y.-K., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys 31*, 4 (Dezembro 1999), 406–471.

[101] LAM/MPI Parallel Computing. `http://www.lam-mpi.org/`, Último acesso 10/10/2008.

[102] Lastovetsky, A., and Reddy, R. Heterompi: towards a message-passing library for heterogeneous networks of computers. *J. Parallel Distrib. Comput. 66*, 2 (2006), 197–220.

[103] Large Hadron Collider. `http://lhc.web.cern.ch/lhc/`, Último acesso 10/10/2008.

[104] Lo, V. M. Temporal communication graphs: Lamport's process-time graphs augmented for the purpose of mapping and scheduling. *Journal of Parallel and Distributed Computing 16* (1992), 378–384.

[105] Lo, V. M., Rajopadhye, S., Gupta, S., Keldsen, D., Mohamed, M. A., Nitzberg, B., Telle, J. A., and Zhong, X. Oregami: Tools for mapping parallel computations to parallel architectures. *International Journal of Parallel Programming 20* (1991), 237 – 270.

[106] Maghraoui, K., Desell, T., Szymanski, B., and Varela, C. Dynamic malleability in iterative MPI applications. In *Proceedings of 7th International Symposium on Cluster Computing and the Grid (CCGrid 2007)* (Rio de janeiro, Brasil, Maio 2007), IEEE Computer Society, pp. 591–598.

[107] Maheswaran, M., and Siegel, H. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of the 7th Heterogeneous Computing Workshop (HCW98)* (Orlando, Florida, EUA, Março 1998), IEEE Computer Society, pp. 57–69.

[108] Makino, J. An efficient parallel algorithm for $O(N^2)$ direct summation method and its variations on distributed-memory parallel machines. *New Astronomy 7*, 7 (Outubro 2002), 373–384.

[109] Martello, S., and Toth, P., Eds. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons Ltd, 1990.

[110] Mendes, H. A. HLogP: Um modelo de escalonamento para a execução de aplicações MPI em grades computacionais. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2004.

[111] Message Passing Forum. MPI: A Message Passing Interface. Technical report, University of Tennessee, 1995.

[112] Milosavljevic, M., and Merritt, D. Formation of galactic nuclei. *Astrophysical Journal 563* (2001), 34–62.

[113] MPICH-G2. `http://www3.niu.edu/mpi/`, Último acesso 10/10/2008.

[114] Nagel, W. E., Arnold, A., Weber, M., Hoppe, H. C., and Solchenbach, K. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 12*, 1 (1996), 69–80.

[115] Nakajima, J., and Pallipadi, V. Enhancements for hyper-threading technology in the operating system: seeking the optimal scheduling. In *WIESS'02: Proceedings of the 2nd conference on Industrial Experiences with Systems Software* (Berkeley, EUA, 2002), USENIX Association, pp. 3–18.

[116] Nascimento, A. P. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brasil, Maio 2008.

[117] Nascimento, A. P., Boeres, C., and Rebello, V. E. F. Dynamic self-scheduling for parallel applications with task dependencies. In *Proceedings of the 6th international Workshop on Middleware for Grid Computing* (Leuven, Belgium, December 2008), ACM Press.

[118] Nascimento, A. P., Sena, A. C., Boeres, C., and Rebello, V. E. F. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience 19*, 14 (2007), 1955–1974.

[119] Nascimento, A. P., Sena, A. C., da Silva, J. A., Vianna, D. Q. C., Boeres, C., and Rebello, V. Managing the execution of large scale MPI applications on computational grids. In *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), IEEE Computer Society.

[120] Nascimento, A. P., Sena, A. C., Silva, J. A., Vianna, D. Q. C., Boeres, C., and Rebello, V. E. F. Autonomic application management for large scale MPI programs. *International Journal of High Performance Computing and Networking (IJHPCN) (aceito para publicação em 2008)* (2008).

[121] Norman, M. G., Chochia, G., Thanisch, P., and Issman, E. Predicting the performance of the diamond DAG computational. Technical report EPCC-TR-92-07, Edinburgh Parallel Computing Centre, 1992.

[122] Németh, Z., and Sunderam, V. A comparison of conventional distributed computing environments and computational grids. In *Proceedings of the International Conference on Computational Science-Part II (ICCS '02)* (Londres, Reino Unido, 2002), Springer-Verlag, pp. 729–738.

[123] Openmp. `http://openmp.org/`, Último acesso 19/01/2009.

[124] Pacheco, P. S. *Parallel Programming with MPI.* Morgan Kaufmann Publishers Inc., 1996.

[125] Papadimitriou, C., and Yannakakis, M. Towards an architecture independent analysis of parallel algorithms. *SIAM Journal on Computing 19*, 2 (Abril 1990), 322–328.

[126] Pezzi, G., Cera, M., Mathias, E., and Maillard, N. On-line scheduling of MPI-2 programs with hierarchical work stealing. *19th International Symposium on Computer Architecture and High Performance Computing.* (Outubro 2007), 247–254.

[127] Plastino, A., Thomé, V., Vianna, D., Costa, R., and da Silveira Filho, O. T. Load balancing in SPMD applications: concepts and experiments. 95–107.

[128] Quinn, M. J., Ed. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.

[129] Quinn, M. J. *Parallel programming in C with MPI and OpenMP.* Mc Graw Hill, 2003.

[130] Rodrigues, H., Trannin, H., Sena, A. C., and Rebello, V. Usando grades computacionais para avaliação de heurísticas de escalonamento de tarefas. In *Proceedings of the VI Workshop em Sistemas Computaionais de Alto Desempenho (WSCAD 2005)* (Outubro 2005), M. C. S. de Castro and E. Midorikawa, Eds.

[131] Sakellariou, R., and Zhao, H. A hybrid heuristic for dag scheduling on heterogeneous systems. *ipdps 02* (2004), 111b.

[132] Sena, A. C., Nascimento, A. P., Boeres, C., and Rebello, V. E. F. Easygrid enabling of iterative tightly-coupled parallel MPI applications. *International Symposium on Parallel and Distributed Processing with Applications (ISPA-08)* (Dezembro 2008).

[133] Sena, A. C., Nascimento, A. P., Silva, J. A., Vianna, D. Q. C., Boeres, C., and rebello, V. E. F. On the advantages of an alternative MPI execution model for grids. In *Proceedings of 7th International Symposium on Cluster Computing and the Grid (CCGrid 2007)* (Rio de janeiro, Brasil, Maio 2007), IEEE Computer Society, pp. 575–582.

[134] Shires, D. R., Pollock, L. L., and Sprenkle, S. Program flow graph construction for static analysis of MPI programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (1999), pp. 1847–1853.

[135] Sievert, O., and Casanova, H. A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications 18*, 3 (2004), 341–352.

[136] Silva, R. E., Pezzi, G., Maillard, N., and Diverio, T. Automatic dataflow graph generation of MPI programs. In *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), IEEE Computer Society.

[137] Grid Sinergia. `http://easygrid.ic.uff.br/`, Último acesso 10/10/2008.

[138] Sinnen, O. *Task Scheduling for Parallel Systems.* Wiley, 2007.

[139] Sinnen, O., and Sousa, L. A comparative analysis of graph models to develop parallelising tools. In *Proceedings of the 8th IASTED Int'l Conference on Applied Informatics (AI'2000), Innsbruck, Austria, February 2000.* (2000).

[140] Sinnen, O., and Sousa, L. A platform independent parallelising tool based on graph theoretic models. *Lecture Notes in Computer Science 1981* (2001), 154–167.

[141] Smallen, S., Casanova, H., and Berman, F. Applying scheduling and tuning to on-line parallel tomography. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)* (Nova Iorque, EUA, 2001), ACM Press, pp. 12–12.

[142] Souto, H., da Silveira Filho, O., Moyne, C., and Didierjean, S. Thermal dispersion in porous media: Computations by the random walk method. *Journal of Computational and Applied Mathematics 21*, 2 (2002), 513–544.

[143] Stiles, J. R., Thomas M. Bartol, J., Salpeter, E. E., and Salpeter, M. M. Monte carlo simulation of neuro-transmitter release using mcell, a general simulator of cellular physiological processes. In *CNS '97: Proceedings of the sixth Annual Conference on Computational Neuroscience : Trends in research, 1998* (Nova Iorque, EUA, 1998), Plenum Press, pp. 279–284.

[144] Suter, F., Casanova, H., Desprez, F., and Boudet, V. From heterogeneous task scheduling to heterogeneous mixed data and task parallel scheduling. In *Proceedings of the 10th International Euro-Par Conference on Parallel Computing (EuroPar 2004)* (Pisa, Itália, Agosto 2004), Springer.

[145] Thain, D., Tannenbaum, T., and Livny, M. *Condor and the Grid.* John Wiley and Sons Ltd, 2003.

[146] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience 17*, 2-4 (2005), 323–356.

[147] Topcuoglu, H., Hariri, S., and Wu, M. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems 13*, 3 (Março 2002), 260–274.

[148] Ullman, J. D. NP-complete scheduling problems. *Journal of Computer and System Sciences 10*, 3 (1975), 384–393.

[149] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[150] VAMPIR. `http://www.vampir.eu/`, Último acesso 10/10/2008.

[151] Vianna, B. A., Fonseca, A. A., Moura, N. T., Menezes, L. T., Mendes, H. A., Silva, J. A., Boeres, C., and Rebello, V. E. F. A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In *Proceedings of the 2nd workshop on Middleware for Grid Computing* (2004), ACM Press, pp. 41–46.

[152] Vianna, D. Q. C. Um sistema de gerenciamento de aplicações MPI para ambientes grid. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2005.

[153] Williams, T. L., and Parsons, R. J. The heterogeneous bulk synchronous parallel model. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing (IPDPS '00)* (Londres, Reino Unido, 2000), Springer-Verlag, pp. 102–108.

[154] Wolski, R., Spring, N., and Hayes, J. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing 3*, 4 (Outubro 2000), 293–301.

[155] Woo, N., Jung, H. S., Yeom, H. Y., Park, T., and Park, H. MPICH-GF: Transparent checkpointing and rollback recovery for grid-enabled MPI processes. *IEICE Transactions on Information and Systems E87-D*, 7 (Julho 2004), 1820–1828.

[156] Wright, A., and Webb, R. The large hadron collider. *Nature Insight: 448*, 7151 (2007), 269–312.

[157] Yang, T., and Fu, C. Heuristic algorithms for scheduling iterative task computations on distributed memory machines. *IEEE Transaction on Parallel and Distributed Systems 8*, 6 (1997), 608–622.

[158] Yang, T., and Gerasoulis, A. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th International Conference on Supercomputing* (Nova Iorque, EUA, 1992), ACM Press, pp. 428–437.

[159] Yu, J., and Buyya, R. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing 3*, 3-4 (2005), 171–200.