Estudo comparativo de técnicas de escalonamento de tarefas dependentes para grades computacionais

Alvaro Henry Mamani Aliaga

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Alfredo Goldman vel Lejbman

Durante o desenvolvimento deste trabalho o autor recebeu apoio financeiro da CNPq, processo 133147/2009-6

São Paulo, julho de 2011

Estudo comparativo de técnicas de escalonamento de tarefas dependentes para grades computacionais

Esta dissertação contém as correções e alterações
sugeridas pela Comissão Julgadora durante a defesa
realizada por Alvaro Henry Mamani Aliaga em 22/08/2011.
O original encontra-se disponível no Instituto de
Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Alfredo Goldman vel Lejbman - IME-USP
- Prof. Dra. Liria Matsumoto Sato - EP-USP
- Prof. Dr. Philippe Olivier Alexandre Navaux - UFRGS

Dedico este trabalho à minha querida mãe Toribia,
ao meu pai Emigio e ao Jorge Miguel.
Muito obrigado pelo carinho, dedicação e por sempre estar comigo.

Agradecimentos

Este trabalho, não foi somente uma dissertação de mestrado, foi muito mais do que isso, pois durante este tempo aprendi, sofri, ri e principalmente conheci ótimas pessoas.

Primeiramente quero agradecer à minha família, principalmente aos meus pais Toribia e Emigio, pelo apoio incondicional, pelo carinho e dedicação. Ao meu irmão Jorge Miguel, exemplo a seguir, obrigado pelos conselhos e por ser sempre um exemplo para mim. À Marilu, ao Adolfo, à Hermes, à Pahola, à Maru, e aos meus queridos tios Andres e Angelica, obrigado a todos vocês pelas demostrações de afeto e carinho.

Ao meu orientador e amigo Professor Alfredo Goldman, foi ele que me ofereceu um grande apoio aqui no Brasil, ele é uma ótima pessoa, excelente orientador e bom amigo. Obrigado Alfredo por todas as dicas, palavras e principalmente por acreditar em mim. Ao Yanik Ngoko pelas sugestões sobre o meu trabalho e as constantes observações para melhorá-lo.

Quero agradecer em especial à *Edith*, minha namorada, ela sempre está ao meu lado, nos momentos bons e nos momentos ruins, mesmo sendo eu uma pessoa muito difícil ela esta ai para me apoiar, ela é meu "braço direito", amiga, parceira, e boa namorada, muito obrigado Didito.

Eu quero agradecer ao Jesus Mena, um bom amigo que sempre me ofereceu um apoio incondicional e voluntario, ao qual eu sempre procurava quando tinha duvidas e problemas, o Jesus sempre tinha as palavras exatas para eu entender as coisas, obrigado Jesus, você é o caminho :).

Também aos meus amigos peruanos, especialmente ao Caratos, o Alfonso, o Carlitos, a Erika, o Edwin, o Daniel, o Jorge, a Leissi, o Harry e aos amigos Cuzqueños :).

Um paragrafo especial merecem os meus amigos do laboratório de computação paralela e distribuída (LCPD). Meu amigo colombiano Edwin, ótima pessoa, um cara muito legal. Meu amigo Gustavo, um rapaz muito legal, bom parceiro, sempre gostando das quintas-feiras. O Vinicius, o mestre, sempre dando dicas positivas. O Felipe, o Paulo, a Claudia, o Yanik, o Raphael, ... Obrigado por todos os momentos dentro e fora do laboratório, vocês também foram uns importantes autores no meu trabalho. Obrigado por todas as críticas e correções do português durante todo o meu mestrado, por tudo, obrigado gente!!!

Quero agradecer também aos meus amigos de Arequipa-Peru e da UNSA que fazem que minhas visitas à minha cidade natal seja sempre inesquecível, são muitos que não da para mencioná-los.

Às equipes que desenvolveram as ferramentas que usei no meu mestrado. À gente do projeto SimGrid, tanto da parte do simulador quanto das contribuições. Às pessoas do projeto Pegasus, que disponibilizaram as especificações das aplicações para grade, esse foi um grande aporte para pessoas que pesquisam em escalonamento e aos variados projetos GNU/Linux.

Ao CPNq, a bolsa foi um grande alivio, e deixou para um lado as minhas preocupações econômicas.

Finalmente a toda a galera do IME que me conhece, alguns amigos da matemática e estatística

e funcionários.

Tem muitas pessoas além das que eu mencionei, as quais merecem ser agradecidas, se for que esqueci de colocar alguém, não se sintam ofendidos, sabem que eu fico muito grato com todos vocês :D

Resumo

À medida que a ciência avança, muitas aplicações em diferentes áreas precisam de grande poder computacional. A computação em grade é uma importante alternativa para a obtenção de alto poder de processamento, no entanto, esse alto poder computacional deve ser bem aproveitado. Mediante o uso de técnicas de escalonamento especializadas, os recursos podem ser utilizados adequadamente. Atualmente existem vários algoritmos propostos para computação em grade, portanto, é necessário seguir uma boa metodologia para escolher o algoritmo que ofereça melhor desempenho, dadas determinadas características. No presente trabalho comparamos os algoritmos de escalonamento para tarefas dependentes: (a) *Heterogeneous Earliest Finish Time* (HEFT), (b) *Critical Path on a Processor* (CPOP) e (c) *Path Clustering Heuristic* (PCH); cada algoritmo é avaliado com diferentes aplicações e sobre diferentes arquiteturas usando técnicas de simulação, seguindo quatro critérios: (i) desempenho, (ii) escalabilidade, (iii) adaptabilidade e (iv) distribuição da carga do trabalho. Diferenciamos as aplicações para grade em dois tipos: (i) aplicações regulares e (ii) aplicações irregulares; dado que em aplicações irregulares não é facil comparar o critério de escalabilidade. Seguindo esse conjunto de critérios o algoritmo HEFT possui o melhor desempenho e escalabilidade; enquanto que os três algoritmos possuem o mesmo nível de adaptabilidade. Na distribuição de carga de trabalho o algoritmo HEFT aproveita melhor os recursos do que os outros. Por outro lado os algoritmos CPOP e PCH usam a técnica de escalonar o caminho crítico no processador que ofereça o melhor tempo de término, mas essa abordagem nem sempre é a mais adequada.
Palavras-chave: Computação em grade, algoritmos de escalonamento, plataformas para grades, *workflows*, simulação.

Abstract

As science advances, many applications in different areas need a big amount of computational power. Grid computing is an important alternative to obtain high processing power, but this high computational power must be well used. By using specialized scheduling techniques, resources can be properly used. Currently there are several algorithms for grid computing, therefore, is necessary to follow a good methodology to choose an algorithm that offers better performance given certain settings. In this work, we compare task dependent scheduling algorithms: (a) Heterogeneous Earliest Finish Time (HEFT), (b) Critical Path on a Processor (CPOP) e Path Clustering Heuristic (PCH); each algorithm is evaluated with different applications and on different architectures using simulation techniques, following four criterias: (i) performance, (ii) scalability, (iii) adaptability and (iv) workload distribution. We distinguish two kinds of grid applications: (i) regular applications and (ii) irregular applications, since in irregular applications is not easy to compare scalability criteria. Following this set of criteria the HEFT algorithm reaches the best performance and scalability, while the three algorithms have the same level of adaptability. In workload distribution HEFT algorithm makes better use of resources than others. On the other hand, CPOP and PCH algorithms use scheduling of tasks which belong to the critical path on the processor which minimizes the earliest finish time, but this approach is not always the most appropriate.
Keywords: Grid computing, scheduling algorithms, task scheduling, grid platforms, workflows, simulation.

Sumário

Lista de Abreviaturas **xiii**

Lista de Algoritmos **xv**

Lista de Figuras **xvii**

Lista de Tabelas **xix**

1 Introdução **1**
1.1 Motivação 2
1.2 Objetivos 2
1.3 Contribuições 2
1.4 Organização do Trabalho 3

2 Conceitos **5**
2.1 Aglomerado 5
2.2 Computação em Grade 5
2.3 Escalonamento 6
2.3.1 Escalonamento Estático 6
2.3.2 Escalonamento Dinâmico 7
2.4 Escalonamento de Tarefas 7

3 Escalonadores **9**
3.1 O escalonador OAR 9
3.1.1 Arquitetura 9
3.1.2 Escalonamento 10
3.2 O escalonador Condor 11
3.2.1 Arquitetura 11
3.2.2 Escalonamento 12
3.3 O escalonador PBS/OpenPBS/Torque 13
3.3.1 Arquitetura 14
3.3.2 O escalonador Torque 14
3.4 Considerações Finais do Capítulo 16

4 Middlewares para Grades **19**
4.1 Boinc 19

4.1.1 Arquitetura . . . 19
4.1.2 Escalonamento . . . 20
4.2 InteGrade . . . 20
4.2.1 Arquitetura . . . 21
4.2.2 Escalonamento . . . 21
4.3 OurGrid . . . 22
4.3.1 Arquitetura . . . 22
4.3.2 Escalonamento . . . 23
4.4 XtremWeb . . . 24
4.4.1 Arquitetura . . . 24
4.4.2 Escalonamento . . . 24
4.5 Considerações Finais do Capítulo . . . 25

5 Arquiteturas para Grades **27**
5.1 O Projeto DAS-3 . . . 27
5.1.1 Arquitetura . . . 28
5.2 O Projeto Grid5000 . . . 28
5.2.1 Arquitetura . . . 29
5.3 O Projeto GridPP . . . 31
5.3.1 Arquitetura . . . 31
5.4 Considerações Finais do Capítulo . . . 32

6 Aplicações para Grades **35**
6.1 A aplicação Montage . . . 35
6.2 A aplicação CyberShake . . . 37
6.3 A aplicação Epigenomics . . . 38
6.4 A aplicação LIGO . . . 40
6.5 Considerações Finais do Capítulo . . . 41

7 Simulador **43**
7.1 Simuladores para Grades . . . 43
7.1.1 Bricks . . . 43
7.1.2 Optorsim . . . 44
7.1.3 GridSim . . . 44
7.1.4 SimGrid . . . 44
7.2 O Simulador SimGrid . . . 45
7.2.1 Arquitetura . . . 45
7.2.2 Componentes do SimGrid . . . 45
7.2.3 Implementação e Documentação . . . 47
7.2.4 Modelagem da Plataforma da Grade e os Workloads . . . 47
7.3 Considerações Finais do Capítulo . . . 48

8 Algoritmos de Escalonamento **49**
8.1 Algoritmos de Escalonamento para Tarefas Independentes 49
8.1.1 O Algoritmo WQR 49
8.1.2 O Algoritmo XSufferage 50
8.1.3 O Algoritmo Storage Affinity 51
8.2 Algoritmos de Escalonamento para Tarefas Dependentes 51
8.2.1 Problema de Escalonamento para Tarefas Dependentes 51
8.2.2 Atributos do Grafo usado pelos Algoritmos de Escalonamento 53
8.2.3 O Algoritmo HEFT 54
8.2.4 O Algoritmo CPOP 55
8.2.5 O Algoritmo PCH 57
8.3 Considerações Finais do Capítulo 59

9 Resultados Experimentais **61**
9.1 Análise das Aplicações 62
9.1.1 Análise das Estruturas das Aplicações 64
9.2 Resultados Experimentais 64
9.2.1 Resultados 65

10 Conclusões **81**
10.1 Considerações Finais 81
10.2 Sugestões para Pesquisas Futuras 82

A Exemplo de Especificação das Arquiteturas **83**

Referências Bibliográficas **89**

Índice Remissivo **95**

Lista de Abreviaturas

API	Application Programming Interface
ASCI	Advanced School for Computing and Imaging
ASCT	Application Submission and Control Tool
BOINC	Berkeley Open Infrastructure for Network Computing
BoT	Bag of Tasks (Saco de Tarefas)
CERN	European Organization for Nuclear Research
CPOP	Critical Path on a Processor
DAG	Directed Acyclic Graph (Grafos Acíclicos Dirigidos)
DAS-3	Distributed ASCI Supercomputer 3
DAX	Directed Acyclic Graph in XML
EGEE	Enabled Grid for E-sciencE
EGI	European Grid Intrastructure
FCFS	First Come First Served
FIFO	First In First Out
FITS	Flexible Image Transport System
GRM	Global Resource Manager
GUPA	Global Usage Pattern Analyzer
HEFT	Heterogeneous Earliest Finish Time
LCG	LCH Computing Grid
LHC	Large Hadron Collider
LRM	Local Resource Manager
LUPA	Local Usage Pattern Analyzer
NCC	Node Control Center
PCH	Path Clustering Heuristic
PSHA	Probabilistic Seismic Hazard Analysis
RENATER	Le Réseau National de télécommunications pour la Technologie l'Enseignement et la Recherch
SETI	Search for ExtraTerrestrial Intelligence
WQR	Workqueue With Replication

Lista de Algoritmos

8.1 O Algoritmo HEFT 55
8.2 O Algoritmo CPOP 56
8.3 Gerar_agrupamento 57
8.4 Seleciona_melhor_recurso 58
8.5 O Algoritmo PCH 58

Lista de Figuras

2.1 Classificação dos métodos de escalonamento. 6
2.2 Aplicação com tarefas dependentes. 7
2.3 Aplicação com tarefas não dependentes. 7

3.1 Arquitetura global do escalonador OAR. 10
3.2 Visão geral da arquitetura no Condor. 12
3.3 Mecanismo Matchmaking no Condor. 13
3.4 Dois exemplos de *Classified Advertisements* do escalonador Condor. 13
3.5 Componentes do PBS. 14

4.1 Componente de um servidor Boinc. 20
4.2 Arquitetura do InteGrade. 21
4.3 Arquitetura do *middleware* OurGrid. 23

5.1 A arquitetura do projeto DAS-3. 29
5.2 A arquitetura do projeto Grid5000. 30
5.3 A arquitetura do projeto GridPP. 32

6.1 Estrutura da aplicação Montage. 36
6.2 Estrutura da aplicação CyberShake. 38
6.3 Estrutura da aplicação Epigenomics. 39
6.4 Estrutura da aplicação LIGO. 41

7.1 Componentes do simulador SimGrid. 46
7.2 Camada: Ambientes de Programação. 46

9.1 Representação da arquitetura *Small Grid* usado nos experimentos. 61
9.2 Representação dos comportamentos das aplicações. 63
9.3 Resultados do desempenho avaliado na arquitetura *Small Grid*. 66
9.4 Resultados do desempenho avaliado nas arquiteturas de larga escala. 67
9.5 Resultados da simulação da aplicação Montage avaliada sobre a arquitetura *Small Grid*. 68
9.6 Resultados da simulação da aplicação Montage avaliada sobre as arquiteturas de grande porte. 69
9.7 Resultados da simulação da aplicação CyberShake avaliada sobre a arquitetura *Small Grid*. 70

9.8 Resultados das simulações de 20 instâncias da Aplicação CyberShake. 70
9.9 Resultados da simulação da aplicação CyberShake avaliada sobre as arquiteturas de grande porte. 71
9.10 Resultados da simulação da aplicação Ligo avaliada sobre a arquitetura *Small Grid*. . 72
9.11 Resultados da simulação da aplicação Ligo avaliada sobre as arquiteturas de grande porte. 73
9.12 HEFT: Montage de 50 Tarefas sobre DAS-3. 76
9.13 CPOP - Montage de 50 Tarefas sobre DAS-3. 76
9.14 PCH - Montage de 50 Tarefas sobre DAS-3. 77
9.15 HEFT: Montage de 500 Tarefas sobre DAS-3. 77
9.16 CPOP: Montage de 500 Tarefas sobre DAS-3. 78
9.17 PCH: Montage de 500 Tarefas sobre DAS-3. 78
9.18 HEFT: Montage de 1000 Tarefas sobre DAS-3. 79
9.19 CPOP: Montage de 1000 Tarefas sobre DAS-3. 79
9.20 CPOP: Montage de 1000 Tarefas sobre DAS-3. 80

Lista de Tabelas

5.1 Características da arquitetura DAS-3. 28
5.2 Características da arquitetura Grid5000. 30
5.3 Características da arquitetura GridPP. 33

6.1 Quantidade de tarefas por nível da aplicação Montage. 36
6.2 Quantidade de tarefas por nível da aplicação Montage. 37
6.3 Quantidade de tarefas por nível da aplicação CyberShake. 38
6.4 Quantidade de tarefas por nível da aplicação Epigenomics. 40
6.5 Quantidade de tarefas por nível da aplicação Ligo. 42

9.1 Conjunto de recursos computacionais da arquitetura *Small Grid*. 62
9.2 Critérios para a comparação dos algoritmos. 65
9.3 Adaptabilidade do algoritmo HEFT. 74
9.4 Adaptabilidade do algoritmo CPOP. 74
9.5 Adaptabilidade do algoritmo PCH. 74

Capítulo 1

Introdução

Nas últimas décadas, a computação tornou-se necessária para todas as áreas de pesquisa. À medida que a ciência avança, grandes quantidades de dados precisam ser processados e cientistas precisam dos resultados desses processamentos, os quais, precisam de recursos computacionais de alto desempenho.

Nos anos recentes, a disponibilidade de computadores poderosos tem aumentado enormemente, assim como a sua interligação com redes de alta velocidade. Este fato tem permitido a agregação de recursos sem importar a distância entre eles, para obter grande capacidade no processamento, principalmente para execução de tarefas de aplicações computacionais de grande escala e com uso intensivo de recursos. Esta agregação de recursos geograficamente dispersos tem sido denominada Computação em Grade (*Grid Computing*) [FKNT02], uma alternativa para obter grande capacidade de processamento.

A computação em grade compreende uma complexa infraestrutura composta por soluções integradas de hardware e software que permitem o compartilhamento de recursos distribuídos sob a responsabilidade de instituições distintas [FK04]. Esses ambientes são alternativas atraentes para a execução de aplicações paralelas ou distribuídas que demandam alto poder computacional, tais como mineração de dados, previsão do tempo, biologia computacional, física de partículas, processamento de imagens médicas, entre outras [BFH03]. Essas aplicações são compostas por diversas tarefas que, a depender do tipo de aplicação, podem se comunicar durante a fase de execução.

Na computação em grade, os recursos computacionais são heterogêneos e podem ser agregados ou retirados do ambiente a qualquer momento. Neste cenário, o "escalonamento (*scheduling* em inglês) de tarefas" é um grande desafio que tem como objetivo principal atingir um bom desempenho no tempo de execução de aplicações independentemente do tipo destas.

Um dos objetivos principais nos algoritmos de escalonamento é minimizar o tempo de término das aplicações (*Makespan*), mediante o escalonamento dos seus componentes de forma a maximizar o paralelismo na execução das tarefas e minimizar a comunicação. No escalonamento, uma aplicação pode ser composta de tarefas que possuem dependência de dados (onde a execução de cada tarefa deve respeitar uma precedência), ou de tarefas independentes, isto significa que não possuem dependência, como por exemplo, aplicações do tipo *Bag-of-Tasks* [dSCB03].

O problema de escalonamento de tarefas é um problema antigo que motiva muitas pesquisas em diversas áreas [CK88, THW02, dSCB03, BM06]. Atualmente existem vários algoritmos de escalonamento propostos na literatura [dSCB03, SNCBL04, CLZB00, BM06, THW02], no entanto, não é fácil estimar qual algoritmo é melhor usar sobre determinadas características para minimizar, por

exemplo, o tempo de resposta. O uso de um algoritmo de escalonamento eficiente deve garantir essa eficiência em diferentes arquiteturas para grade, e de forma similar em diferentes aplicações, assim, oferecer um bom desempenho na execução dessas aplicações. Neste trabalho é apresentada uma comparação de técnicas de escalonamento para tarefas com dependência sobre diferentes arquiteturas. Além disso, propõe uma abordagem de quatro critérios para a comparação de algoritmos de escalonamento tomando em consideração o tipo de aplicação e o tipo de arquitetura da grade.

1.1 Motivação

A computação em grade é uma alternativa para obter grande capacidade no processamento, sendo assim, uma ferramenta necessária na computação em grande escala. Muitas áreas em ciência atualmente utilizam a computação em grade para o processamento de dados. Um bom exemplo disso é o *Large Hadron Collider* (LHC) do projeto LCG no CERN[1].

O escalonamento de tarefas é um problema NP-Completo [Pin08], estudado já em muitas pesquisas, mas no ambiente da computação em grade o problema de escalonamento se torna um grande desafio devido às características da grade: dinamicidade, heterogeneidade e recursos fisicamente distantes uns dos outros.

Para que a computação em grade possa atingir um bom desempenho, é necessário um conhecimento adequado do cenário: (i) um estudo aprofundado do escalonamento, (ii) o tipo de aplicação que será escalonada e (iii) o tipo de arquitetura onde se realizará o escalonamento. Portanto, uma análise dos escalonadores, dos algoritmos de escalonamentos, da arquitetura e o tipo de aplicação são importantes para atingir um bom desempenho.

1.2 Objetivos

O objetivo geral do presente trabalho consiste em comparar diferentes técnicas de escalonamento para grades computacionais em diferentes cenários, avaliando o desempenho de cada técnica escolhida, levando em consideração o tipo de arquitetura e o tipo de aplicação.

Entre os objetivos específicos do trabalho, temos os seguintes:

(a) Comparamos algoritmos de escalonamento com base em um conjunto de traços de execução (*traces*) gerados a partir de aplicações reais;

(b) Desenvolver uma metodologia que baseada nas características das aplicações e arquiteturas para grade apoie na decisão de qual algoritmo de escalonamento oferecerá o menor *Makespan*;

(c) Determinar se dado um tipo de aplicação é adequado fazer uma comparação entre algoritmos de escalonamento.

1.3 Contribuições

As principais contribuições deste trabalho estão discriminadas abaixo:

[1] http://public.web.cern.ch/public/, último acesso no 02 de junho de 2011.

(a) Classificação dos tipos de aplicações para grade com tarefas dependentes: (i) regulares e (ii) irregulares;

(b) Uma metodologia para fazer comparação de algoritmos de escalonamento, baseada em determinadas configurações e métricas;

(c) Atualização, modelagem e especificação para a simulação das arquiteturas para grade: (i) DAS-3, (ii) Grid5000 e (iii) GridPP, sobre o simulador SimGrid v3.5;

(d) Repositório de imagens da distribuição das tarefas dos resultados do escalonamento, criadas a partir das simulações dos algoritmos.

1.4 Organização do Trabalho

No Capítulo 2 apresentamos os conceitos básicos da computação em grade, escalonadores e algoritmos de escalonamento; necessários para o entendimento do escopo do presente trabalho.

No Capítulo 3 é feita uma descrição dos principais sistemas de escalonamento para aglomerados, com ênfase nas técnicas de escalonamento que usam.

No Capítulo 4 são apresentados os principais *middlewares* para computação em grade, atuais, é dada uma explicação sobre como é o escalonamento em cada um deles.

No Capítulo 5 são apresentadas as arquiteturas para computação em grade. Nesse capítulo são apresentadas três arquiteturas que foram criadas principalmente para fazer pesquisa em computação em grade, computação paralela, distribuída e de alto desempenho.

No Capítulo 6 são descritas e analisadas as aplicações usadas no presente trabalho. Essas aplicações foram especificadas no Projeto Pegasus e disponibilizadas para interesses de pesquisa.

No Capítulo 7 são apresentados os detalhes sobre o ambiente de simulação, como é a arquitetura do simulador, os modos na implementação e como representar a arquitetura e ambientes de trabalho.

Os escalonadores estudados são apresentados no Capítulo 8. Nesse capítulo são apresentados tanto os algoritmos para tarefas dependentes quanto para tarefas independentes. Neste capítulo é dado maior detalhe principalmente aos algoritmos para tarefas dependentes.

Resultados experimentais serão mostrados no Capítulo 9, são apresentados os resultados de todas as simulações.

Finalmente, no Capítulo 10 serão apresentadas as conclusões do trabalho.

Em anexos está uma exemplificação da especificação da arquitetura, neste caso é descrita a especificação da arquitetura DAS-3 (Apêndice A).

Capítulo 2

Conceitos

Na computação em grade temos diferentes termos, às vezes dependentes do contexto. Os conceitos básicos de computação em grade e escalonamento são apresentados neste capítulo.

Inicialmente, são apresentados conceitos básicos sobre aglomerados e computação em grade e as suas principais características. Neste capítulo também são descritos conceitos sobre os tipos de escalonamento: estático e dinâmico. Finalmente, o escalonamento de tarefas é aprofundado.

2.1 Aglomerado

Um aglomerado (também chamado *cluster*, em inglês ou *agregados*, como alguns autores se referem em português) [Dan05], em termos de arquiteturas computacionais, é entendido como um conjunto de computadores, os quais podem ser de uso exclusivo ou não, para a execução de aplicações específicas de uma organização. Um aglomerado de uso exclusivo possui todos os recursos computacionais dedicados exclusivamente na execução de aplicações paralelas ou seriais. Por outro lado, na configuração não exclusiva, além da execução de aplicações monoprocessadas, pode ser utilizado como um aglomerado eventual para execução de aplicações que solicitem um maior desempenho computacional agregado.

Os aglomerados, de uma forma geral, são compostos por computadores com uma característica intrínseca de disponibilidade de uma grande quantidade de recursos (processadores, memórias e capacidade de armazenamento) pertencentes a uma única entidade.

2.2 Computação em Grade

A "Computação em Grade" emergiu como uma importante nova área em meados da década de 1990 e nasceu da comunidade de "Processamento de Alto Desempenho", motivada pela ideia de se utilizar computadores independentes e amplamente dispersos como plataforma de execução de aplicações paralelas [FK04].

A ideia de computação em grade fundamenta-se no compartilhamento de recursos de forma coordenada e dinâmica entre várias instituições geograficamente distantes. Cada instituição compartilha seus recursos sob determinadas condições no uso, e faz uso dos recursos disponibilizados na grade.

Este compartilhamento coordenado é feito através de um *middleware*. Um *middleware* é um pacote de software que faz a interface entre o usuário e o ambiente computacional. No caso de

computação em grade existem diversas infraestruturas de *middleware* desenvolvidas até agora, por exemplo, Globus [Fos05], Legion [GW97], InteGrade [GKG+04] e OurGrid [CBA+06]. Essas já permitem que coleções de máquinas heterogêneas distribuídas em aglomerados fisicamente distantes, mas interconectadas por redes de longa distância como a Internet, trabalhem em conjunto para a resolução de problemas computacionalmente pesados.

2.3 Escalonamento

O escalonamento é a atribuição de tarefas aos elementos de processamento (recursos). O objetivo é que essa atribuição seja efetuada de forma eficiente para minimizar o tempo das aplicações.

O objetivo pode ser maximizar a utilização dos recursos computacionais disponíveis, e minimizar os custos relativos à comunicação, isto significa minimizar o tempo de término das aplicações, *Makespan*. Diferentes tipos possíveis de escalonamento foram estudados por vários pesquisadores. Uma abordagem da classificação mais aceita está apresentada na Figura 2.1 [Dan05].

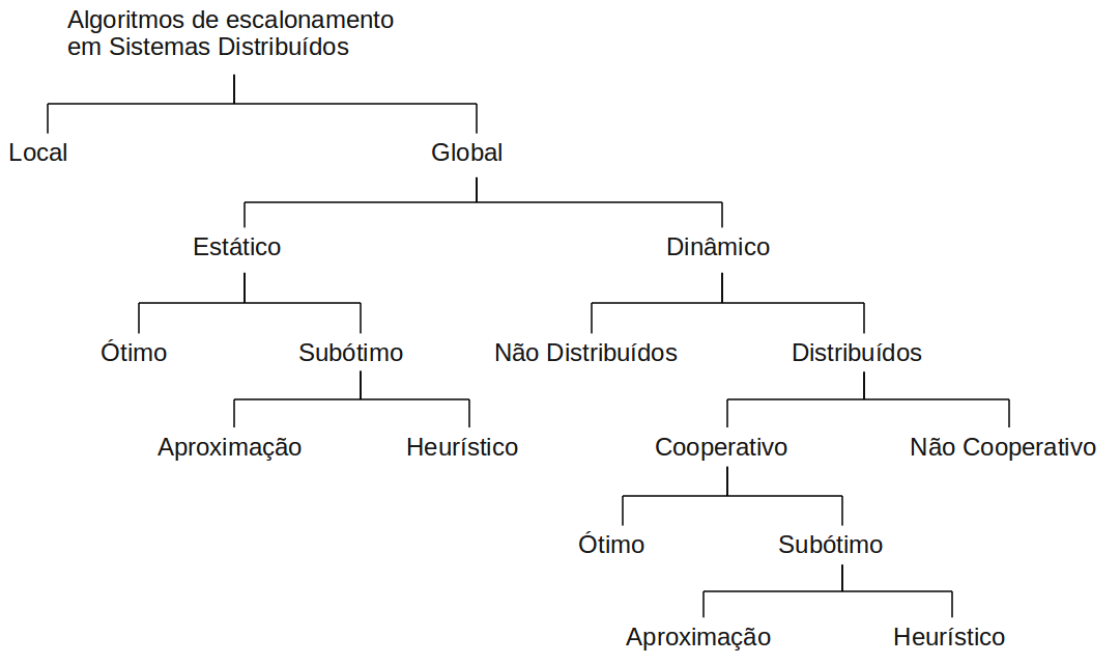


Figura 2.1: *Classificação dos métodos de escalonamento. Esquema adaptado de [CK88].*

Na classificação, inicialmente, os métodos de escalonamento são divididos em local e global. O escalonamento local refere-se ao problema de atribuição das tarefas ao processador local, ou seja, é aquele realizado normalmente pelo sistema operacional. O escalonamento global refere-se ao problema de decidir sobre quais recursos executar as tarefa de uma aplicação em um ambiente distribuído.

2.3.1 Escalonamento Estático

No escalonamento estático, a atribuição de tarefas aos processadores é realizada antes do início do programa. Assim, a atribuição de uma aplicação é estática, e uma estimativa do custo computacional deve ser feita com antecedência. Uma das principais vantagens do modelo estático é que é mais fácil programar do ponto de vista do escalonador. A atribuição de tarefas é fixada *a priori*, e

a estimativa do custo da tarefa também é simplificada.

2.3.2 Escalonamento Dinâmico

O escalonamento dinâmico é geralmente aplicado quando é difícil estimar o custo das aplicações, ou as tarefas sendo submetidas em tempo real, ou dinamicamente (*online scheduling*). Estes assumem que muito pouco se sabe, *a priori*, acerca das necessidades dos recursos de uma tarefa, ou do ambiente do qual a mesma irá ser executada. No escalonamento dinâmico, pode ser realizada a redistribuição das tarefas aos processadores durante a execução da aplicação.

2.4 Escalonamento de Tarefas

Um dos principais passos realizados no processo de execução de aplicações em grades é o escalonamento de tarefas que compõem essas aplicações. As tarefas que compõem uma aplicação podem ter dependências entre si. Quando as dependências existem, as tarefas formam grafos direcionados para representar as dependências.

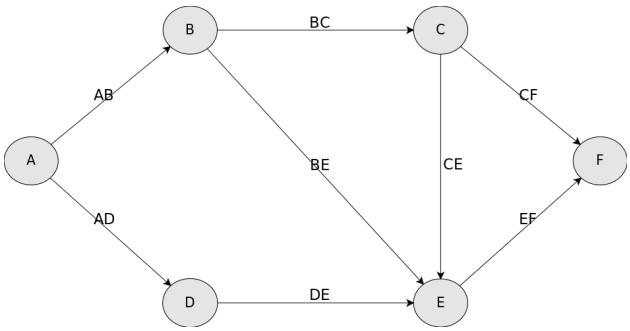


Figura 2.2: *Exemplo da descrição de uma aplicação com tarefas dependentes. Cada círculo representa uma tarefa a qual possui um custo de computação e o valor da seta representa um custo comunicação entre tarefas.*

Quando as dependências não existem, as tarefas formam grafos vazios, ou seja, grafos que não possuem arestas, e são denominadas com o nome de *Bag-of-Tasks* (BoT). A descrição da aplicação passada como entrada para o escalonador de tarefas depende do tipo de aplicação. Na Figura 2.2 é exemplificado um grafo direcionado acíclico, que representa uma aplicação na qual há dependências entre suas tarefas, onde cada círculo representa uma tarefa que possui um custo computacional e cada valor da seta que conecta dois círculos representa o custo de comunicação entre tarefas. Por outro lado a Figura 2.3 exemplifica grafos vazios de uma aplicação BoT [Bat10], onde cada círculo representa uma tarefa, cada tarefa é independente, isto significa que não possui dependência a respeito das outras.

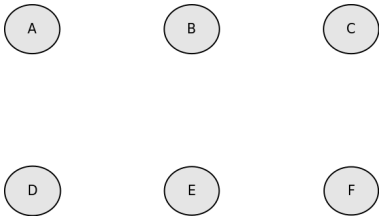


Figura 2.3: *Exemplo da descrição de uma aplicação com tarefas não dependentes. Cada círculo representa uma tarefa e cada tarefa é independente, isto significa que não possui dependência a respeito das outras.*

Capítulo 3

Escalonadores

Um sistema escalonador de tarefas (também denominado por alguns autores como gerenciador de recursos ou sistema de processamento por lote) é aquele software responsável por garantir o bom funcionamento de um sistema computacional. Dentre as funções principais temos: recebimento de tarefas por recursos e a atribuição de recursos a essas tarefas, realizando a alocação do que é buscado com o que é oferecido pela infraestrutura.

O emparelhamento entre um recurso e uma tarefa é realizado com o intuito de maximizar a qualidade de serviço estabelecida pelos usuários. A qualidade de serviço pode ser dada pela minimização do tempo de espera pelos resultados de uma tarefa. Assim, para atingir essa qualidade de serviço, existem políticas de escalonamento que ditam como, quando e onde determinada aplicação deve ser executada.

Para analisar como fazem o escalonamento de aplicações neste tipo de cenários, foram escolhidos os escalonadores mais representativos na literatura atual, os quais são usados por diferentes instituições importantes na área de computação de alto desempenho, tomando em conta somente escalonadores de código não proprietário.

3.1 O escalonador OAR

O sistema OAR [CDCG+05] é um escalonador de recursos para aglomerados de grande porte, desenvolvido no Instituto Politécnico Nacional de Grenoble, na França. O projeto é livre e possui código aberto com licença GPL, foi feito com ferramentas de alto nível, entre as quais temos:

- Gestores de Banco de dados: MySQL ou PostgreSQL;
- A linguagem *scripting* Perl;
- Um mecanismo para trabalhar com CPUSET[1];
- Uma ferramenta de execução paralela em recursos remotos denominada Taktuk [CHR09].

3.1.1 Arquitetura

O escalonador OAR é baseado em um nível mais abstrato que minimiza a complexidade de concepção de seu software. A arquitetura interna, mostrada na Figura 3.1, é construída em cima de

[1] *CPUSET* é um módulo integrado ao *kernel* de Linux com o intuito de fornecer um mecanismo para atribuir um conjunto de recursos computacionais (processadores, memória) para um conjunto de tarefas.

dois componentes principais: (i) uma ferramenta genérica e escalável para a administração do aglomerado escrita na linguagem de programação Perl, e (ii) um gestor de banco de dados (PostgreSQL ou MySql) como único jeito de compartilhar informação.

No banco de dados são armazenados todos os dados internos sobre aplicações e recursos, o acesso é unicamente por meio de comunicação entre módulos. O casamento entre recursos e o armazenamento e consulta de *logs* do sistema são realizados através de chamadas SQL. Essa prática atribui características de robustez e eficiência ao escalonador, posto que os bancos de dados têm poucas chances de se tornar um gargalo na escalabilidade do sistema por serem capazes de processar eficientemente milhares de consultas simultaneamente. A robustez somente depende dos módulos que precisam deixar o sistema em um estado coerente.

A outra parte do servidor é composta por um conjunto de módulos independentes implementados como *scripts* Perl. Cada um dos módulos é responsável por tarefas específicas, entre as principais temos: (a) iniciar e controlar a execução das tarefas e (b) escalonar e controlar as tarefas.

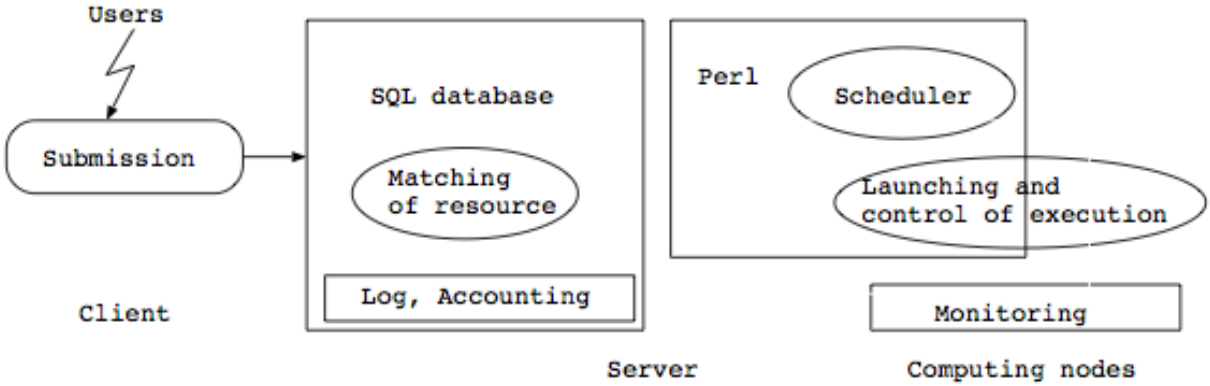


Figura 3.1: *Arquitetura global do escalonador OAR [CDCG+05].*

3.1.2 Escalonamento

Um dos objetivos do escalonador OAR, é a sua simplicidade e oferecer uma plataforma aberta para experimentos e pesquisa. Assim, embora o módulo de escalonamento implementado possua muitas funcionalidades, os algoritmos que ele usa são ainda bastante simples, no entanto, as funcionalidades mais importantes, tais como, prioridades sobre tarefas, reserva de recursos e *backfilling* são implementados.

O algoritmo padrão implementado neste escalonador consiste na política *First Come First Served* (FCFS), assim, todas as aplicações são ordenadas de acordo com o seu tempo de chegada na fila. Políticas de escalonamentos podem ser implementados, com outras linguagens de programação.

O monitoramento de tarefas é tratado por uma ferramenta separada chamada Taktuk. A ferramenta Taktuk permite a execução paralela de comandos em grandes aglomerados. Este programa é altamente paralelizado e distribuído.

Dentro do escalonador, a ferramenta Taktuk é utilizada para realizar tarefas administrativas de cada um dos nós de um aglomerado, por meio de um serviço de execução remota. Através desta ferramenta, nós falhos são detectados pelo tempo de resposta dos mesmos, respeitando-se um tempo limite que pode ser modificado pelo administrador do escalonador. Porém, apesar da sua versabilidade, o Taktuk não faz análise de padrões de uso dos recursos.

O escalonador OAR atualmente é o responsável pelo gerenciamento de recursos e agendamento de tarefas do projeto Grid5000[2] e do projeto CIMENT[3], realizando principalmente três tarefas:

- Reserva nós para um determinado período, em nome de um usuário solicitante;
- Agenda as tarefas dos usuários sobre os nós reservados, garantindo um tempo de utilização coerente;
- Libera recursos no final de cada reserva.

3.2 O escalonador Condor

O sistema Condor [LLM88, FTF+02, TTL05] é um software especializado para gerenciar aplicações de computação intensiva. Como outros escalonadores de tarefas (*batch systems*), o projeto Condor fornece mecanismos de enfileiramento e priorização de tarefas, políticas de escalonamento e monitoração de recursos. O projeto Condor é software livre e possui licença Apache versão 2.0.

O projeto Condor é um dos sistemas pioneiros na área da computação distribuída. Lançado em 1984 e desenvolvido pela equipe do projeto Condor na universidade de *Wisconsin-Madison*. Este projeto influenciou o interesse acadêmico na área de computação em alto desempenho para buscar soluções que permitam um uso de ciclos ociosos de recursos computacionais.

O fluxo no escalonador Condor começa com os usuários finais, os quais, submetem aplicações paralelas ou seriais, depois disso as aplicações são enfileiradas pelo software que baseado em uma política escolhe quando e onde vai executar os trabalhos. Além disso o software monitora cuidadosamente o progresso da execução da aplicação e quando a execução é finalizada informa ao usuário final os resultados.

3.2.1 Arquitetura

O escalonador Condor pode ser usado como gerenciador de um aglomerado com nós computacionais dedicados ou não dedicados (por exemplo, um aglomerado *Beowulf*[4]). Mecanismos próprios do Condor permitem aproveitar o poder computacional ocioso dos recursos computacionais.

Na Figura 3.2 é mostrado um conjunto de recursos que representam uma infraestrutura Condor formada por um nó chamado gerenciador central e um conjunto de nós divididos entre: (i) nós de execução e (ii) nós submissores; esse conjunto de recursos é chamado *pool*. Cada tipo de nó está composto por um conjunto de programas implementados através de *daemons*. Cada componente é aprofundado a seguir:

- Gerenciador Central (*Central Manager*). Em um *pool* no Condor existe somente um gerenciador central responsável por recolher informações como as características e o uso dos recursos computacionais pertencentes ao *pool*, este mecanismo é feito através do *daemon* `condor_collector`. Baseado na informação coletada, o gerenciador central pode fazer o casamento entre as tarefas e os recursos que as satisfazem, através do *daemon* `condor_negotiator`.

[2] http://www.grid5000.fr/, último acesso em 06 de junho de 2011.
[3] http://ciment.ujf-grenoble.fr/, último acesso em 06 de junho de 2011.
[4] Um aglomerado é chamado de *Beowulf* quando é feito usando computadores pessoais, não especializados, porém, mais baratos.

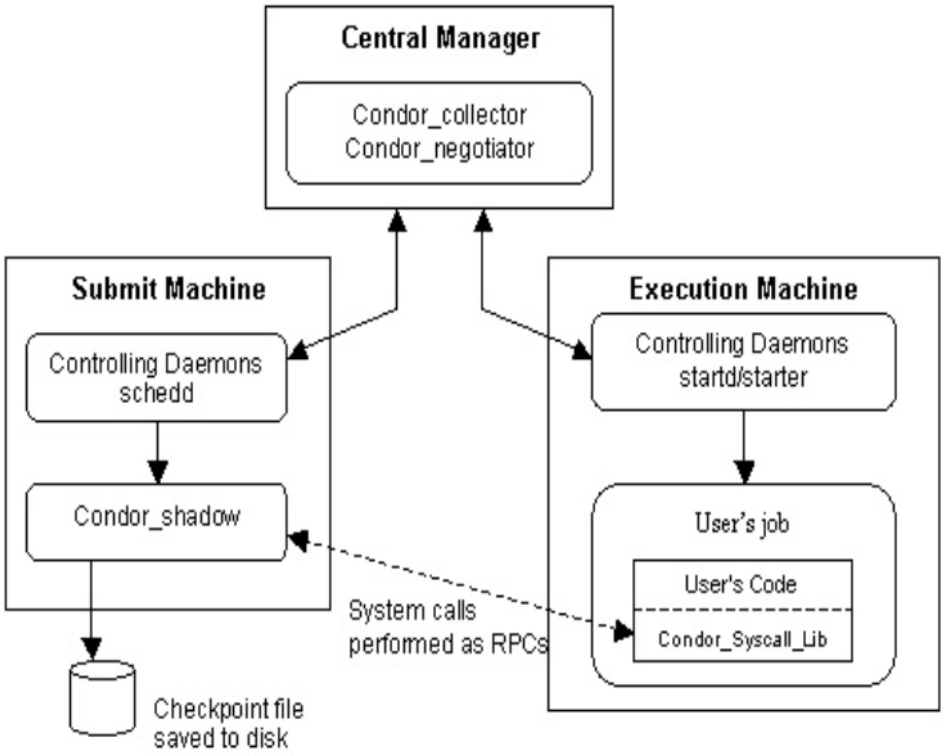


Figura 3.2: *Visão geral da arquitetura no Condor [CWT+04].*

- Nós submissores (*Submit Machine*). Através do *daemon* `schedd`, este tipo de nó permite ao usuário submeter tarefas e inseri-las em uma fila. Esse *daemon* solicitará escalonamento de suas tarefas nos recursos ao gerenciador central durante um ciclo de negociação. Depois de acontecido o escalonamento, o *daemon* `schedd` executa o *daemon* `shadow` responsável para gerenciar a execução remota das tarefas, também controlar o estado de cada tarefa para efeitos de *checkpointing*, em caso de falhar alguma tarefa será feito um reescalonamento.

- Nós de execução (*Execution Machine*). O nó de execução representado pelo *daemon* `startd`, executa tarefas em nome do usuário. Através deste *daemon* o gerenciador central conhece das informações do recurso computacional e as especificações no uso dele.

3.2.2 Escalonamento

Uma importante característica no projeto Condor é o seu escalonamento. Este mecanismo é denominado *matchmaking* [RLS98], no qual é decidido quando, onde e como será executada uma determinada tarefa. Na Figura 3.3 é apresentado o fluxo de como funciona esse mecanismo.

Quando uma tarefa precisa ser executada, um componente denominado "agente" anuncia sua existência, especificando os requerimentos computacionais necessário (*Advertisement*, passo 1). De forma similar quando um recurso é adicionado ao *pool* este recurso anuncia sua existência, especificando todas as suas características (*Advertisement*, passo 1).Essa atividade é chamada de *Classified advertisements* ou *ClassAds*. Este anunciamento é feito às entidades de *matchmaker*, uma vez anunciados, o *matchmaker* encontra qual recurso é adequado para executar determinada tarefa (*Matchmaking Algorithm*, passo 2). Depois desse processo tanto à tarefa quanto o recurso elegido serão notificados (*Notification*, passo 3). Finalmente, no passo 4, essas partes negociam possíveis

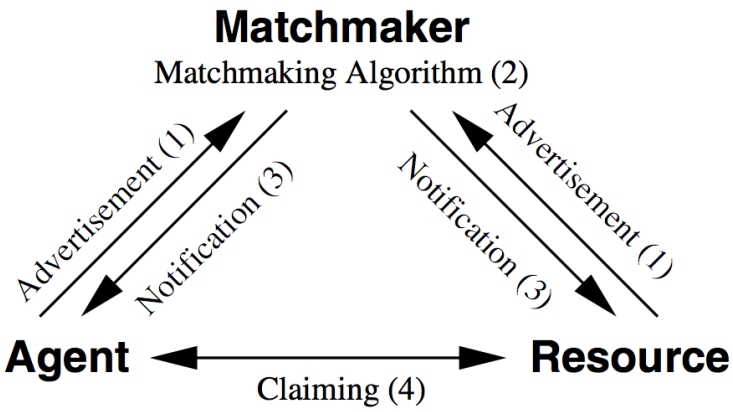


Figura 3.3: *Mecanismo Matchmaking [TTL05] no Condor.*

termos e começam a execução.

O mecanismo de *classAds* é feito especificando uma linguagem própria do Condor. Esta linguagem é da forma $nome = valor$, um exemplo é apresentado na Figura 3.4.

Job ClassAd	Machine ClassAd
[	[
MyType = "Job"	**MyType** = "Machine"
TargetType = "Machine"	**TargetType** = "Job"
Requirements =	**Machine** = "nostos.cs.wisc.edu"
((other.Arch=="INTEL" &&	**Requirements** =
other.OpSys=="LINUX")	(LoadAvg <= 0.300000) &&
&& other.Disk > my.DiskUsage)	(KeyboardIdle > (15 * 60))
Rank = (Memory * 10000) + KFlops	**Rank** = other.Department==self.Department
Crod = "/home/tannenba/bin/sim-exe"	**Arch** = "INTEL"
Department = "CompSci"	**OpSys** = "LINUX"
Owner = "tannenba"	**Disk** = 3076076
DiskUsage = 6000	]
]	

Figura 3.4: *Dois exemplos do mecanismo de* Classified Advertisements *do escalonador Condor [TTL05].*

3.3 O escalonador PBS/OpenPBS/Torque

O escalonador *Portable Batch Scheduler* (PBS)[5] [Hen95] é um sistema de processamento de filas que segue o padrão POSIX[6] Entre as principais características o projeto PBS foi feito para monitorar e gerenciar de recursos e tarefas sobre um conjunto de um ou mais computadores. Originalmente desenvolvido para gerenciar recursos de computação aeroespacial da NASA. O escalonador desde então se tornou o líder na gestão da carga em supercomputadores e o padrão para gerenciar tarefas no Linux [WET03].

O projeto PBS é configurável sobre grande quantidade de recursos computacionais, aglomerados heterogêneos, supercomputadores massivamente paralelos, entre outros.

Atualmente o projeto foi adquirido pela empresa chamada *Altair Engineering*, que atualmente distribui duas versões do escalonador PBS: (i) *PBS Professional*, versão comercial e (ii) *OpenPBS*: distribuição livre, porém não mantida pela empresa.

[5] Altair Engineering, http://www.pbsgridworks.com, último acesso em 06 de junho de 2011.
[6] Portable Operating System Interfaces for UniX.

3.3.1 Arquitetura

A arquitetura do escalonador PBS segue a ideia de ter um servidor que gerencia aos nós pertencentes ao aglomerado, como é mostrado na Figura 3.5.

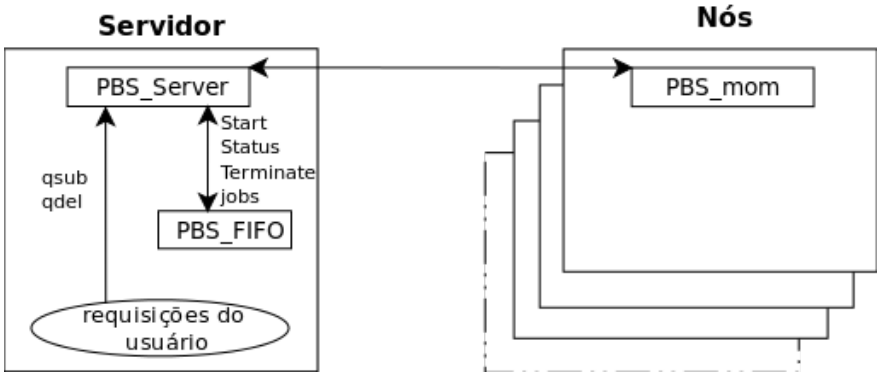


Figura 3.5: *Componentes do PBS, esquema adaptado de [BHK+00].*

No servidor acontecem as principais funcionalidades do escalonador, entre as quais, temos:

- Enfileiramento: é realizada a coleta de tarefas para serem executadas nos recursos disponíveis. Os usuários enviam as tarefas para o sistema de gerenciamento de recursos onde eles são colocados em fila até que o sistema esteja pronto para executá-los;
- Escalonamento: determina quando e onde, e seleciona as tarefas para executá-las, de acordo com uma política específica. O escalonador fornece diversas alternativas para a distribuição da carga de tarefas entre os recursos disponíveis, baseados na disponibilidade de recursos ou mesmo na utilização de dispositivos de entrada, com o objetivo de maximizar o uso eficiente de recursos;
- Monitoramento: é o processo de rastreamento e reserva dos recursos. Isso abrange o monitoramento tanto a nível de usuário e de nível de sistema, bem como a monitorização de tarefas em execução.

A partir da versão livre do escalonador PBS, foi desenvolvido um derivado feito a partir do OpenPBS, chamado Torque [Sta06], este escalonador e é ativamente desenvolvido, suportado e mantido pela empresa *Cluster Resources Inc.*

3.3.2 O escalonador Torque

O escalonaodor *Terascale Open-Source Resource and QUEue Manager* Torque [Sta06] é um gerenciador de recursos de código aberto. Foi desenvolvido a partir da versão livre do PBS (*OpenPBS, v2.3.12*). O escalonador *Torque* modificou grande parte do código do OpenPBS melhorando e adicionando importantes funcionalidades, como por exemplo: na escalabilidade, tolerância a falhas e extensões das características básicas. O projeto Torque recebeu as contribuições de importantes organizações, tais como: *National Center for Super-computing Applications* (NCSA), *Ohio Supercomputer Center* (OSC), *University of Southern California* (USC), *U.S. Dept. of Energy*, *U. de Buffalo*, TeraGrid, e muitas outras organizações líderes no *High Performance Computing* (HPC).

O escalonador Torque é software livre, usado em milhões de sítios de pesquisa no globo. Com os comandos disponibilizados é possível alocar recursos, escalonar, gerenciar a execução e monitorar o estado das tarefas submetidas.

Algumas características inseridas ao OpenPBS pelo Torque são:

- **Tolerância a Falhas**: condições de falha são verificadas e tratadas. Fornece um *script* para verificar periodicamente cada recurso.
- **Interface de Escalonamento**:
  - Extensão da interface de consulta: este mecanismo proporciona ao escalonador informações adicionais mais precisas;
  - Extensão da interface de controle: permite ao escalonador um maior controle sobre o comportamento das tarefas e seus atributos;
  - Permite recolher estatísticas das tarefas concluídas.
- **Escalabilidade**:
  - Melhora significativa no servidor. Implementação do modelo de comunicação *Machine-Oriented Miniserver* (MOM) [Hen95];
  - Habilidade para lidar com aglomerados maiores (mais de 15 TF/2.500 processadores);
  - Habilidade para lidar com tarefas maiores (mais de 2.000 processos);
  - Capacidade para suportar mensagens maiores no servidor.
- **Usabilidade**: adição de novas funcionalidades de *logging*.

Arquitetura

Um aglomerado baseado no *Torque* consiste de um nó principal e muitos outros nós secundários. O nó principal executa o *daemon* `pbs_server` e os outros nós executam o *daemon* `pbs_mom`. Os comandos clientes para submeter e gerenciar tarefas são instalados em qualquer nó, incluindo nós onde não foram executados os *daemons* `pbs_server` ou `pbs_mom`. Esta arquitetura segue a ideia da Figura 3.5.

Escalonamento

O nó principal também executa um *daemon* para o escalonador. O escalonador interage com o `pbs_server` para tomar decisões de política local para o uso dos recursos e assim alocar tarefas aos nós. Um simples escalonador de tipo FIFO, e um código para construir um escalonador mais avançado é fornecido na distribuição fonte do *Torque*. A maioria dos usuários *Torque* optam por usar um pacote, avançado de escalonador, como Maui ou Moab.

Os usuários submetem tarefas ao `pbs_server` usando o comando `qsub`. Quando `pbs_server` recebe uma nova tarefa informa ao escalonador. Quando o escalonador encontra os nós para a tarefa, envia instruções para executar a tarefa com a lista de nós ao `pbs_server`. Então, o `pbs_server` envia a nova tarefa para o primeiro nó na lista de nós com instruções para iniciar a tarefa.

Maui

O Maui [BHK+00] é um escalonador de tarefas de código livre para aglomerados e supercomputadores. Ele surgiu com o propósito de auxiliar algumas carências de desempenho das políticas de escalonamento implementadas no sistema *IBM LoadLeveler*, por exemplo, a grande taxa de ociosidade de recursos paralelos na espera de tarefas.

O objetivo principal no Maui é escalonar as tarefas de forma especializada. Ele pode ser usado como um componente adicional em sistemas de escalonamentos, por exemplo, o PBS e o Torque. O Maui estende as capacidades desses sistemas, acrescentando as seguintes características:

- **Priorização de tarefas**: o Maui atribui pesos aos diferentes objetivos, assim, um valor global ou prioridade pode ser associado no escalonamento;
- **Reserva de recursos**: cada reserva é constituída por três componentes principais: a lista de recursos, o tempo de reserva e a lista de controle de acesso. A tecnologia de reserva antecipada fornece muitas características ao Maui, incluindo *backfill*, escalonamento baseado em prazos, suporte a QoS (*Quality of Service*) e metaescalonamento;
- **Suporte QoS**: o soporte para QoS é configurado pelos administradores. Cada tipo de QoS apresenta um conjunto de prioridades, políticas de isenções e configurações de acesso aos recursos. Os administradores podem configurar usuários, grupos, contas e classes como beneficiários dos privilégios de QoS;
- **Abrangente equidade de políticas**: o Maui oferece algumas ferramentas flexíveis que ajudam nas necessidades comuns de equidade, isto implica igualdade de acesso aos recursos computacionais para todos os usuários;
- **Política de Backfill**: é uma otimização de escalonamento que permite que um escalonador faça melhor uso dos recursos disponíveis, executando trabalhos fora da ordem estabelecida em determinada fila. As tarefas com maior prioridade e com requisições simples de recursos serão executadas antes que as tarefas no topo da lista;
- **Suporte de diagnóstico**: o Maui fornece um número de comandos para o diagnóstico do comportamento do sistema. Esses comandos de diagnóstico apresentam detalhadamente aspecto sobre problema no escalonamento, relatório do desempenho e sobre condições inesperadas ou potencialmente erradas de qualquer operação em curso;
- **Modo de teste**: o Maui possui um modo de escalonamento, chamado *test*. Nesse modo, o escalonador funciona como se estivesse executando normalmente, mas o Maui torna-se incapaz de iniciar, interromper, cancelar ou modificar atributos das aplicações ou dos recursos.

3.4 Considerações Finais do Capítulo

Neste capítulo foram apresentados os principais escalonadores usados em aglomerados em funcionamento: o Condor, o OAR, o PBS e o Torque. De forma geral os escalonadores para aglomerados usam técnicas de escalonamento relativamente simples, usando o mecanismo de enfileiramento. Cada

tarefa é escalonada respeitando a ordem de chegada e os requisitos necessários para seu processamento.

No caso do escalonador OAR, o escalonamento é feito usando informações da tarefa e dos recursos armazenadas em um banco de dados no servidor, procurando quais recursos satisfazem as requisições das tarefas para a sua execução. De forma similar no sistema Condor, usando os mecanismos de *ClassAds* e *Matchmaking*, é feito um casamento entre as tarefas e os recursos. Por outro lado o sistema PBS oferece mecanismos básicos no escalonamento, enquanto que o sistema Torque, melhora esses mecanismos, no entanto, continua com o sistemas de filas.

O mecanismo de enfileiramento é alternativo em cada escalonador, deixando aberta a possibilidade de implementar mecanismos especializados para escalonar as tarefas.

Capítulo 4

Middlewares para Grades

Atualmente, existem diferentes soluções para implantar uma estrutura de computação em grade. Estas soluções são feitas mediante um *middleware*. Esses softwares possuem diferentes características em diversos níveis, por exemplo, segurança, monitoramento, escalonamento entre outros.

Neste capítulo são descritos quatro *middlewares* para grades, principalmente é mostrado como os *middlewares* fazem escalonamento de tarefas, com o intuito de ter uma ideia de como é o escalonamento em soluções de software já existentes na área da computação em grade.

4.1 Boinc

O sistema *Berkeley Open Infrastructure for Network Computing* (Boinc) [And04, AKW05] é um *middleware* de código aberto para computação voluntária (*Volunteer Computing*) e computação em grade. O principal objetivo do projeto é motivar usuários do mundo à disponibilização do seus recursos computacionais em tempos ociosos para projetos de pesquisa.

Diferentes projetos que precisam de grande poder computacional para o processamento dos seus dados usam o sistema Boinc de maneira independente. Cada projeto trabalha sobre seu próprio servidor e banco de dados, no entanto, esses projetos obtêm esse poder computacional pelo compartilhamento de recursos voluntários ao redor do mundo. Neste cenário, entre os projetos mais representativos que usam Boinc, temos:

- SETI@home;
- distributed.net;
- Climateprediction.net;
- LCH@home.

Pessoas participantes executam o programa cliente do sistema Boinc nos seus computadores e agregam os projetos que atualmente se encontram disponíveis, para doar uma fração dos seus recursos computacionais.

4.1.1 Arquitetura

Os projetos baseados no Boinc são autônomos. Cada projeto é operado por um servidor que é constituído por diversos componentes:

- Interfaces web;
- Um servidor de tarefas;
- Um servidor de dados.

Como é apresentado na Figura 4.1 estes componentes compartilham dados armazenados em disco, banco de dados, envio e descarrega de arquivos.

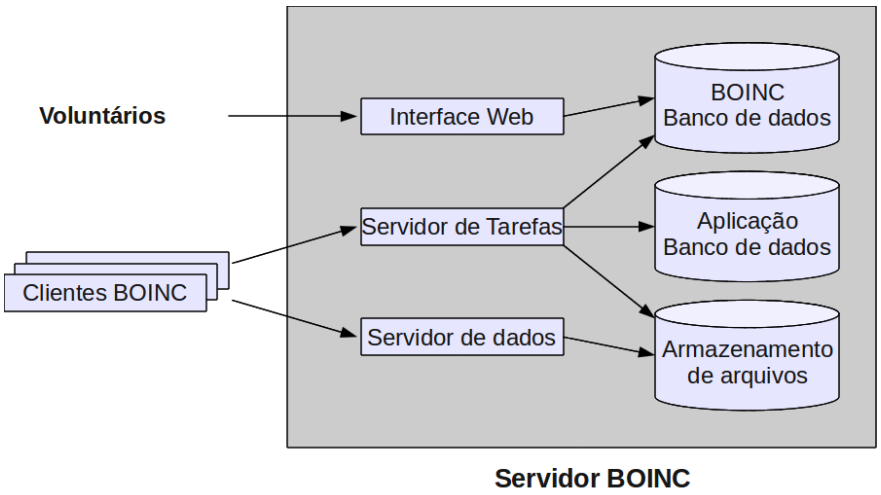


Figura 4.1: *Componente de um servidor Boinc. Esquema adaptado de [AKW05].*

4.1.2 Escalonamento

Os tipos de aplicações que o sistema Boinc processa são aplicações altamente paralelizáveis ou aplicações que possuem tarefas sem dependência (BoT).

Toda comunicação no Boinc é iniciada pelo cliente, o qual se comunica com o servidor via protocolo *HTTP*, enviando uma descrição das características do hardware e disponibilidade, uma lista das tarefas completadas e requisição por uma nova tarefa.

O escalonamento neste *middleware* [And07] consiste em duas partes: a "seleção de tarefas", na qual é criada uma fila de tarefas, e a "execução de tarefas", aquela que é responsável pela execução de cada tarefa localizada na fila. A tarefa que será executada será aquela que possui uma *deadline* ou instante de tempo no qual a tarefa deve estar pronta, mais próxima.

Este *middleware* possui mecanismos de *checkpointing*, com o intuito de salvar estados da execução da tarefa, para que possam ser retomados novamente, após um eventual desligamento da máquina voluntária.

4.2 InteGrade

O projeto InteGrade[1] [GKG+04] é um software desenvolvido em parceria entre várias universidades brasileiras, cujo alvo foi construir um *middleware* para grades orientado a objetos, focado especificamente em computação oportunista e voluntária.

O objetivo do projeto InteGrade é fornecer uma ferramenta para permitir que organizações façam uso dos seus recursos computacionais ociosos para obter um maior poder de processamento.

[1] http://www.integrade.org.br/, último acesso em 07 de junho de 2011.

4.2.1 Arquitetura

A arquitetura deste projeto é baseada em agrupamentos. Esses grupos podem ser compostos de um computador pessoal até centenas de computadores. Os agrupamentos são organizados em uma hierarquia, permitindo uma estrutura de grade simples, para abranger potencialmente milhões de máquinas.

Na Figura 4.2 são apresentados os componentes de uma grade usando o InteGrade. O nó "gerenciador do aglomerado" (*Cluster Manager*) representa o nó responsável pelo gerenciamento do aglomerado a que pertence e pela comunicação entre outros gerenciadores de outros aglomerados. O "nó usuário" (*User Node*) é o nó onde o usuário da grade submete aplicações à grade. O "nó fornecedor de recurso" é o nó que doa os recursos computacionais à grade. O "nó dedicado" é o nó voluntário que oferece os seus recursos computacionais especificamente à grade.

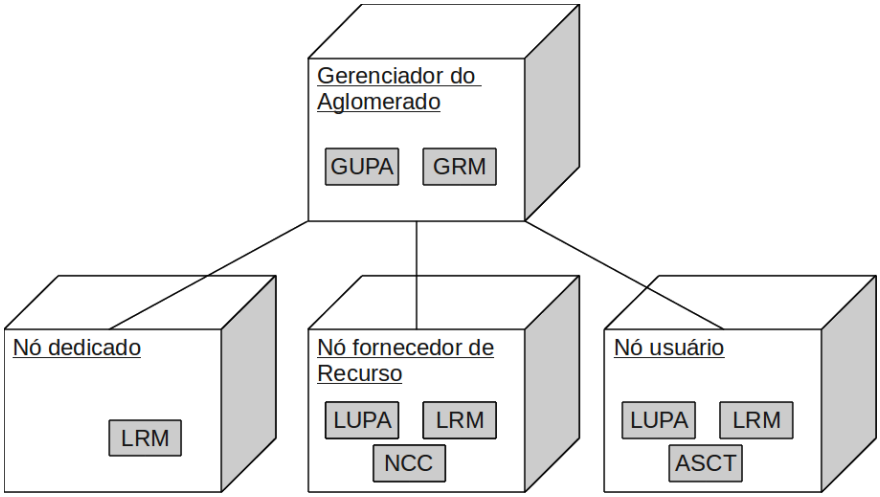


Figura 4.2: *Arquitetura do InteGrade. Esquema adaptado de [GKG+04].*

4.2.2 Escalonamento

O gerenciador local do recurso (LRM, *Local Resource Manager*) e o gerenciador global do recurso (GRM, *Global Resource Manager*) trabalham de forma cooperativa. O LRM é executado em cada nó aglomerado, coletando informações sobre o estado do nó (memória, CPU, espaço em disco, uso da rede). LRMs enviam a informação coletada ao GRM, e este usa essa informação para fazer o escalonamento dentro do aglomerado.

O analisador de padrões de uso local (LUPA, Local Usage Pattern Analyzer) e o analisador de padrões de uso global (GUPA, Global Usage Pattern Analyzer) fazem uma análise do uso mediante padrões entre os aglomerados. O LUPA coleta dados sobre padrões de uso do nó, estes dados são enviados ao GUPA. Essa informação é usada pelo GRM para melhorar mecanismos de escalonamento de tarefas.

O nó central de controle (NCC, *Node Control Center*) permite aos usuários setar restrições no uso dos seus recursos computacionais. A ferramenta de controle e submissões de aplicações (ASCT, *Application Submission and Control Tool*) permite aos usuários da grade submeter aplicações para estas serem executadas. Os usuários especificam requisitos necessários para a execução da sua aplicação.

Quando um usuário submete uma aplicação para executar, isso é feito mediante o ASCT. O GRM seleciona os nós candidatos para a execução, baseado na disponibilidade dos recursos e necessidades da aplicação. Então o GRM reenvia as requisições aos LRM, os quais verificam se possuem recursos computacionais disponíveis nesse momento. Se não existir recurso computacional disponível o GRM procura por outro. Este processo é repetido até encontrar algum LRM que aceite executar a aplicação. Quando se encontra um recurso disponível, o LRM solicita a aplicação e os arquivos necessários ao ASCT. A aplicação é executada no recurso computacional selecionado e uma notificação de execução é enviada ao ASCT. Finalmente o usuário pode ver os resultados pelo ASCT.

4.3 OurGrid

O *middleware* OurGrid[CBA+06] é um projeto de software livre com licença GPL para computação em grade. Permite a criação de uma grade *peer-to-peer free-to-join*. A primeira versão foi liberada em dezembro de 2004 e foi usada por centenas de usuários para acelerar a execução de aplicações *BoT* (Saco de Tarefas), ou seja, aplicações paralelas cujas tarefas são independentes. Este tipo de aplicação é executada de forma paralela na grade, porém não há comunicação entre as tarefas.

O OurGrid tem uma comunidade ativa de usuários e desenvolvedores. O software é escrito na linguagem de programação *Java*, permitindo que qualquer recurso capaz de executar uma máquina virtual *Java* possa ser aproveitado pela grade.

4.3.1 Arquitetura

Como é mostrado na Figura 4.3 o *middleware* OurGrid possui principalmente três componentes:

- *The MyGrid broker*;
- *The OurGrid peer*;
- *The SWAN security service.*

Através do componente MyGrid broker [CPC+03], o usuário consegue executar suas aplicações em todos os recursos que tenha acesso. O MyGrid envia uma requisição para o componente *OurGrid Peer*, o qual tentará obter recursos nos diferentes *peers* (ou nós) da grade. Desta forma, é possível a cooperação dos diferentes *peers* da grade. Neste processo, os usuários locais sempre têm prioridade maior que os outros usuários da grade. Se não houver nenhuma requisição de usuários locais, os recursos disponíveis são considerados ociosos. Então, um esquema chamado *Network of Favors*[2] [CBA+06] é utilizado,

Network of Favors é um esquema de alocação de recursos baseado em reputação. *Peers* que são mais usados têm uma melhor reputação e, desta forma, recebem uma prioridade maior no momento que requisitam recursos de outros *peers*. Esta abordagem evita o fenômeno de *free-riding*[3], na qual

[2]Neste esquema, os *peers* que oferecem maior uso dos seus recursos computacionais possuem maior preferência na execução de tarefas.

[3]Este fenômeno acontece quando um *peer* somente consome recursos.

os *peers* não doam recurso nenhum, e somente estão presentes na grade para usar os recursos computacionais.

O componente *Sandboxing Without A Name* (SWAN) é uma solução de segurança do middleware OurGrid, baseada em máquinas virtuais Xen[4] que isola o código desconhecido em um *sandbox*. Desta forma, as tarefas da grade que executam em uma máquina específica não podem danificá-la ou utilizar a rede de maneira indevida.

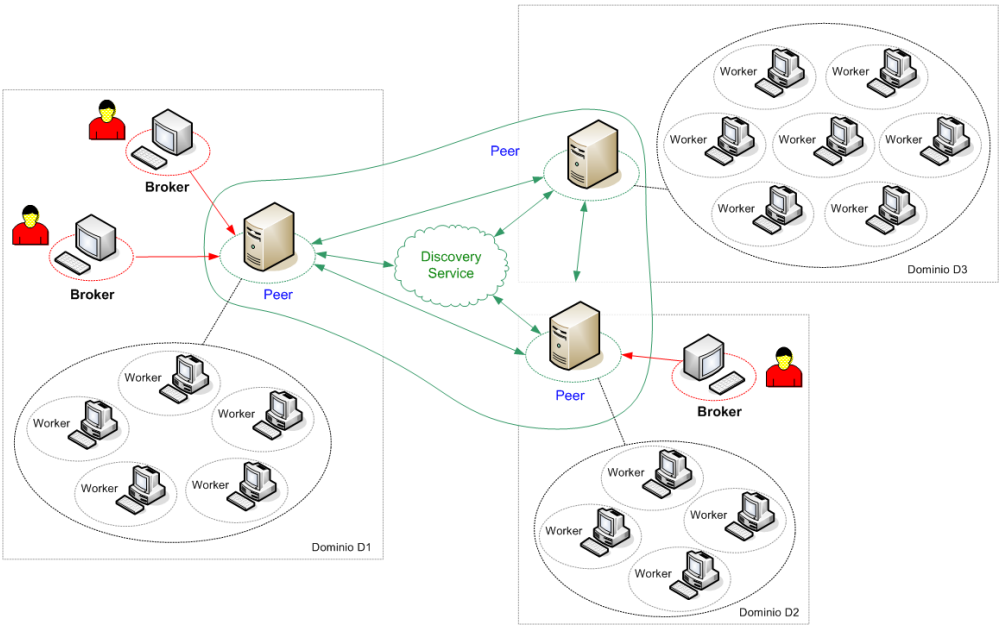


Figura 4.3: *Arquitetura do* middleware *OurGrid [CBA+06].*

4.3.2 Escalonamento

Ainda com a simplicidade das aplicações *BoTs*, o escalonamento delas sobre grades é difícil. Primeiro, escalonamentos eficientes dependem de informações sobre a aplicação (por exemplo, tempo de execução estimado, tamanho das tarefas) e recursos (velocidade de processamento, topologia da rede, carga dos recursos entre outros). Mas, é muito complexo obter este tipo de informações num sistema tão grande e amplamente disperso como uma grade. Assim, o MyGrid, escalonador do *middleware* OurGrid, usa algoritmos de escalonamento que não dependem desse tipo de informações:

- *Workqueue*;
- *Workqueue with Replication*[dSCB03];
- *Storage Affinity*[SNCBL04].

A heurística *Workqueue* simplesmente escalona as tarefas submetidas aos recursos disponíveis em uma ordem arbitrária. Enquanto a heurística *WorkQueue with Replication* (WQR) é uma extensão do *Workqueue*, sendo que, após o escalonamento de todas as tarefas, o escalonador passa a escalonar réplicas das tarefas em execução até que não haja mais recursos disponíveis. Como a estratégia de *Workqueue* não utiliza qualquer informação acerca das aplicações ou dos recursos, a replicação funciona como um mecanismo que procura compensar alocações mal sucedidas (por exemplo, escalonar

[4] http://www.xen.org/, último acesso em 08 de junho de 2011.

tarefas em recursos lentos ou sobrecarregados). Isso faz com que o WQR consuma mais recursos do que os escalonadores que utilizam informações sobre a disponibilidade dos recursos. Cientes deste problema, os desenvolvedores lançaram a segunda versão do MyGrid, com uma nova estratégia de escalonamento: a heurística *Storage Affinity*. Esse algoritmo de escalonamento mantém informações sobre a quantidade de dados que os nós contêm sobre uma determinada aplicação. Dessa forma, sempre que uma decisão de escalonamento precisa ser feita, o *Storage Affinity* escolhe o recurso que já contém a maior quantidade de dados necessários para o processamento. Essa abordagem é mais adequada para as aplicações (*BoT*) que processam grandes quantidades de dados, já que o tempo de transferência dos dados para as máquinas que irão processá-los representam uma sobrecarga considerável no tempo total de execução das aplicações.

4.4 XtremWeb

O projeto XtremWeb [CDF+05, FGNC01] é um *middleware* de código aberto para computação em grade largamente usado em pesquisa em sistemas distribuídos em larga escala (*large-scale distributed system*), e para resolver atuais questões sobre computação em grade e sistemas *Peer to peer*.

Similar a outros *middlewares* para computação em grade, o XtremWeb é baseado na filosofia da computação voluntária, tomando recursos remotos ociosos de computadores pessoais de terceiros (CPU, espaço para armazenamento).

Entre as principais características do XtremWeb estão permitir múltiplos usuários, múltiplas aplicações e a execução de aplicações entre diferentes plataformas.

4.4.1 Arquitetura

O XtremWeb oferece a possibilidade de trabalhar em diferentes plataformas, tais como: *Linux*, *Windows* e *Mac OS*; assim, máquinas ao redor do mundo podem participar da grade.

A arquitetura do XtremWeb possui três componentes:

- O coordenador (*Coordinator*);
- O cliente (*Client*);
- O trabalhador (*Worker*).

O cliente é o componente interface entre os usuários e o sistema. Através do cliente os usuários submetem tarefas no coordenador. Os nós trabalhadores são responsáveis pela execução das tarefas, estes requisitam ao coordenador tarefas para processar; em resposta, o coordenador envia um conjunto de parâmetros e também, caso necessário, a aplicação a ser executada. Quando um trabalhador termina de executar uma tarefa, ele envia o resultado ao coordenador.

4.4.2 Escalonamento

O tipo de aplicação que o *middleware* XtremWeb executa é a aplicação onde as tarefas são independentes, isto é, aplicação *BoT*. O XtremWeb possui um mecanismo de escalonamento básico *First Come First Served* (FCFS).

Quando um trabalhador requisita uma tarefa para processar, tanto o coordenador quanto o trabalhador coordenam se esse trabalhador possui todas as características necessárias para a execução da tarefa com sucesso. Depois da coordenação, o coordenador envia a tarefa ou conjunto de tarefas adequadas para o trabalhador. O escalonador aborta a execução da tarefa se o tempo máximo permitido foi esgotado, reescalonando a tarefa se for necessário.

4.5 Considerações Finais do Capítulo

Neste capítulo foram apresentados quatro *middlewares* atualmente usados em computação em grade. Dentre esses *middlewares*, somente o *middleware* OurGrid usa estratégias especializadas para o escalonamento, as quais são: (a) a heurística *Workqueue*, (b) a heurística *Workqueue with Replication* e (c) a heurística *Storage Affinity*, detalhados brevemente no Capítulo 8. Estas soluções são específicas para o escalonamento de aplicações que possuem tarefas independentes, ou também chamadas de Bag-of-Tasks.

Por outro lado os outros três *middlewares* não usam estratégias especializadas para o escalonamento. No caso do *InteGrade*, este *middleware* quando recebe uma aplicação, usa uma estratégia de tipo FIFO, na qual, aloca as tarefas a medida que elas chegam, avaliando se o recurso na qual será alocada a tarefa satisfaz os requerimentos da mesma. Os *middlewares* XtremWeb e Boinc usam um mecanismo similar. No caso do XtremWeb, este *middleware* aloca as tarefas fazendo previamente uma coordenação entre o coordenador (server) e os trabalhadores (workers). No caso do *middleware* Boinc é criada uma fila de tarefas onde a execução de cada tarefa é feita baseada em uma data limite (*deadline*).

Capítulo 5

Arquiteturas para Grades

Existem diversos tipos de plataformas feitas para o uso exclusivo da computação em grade. Diferentes lugares no mundo possuem plataformas exclusivas para estas arquiteturas com características já definidas, formadas por múltiplos nós compostos por aglomerados, computadores pessoais, servidores, supercomputadores entre outros. Cada plataforma possui nós homogêneos ou heterogêneos, tanto em hardware quanto em software, e a quantidade numérica de processadores de cada plataforma varia entre 200 até 5.000 processadores.

Para interesses de pesquisa, existem plataformas com poucas restrições no uso. Estas plataformas geralmente são usadas para pesquisas em comunicação de softwares, computação paralela, escalonamento, aplicações paralelas e aplicações distribuídas. Atualmente, essas plataformas estão disponíveis para simulação. Este grande aporte foi dado pelo projeto *SimGrid*, especificamente na parte das contribuições do projeto[1], desta forma, é possível executar simulações com essas plataformas [FQS08].

5.1 O Projeto DAS-3

O projeto *Distributed ASCI Supercomputer 3* (DAS-3)[2] é uma arquitetura composta por cinco aglomerados heterogêneos geograficamente distribuídos pela Holanda, desenvolvido pela *Advanced School for Computing and Imaging* (ASCI)[3]. Inicialmente, o projeto foi chamado DAS [BBH+00], composto por quatro aglomerados homogêneos, distribuídos em diferentes regiões e projetado para pesquisas em aplicações paralelas e distribuídas.

O projeto DAS-3 é financiado pela fundação *The Netherlands Organization for Scientific Research* (NWO/NCF), pelo projeto *the Virtual Laboratory for e-Science* (VL-e), por universidades e por organizações.

O objetivo do projeto é fornecer aos pesquisadores da ASCI uma infraestrutura com grande poder computacional, principalmente focado em temas de pesquisa que abrangem computação paralela e distribuída, computação em grade e análise de conteúdo multimídia em larga escala. As instituições que estão diretamente envolvidas no projeto são as seguintes:

- Universidade de Vrije, Amsterdã (VU);

[1] http://simgrid.gforge.inria.fr/doc/contrib.html, último acesso em 08 de junho de 2011.
[2] http://www.cs.vu.nl/das3/, último acesso em 08 de junho de 2011.
[3] http://www.asci.tudelft.nl/, último acesso em 08 de junho de 2011.

- Universidade de Leiden (LU);
- Universidade de Amsterdã (UvA);
- Universidade Tecnológica Delft (TUD);
- The MultimediaN Consortium (UvA-MN).

5.1.1 Arquitetura

O projeto DAS-3 possui uma interconexão entre os nós baseada em fibra ótica, alcançando uma grande velocidade na interconexão. É composto por 272 processadores *dual AMD Opteron*, espalhados sobre os cinco aglomerados localizados nas quatro universidades.

Em comparação com versões anteriores, os projetos DAS e o DAS-2, o projeto DAS-3 possui nós heterogêneos, a interconexão entre cada nó usa uma tecnologia de rede chamada *Myri-10G*[4]. A tecnologia *Myri-10G* oferece interconexões de alta velocidade e com baixa latência.

Atualmente, já existe uma quarta geração do projeto, chamada DAS-4[5], anunciada em dezembro de 2010.

Na Figura 5.1 é apresentado um esquema da arquitetura do projeto DAS-3, na qual temos os cinco aglomerados. Este esquema é uma adaptação da especificação gerada no projeto *The Platform Description Archive* (PDA)[6].

Na Tabela 5.1 são apresentadas as quantidades dos números de processadores por cada nó, poder de processamento e largura de banda, por outro lado, a velocidade de conexão entre cada nó em média é $1,00E+011b/s$, essa alta velocidade é produto do uso da tecnologia *Myri-10G* na interconexão entre os nós desta grade. Estas informações foram usadas para o modelamento desta arquitetura.

prefixo	**sufixo**	**nós**	**Poder Comp.** ($GFlops/s$)	**Largura de Banda** (Gb/s)
vrije-universiteit-	.das3.nl	85	$2,40$	$1,25$
leiden-university-	.das3.nl	32	$2,60$	$1,25$
University-of-amsterdam-	.das3.nl	41	$2,20$	$1,25$
Delpft-university-of-technology-	.das3.nl	68	$2,40$	$1,25$
Multimedia-consortium-	.das3.nl	46	$2,40$	$1,25$

Tabela 5.1: *Características da arquitetura DAS-3. O total de processadores é* 272*. A média do poder computacional é* $2,40GFlops/s$*.*

5.2 O Projeto Grid5000

O projeto Grid5000[7] [BCC+06, CCD+05] é uma arquitetura científica criada para o estudo de sistemas paralelos e distribuídos de larga escala. O objetivo do projeto é fornecer uma plataforma experimental altamente monitorável, controlável e reconfigurável para seus usuários. O objetivo

[4] http://www.myri.com/, último acesso em 08 de junho de 2011.
[5] http://www.cs.vu.nl/das4/, último acesso em 08 de junho de 2011.
[6] http://pda.gforge.inria.fr/, último acesso em 08 de junho de 2011.
[7] http://www.grid5000.fr/, último acesso em 08 de junho de 2011.

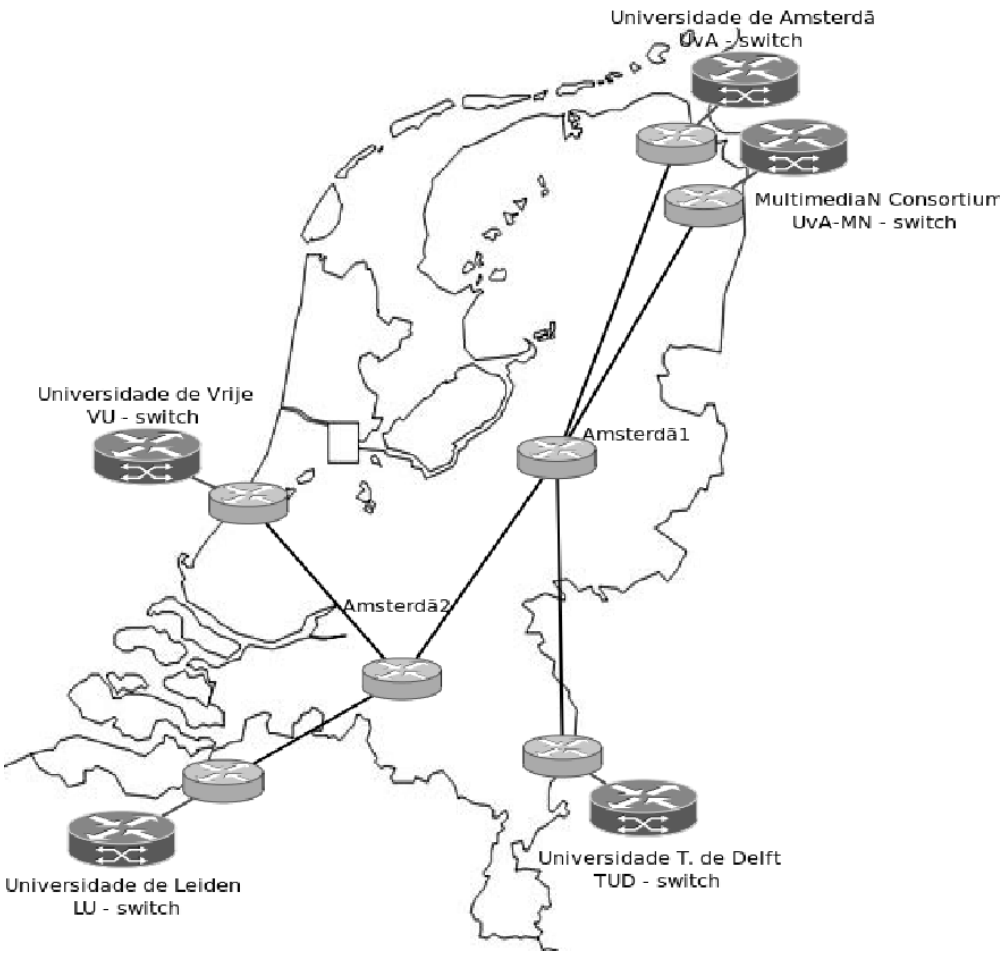


Figura 5.1: *A arquitetura do projeto DAS-3. Cinco instituições conectadas usando a tecnologia* Myri-10G.

inicial do projeto foi alcançar os 5.000 processadores sob a plataforma, posteriormente reformulado para 5.000 núcleos. Este objetivo foi alcançado no ano 2009.

O projeto Grid5000 é um esforço de pesquisa desenvolvendo uma grande infraestrutura de larga escala para pesquisas em computação paralela e distribuída.

5.2.1 Arquitetura

A infraestrutura do projeto está geograficamente distribuída. Inicialmente, estava composta por nove sítios espalhados em diferentes instituições entre universidades e centros de pesquisa pelo território francês. Posteriormente, foram anexados dois nós localizados em Porto Alegre - Brasil e Luxemburgo.

A arquitetura dos processadores que compõem o projeto Grid5000 é heterogênea, entre os principais tipos temos: AMD, Opteron, Intel Xeon, Intel Itanium 2 e PowerPC. Cada aglomerado usa uma interconexão Gigabit (*GigaEthernet ou Myrinet*) e todos os nós no Grid5000 estão interconectados em uma grande rede chamada RENATER[8], cuja velocidade da rede chega aos $10 Gigabit/sec$.

Na Figura 5.2 é apresentada a distribuição dos aglomerados do projeto Grid5000 pelo território francês.

Na Tabela 5.2 são apresentados os aglomerados pertencentes ao Grid5000 os quais serão usados na especificação desta arquitetura. Como podemos notar o número de processadores usados são 462,

[8]http://www.renater.fr/, último acesso em 08 de junho de 2011.

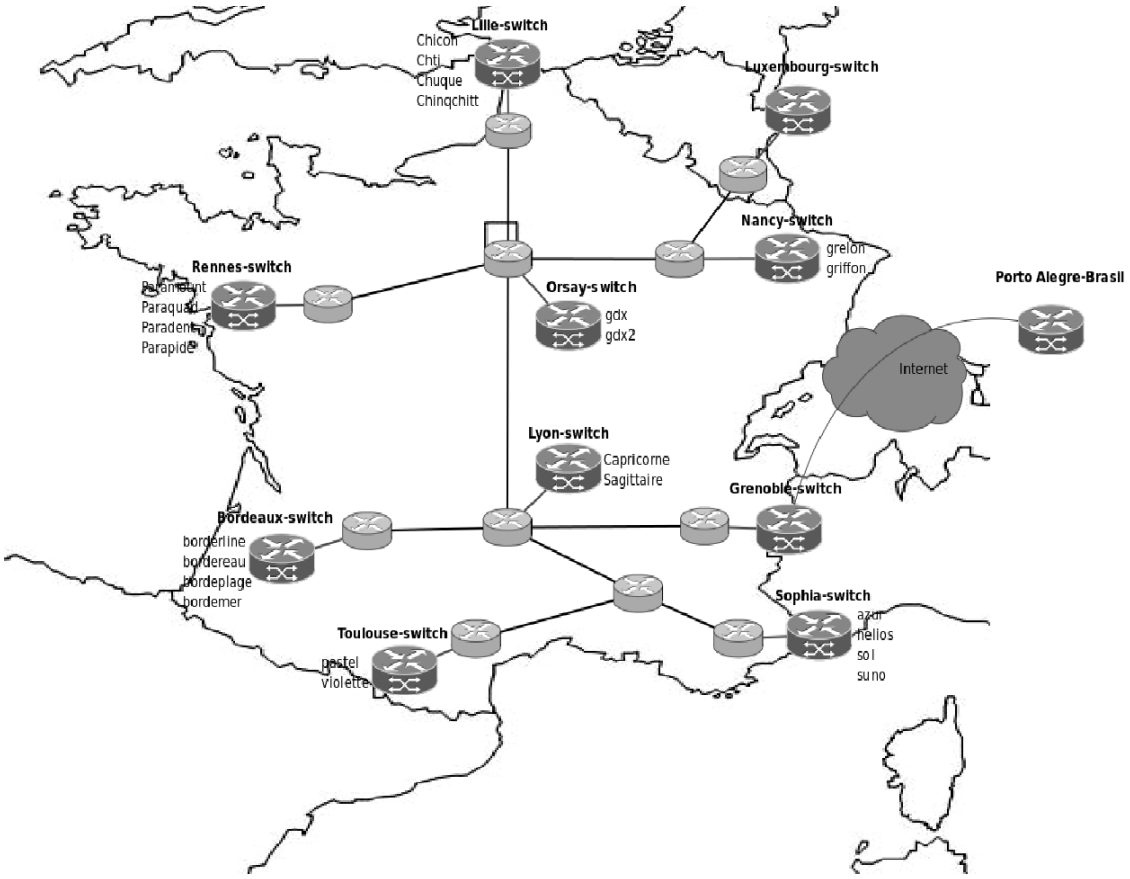


Figura 5.2: *A arquitetura do projeto Grid5000. O mapa da França, onde cada instituição ou universidade estão unidas por uma rede de alta velocidade.*

agrupados em doze aglomerados.

prefixo	**sufixo**	**nós**	**Poder Comp.** ($GFlops/s$)	**Largura de Banda** (Gb/s)
chicon-	.lille.grid5000.fr	26	$4,38$	$1,25$
chti-	.lille.grid5000.fr	20	$4,31$	$1,25$
chuque-	.lille.grid5000.fr	53	$3,65$	$1,25$
chinqchint-	.lille.grid5000.fr	46	$5,00$	$1,25$
paramount-	.rennes.grid5000.fr	33	$4,60$	$1,25$
paraquad-	.rennes.grid5000.fr	30	$4,60$	$1,25$
paradent-	.rennes.grid5000.fr	30	$3,36$	$1,25$
parapide-	.rennes.grid5000.fr	25	$5,00$	$1,25$
azur-	.sophia.grid5000.fr	48	$3,26$	$1,25$
helios-	.sophia.grid5000.fr	56	$3,68$	$1,25$
sol-	.sophia.grid5000.fr	50	$4,39$	$1,25$
suno-	.sophia.grid5000.fr	45	$5,00$	$1,25$

Tabela 5.2: *Características da arquitetura Grid5000. O número de processadores usados nos experimentos é* 462. *A média do poder computacional é* $4,27GFlops/s$.

5.3 O Projeto GridPP

O projeto *Grid for UK Particle Physics* (GridPP)[9] [BCC+09] é uma arquitetura colaborativa entre físicos e cientistas da computação de 19 universidades do Reino Unido, o laboratório Rutherford Appleton[10] e o *CERN*[11], com a finalidade de construir uma grade computacional para pesquisas em física de partículas.

A fase inicial do projeto foi considerada como um protótipo de ambiente para testes da grade (2001-2004); na fase seguinte (2004-2008) foi construída uma estrutura da grade escalável para produção. Atualmente, encontra-se na terceira fase (2008-2011), na qual a grade não somente é um ambiente de testes, além disso, funciona e executa aplicações. Uma das aplicações que o projeto GridPP processa é *Large Hadron Collider* (LHC)[12], o maior acelerador de partículas do mundo.

5.3.1 Arquitetura

O projeto GridPP forma parte da infraestrutura do projeto *LHC Computing Grid (LCG)*. A primeira parte do projeto GridPP que pertence ao LCG é localizada no Laboratório Rutherford Appleton (RAL). A segunda parte é dividida em quatro regiões:

- **LondonGrid**
  - *Brunel University*;
  - *Imperial College*, Londres;
  - *Queen Mary, University of London* (QMUL);
  - *Royal Holloway, University of London* (RHUL);
  - *University College London* (UCL).
- **NorthGrid**
  - *Lancaster University*;
  - *University of Liverpool*;
  - *University of Manchester*;
  - *University of Sheffield*.
- **ScotGrid**
  - *University of Birmingham*;
  - *University of Bristol*;
  - *University of Cambridge*;
  - *University of Oxford*;
  - *Rutherford Appleton Laboratory*.
- **SouthGrid**

[9] http://www.gridpp.ac.uk/, último acesso em 08 de junho de 2011.
[10] http://www.ppd.clrc.ac.uk/, último acesso em 08 de junho de 2011.
[11] http://www.cern.ch/, último acesso em 08 de junho de 2011.
[12] http://lhc.web.cern.ch/lhc/, último acesso em 08 de junho de 2011.

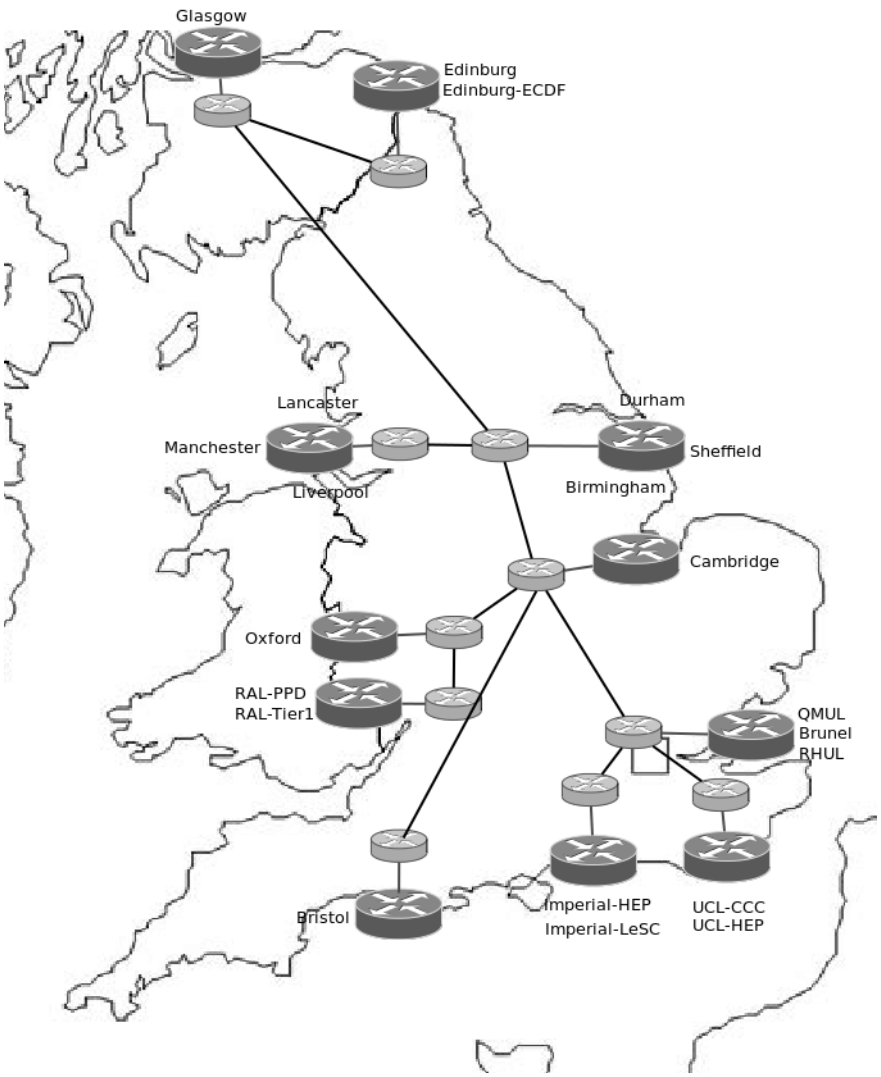


Figura 5.3: *A arquitetura do projeto GridPP.*

- *Durham University*;
- *University of Edinburgh*;
- *University of Glasgow.*

Na Figura 5.3 é representada a arquitetura do projeto GridPP. É representado cada aglomerado no mapa do Reino Unido, cada nó já mencionado acima.

Na Tabela 5.3 são apresentados os aglomerados que serão especificados nos experimentos do presente trabalho.

5.4 Considerações Finais do Capítulo

Neste capítulo, foram apresentadas três arquiteturas atualmente usadas para computação em grade, especificamente para pesquisa relacionada aos sistemas em computação em alto desempenho. A singularidade destas plataformas é que foram criadas com o intuito de fazer pesquisa em computação em grade, paralela e distribuída. Estas plataformas serão usadas nas simulações feitas neste trabalho.

prefixo	**sufixo**	**nós**	**Poder Comp.** ($GFlops/s$)	**Largura de Banda** (Mb/s)
brunel-	.gridpp.uk	101	$2,40$	$125,00$
qmul-	.gridpp.uk	76	$3,60$	$155,00$
rhul-	.gridpp.uk	44	$3,00$	$155,00$
lancaster-	.gridpp.uk	56	$2,40$	$155,00$
liverpool-	.gridpp.uk	65	$2,80$	$155,00$
manchester-	.gridpp.uk	200	$2,40$	$1000,00$
sheffield-	.gridpp.uk	25	$2,20$	$155,00$
durham-	.gridpp.uk	25	$2,00$	$155,00$
birmingham-	.gridpp.uk	32	$2,20$	$622,00$
Edinburgh-ecdf-	.gridpp.uk	107	$3,00$	$1000,00$
edinburgh-	.gridpp.uk	7	$3,20$	$1000,00$
bristol-	.gridpp.uk	24	$2,40$	$622,00$
cambridge-	.gridpp.uk	138	$2,40$	$155,00$

Tabela 5.3: *Características da arquitetura GridPP. O total de processadores é* 900. *A média do poder computacional é* $2,62GFlops/s$.

O projeto DAS-3 possui cinco aglomerados interconectados através da Holanda. Este projeto possui 272 processadores com uma média de poder de processamento de $2,4GFlops/s$. A velocidade de conexão entre os nós da grade é consideravelmente alta, tendo uma média de conexão entre nós de $76,5Gb/s$. Na especificação desta arquitetura foram usadas estas características, vide Apêndice A.

A respeito do projeto Grid5000, ele possui mais de 5000 processadores, espalhados em diferentes partes do território francês. No entanto, neste trabalho serão usados doze aglomerados com um total de 462 processadores; a média de poder de processamento é $4,27GFlops/s$. Nesta grade a tecnologia de comunicação também é alta, com uma média de comunicação entre aglomerados de $10Gigabit/sec$.

O projeto GridPP possui atualmente mais de 7900 processadores em 21 aglomerados espalhados pelo Reino Unido. Desse conjunto de aglomerados para nossos propósitos usamos treze aglomerados, com um total de 900 processadores.

As limitações nos números de processadores das arquiteturas Grid5000 e GridPP foram feitos, principalmente, pelo número de tarefas no conjunto de aplicações, as quais chegam até mil tarefas.

Capítulo 6

Aplicações para Grades

As aplicações usadas nas simulações neste trabalho são provenientes de aplicações paralelas reais (com tarefas dependentes). No trabalho de Shishir Bharathi *et. al.* [BCD+08] aplicações desse tipo foram caracterizadas propondo um formato para a sua especificação denominado DAX (*Directed Acyclic Graph in XML*), esse formato é usado pelo projeto Pegasus[1].

As especificações das aplicações usadas neste trabalho foram geradas usando um gerador de *workflows* (*workflowgenerator*) baseado em estatísticas de execuções das aplicações reais. Essas aplicações são: (a) Montage, (b) CyberShake, (c) Epigenomics e (d) Ligo. No projeto Pegasus, para cada aplicação foram disponibilizadas um conjunto de especificações de diferentes tamanhos em número de tarefas. Essas quantidades de tarefas são: 50, 100, 200, 300, ..., 1000. Além disso, para cada número de tarefas existe um conjunto de 20 instâncias. Nas seguintes seções são descritas as aplicações usadas neste trabalho.

O critério para a escolha dessas aplicações foi principalmente pela estrutura que apresenta a aplicação. Tendo assim, aplicações que apresentam na sua estrutura tarefas críticas, isto significa que na sua estrutura possui alguns gargalos o que faz do escalonamento seja ainda mais complexo.

6.1 A aplicação Montage

O projeto Montage[2] foi criado pela NASA/IPAC e está disponibilizado como software livre. Este sistema é usado para gerar mosaicos personalizados do céu usando pontos de múltiplas imagens de entrada no formato *Flexible Image Transport System* (FITS)[3].

No processo da geração do mosaico final, a geometria do arquivo de saída é calculada a partir das geometrias dos arquivos de entrada. Todos os arquivos de saída são reprojetados para ter a mesma escala e rotação. Os ruídos das imagens são corrigidos no mesmo nível em todas as imagens. Na fase final as imagens são reprojetadas, corrigidas novamente e adicionadas ao mosaico final.

A aplicação Montage tem sido representada como um *workflow* que pode ser executado em ambientes de computação em grade, como o projeto TeraGrid[BBA+04]. Um exemplo da estrutura da aplicação Montage é mostrado na Figura 6.1, esta imagem foi adaptada do trabalho do Shishir Bharathi *et. al.* [BCD+08], cada vértice no grafo representa uma tarefa e cada nível um passo na construção do mosaico final.

[1] https://confluence.pegasus.isi.edu/, último acesso em 07 de junho de 2011.
[2] http://montage.ipac.caltech.edu/, último acesso em 16 de maio de 2011.
[3] Formato padrão usado em astronomia aprovado pela NASA e pela União Astronômica Internacional.

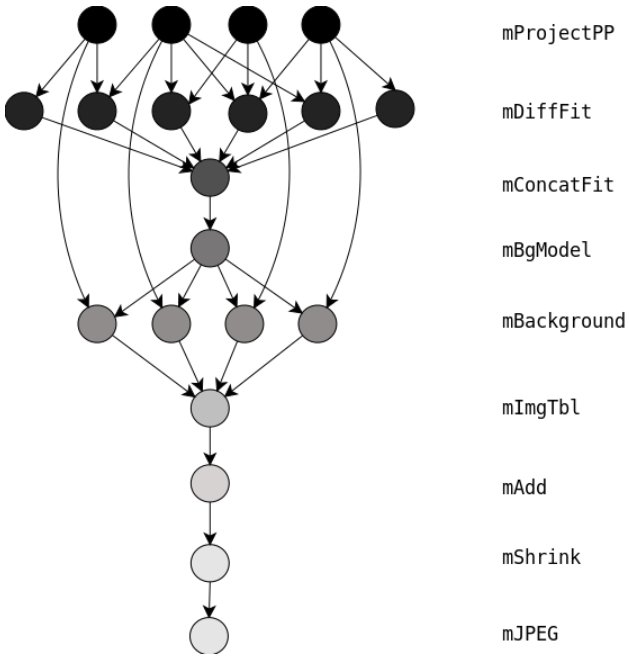


Figura 6.1: *Estrutura da aplicação Montage, cada vértice no grafo representa uma tarefa e cada nível representa um passo na construção do mosaico final.*

O número de entradas na aplicação Montage varia dependendo da quantidade de imagens que serão avaliadas sobre uma determinada região do céu. Desta forma a estrutura muda de acordo com o número de imagens de entrada disponíveis.

Esta aplicação possui uma a particularidade, que na medida em que o número de tarefas é acrescentado, o número de tarefas de alguns níveis aumenta. Neste caso, tarefas nos níveis *mProject*, *mDiffFit* e *mBackGround*, são as que variam, os demais níveis conservam o número de tarefas como fixo. A quantidade de tarefas nos níveis *mProject* e *mBackGround* são sempre é igual.

Por exemplo, na Tabela 6.1 são apresentados os números de tarefas para instâncias de 50, 100 e 200.

Nível	**Número de tarefas**		
	50	100	200
mProject	8	16	32
mDiffFit	28	62	130
mConcatFit	1	1	1
mBgModel	1	1	1
mBackGround	8	16	32
mImgTbl	1	1	1
mAdd	1	1	1
mShrink	1	1	1
mJPEG	1	1	1

Tabela 6.1: *Quantidade de tarefas por nível da aplicação Montage.*

Desta forma, na Tabela 6.2 é mostrada somente a quantidade de tarefas nos níveis *mProject* e *mDiffFit*, dado que, as quantidades nos outros níveis são iguais.

Número de tarefas	Nível	Número de tarefas
50	*mProject*	8
	mDiffFit	28
100	*mProject*	16
	mDiffFit	62
200	*mProject*	32
	mDiffFit	130
300	*mProject*	49
	mDiffFit	196
400	*mProject*	62
	mDiffFit	262
500	*mProject*	82
	mDiffFit	330
600	*mProject*	99
	mDiffFit	396
700	*mProject*	116
	mDiffFit	462
800	*mProject*	132
	mDiffFit	530
900	*mProject*	149
	mDiffFit	596
1000	*mProject*	166
	mDiffFit	662

Tabela 6.2: *Quantidade de tarefas por nível da aplicação Montage.*

6.2 A aplicação CyberShake

O projeto CyberShake[4] tem como propósito, calcular e analisar os riscos de terremoto na região de Los Angeles, usando técnicas de análise probabilística de risco sísmico (*Probabilistic Seismic Hazard Analysis* - PSHA). Também é usado pelo *Southern California Earthquake Center*[5].

Os dados sísmicos são gerados a partir de simulações *3D*[6] de movimentos da terra ao invés de relações empíricas, esses dados são extraídos por tarefas no nível *ExtractSGT* e são agrupados em pares `<fonte, ruptura>`. Por cada par gerado é gerado um sismograma sintético processado no nível *SeismogramSynthesis*, este passo continua em uma seguinte etapa, *PeakValCalcOkaya*, que procura os valores intensivos, ou seja, o epicentro. Finalmente os passos *SeismogramSynthesis* e *PeakValCalcOkaya* são comprimidos pelas tarefas *ZipSeis* e *ZipPSA*, respectivamente. A estrutura de um fluxo do sistema CyberShake é mostrada na Figura 6.2.

Os níveis *ZipSeis* e *ZipPSA* possuem em todos os casos somente uma tarefa. A variação do número de tarefas acontece nos níveis *ExtractSGT*, SeismogramSynthesis e *PeakValCalcOkaya*, mas os dois últimos possuem sempre igual número de tarefas. Portanto, na Tabela 6.3 são mostradas as quantidades de tarefas dos níveis *ExtractSGT* e *SeismogramSynthesis*. Em alguns casos encontramos números da forma 2(10), 4(10), isso significa que das 20 instâncias se têm 10 instâncias de 2 tarefas e 10 instâncias de 4 tarefas.

[4]http://epicenter.usc.edu/cmeportal/CyberShake.html, último acesso em 16 de maio de 2011.
[5]http://www.scec.org/, último acesso em 16 de maio de 2011.
[6]Em três dimensões.

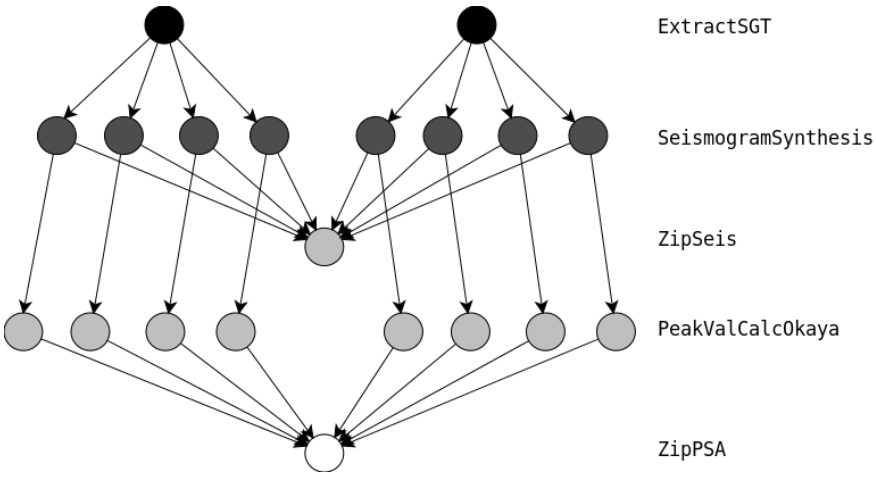


Figura 6.2: *Estrutura da aplicação CyberShake, o vértice do grafo representa uma tarefa e cada nível representa uma fase na extração e empacotamento dos dados na aplicação CyberShake.*

Número de tarefas	Nível	Número de tarefas
50	*ExtractSGT*	2
	SeismogramSynthesis	23
100	*ExtractSGT*	2
	SeismogramSynthesis	48
200	*ExtractSGT*	2
	SeismogramSynthesis	98
300	*ExtractSGT*	2
	SeismogramSynthesis	148
400	*ExtractSGT*	2
	SeismogramSynthesis	198
500	*ExtractSGT*	2(10), 4(10)
	SeismogramSynthesis	247, 248
600	*ExtractSGT*	2(12), 4(8)
	SeismogramSynthesis	297, 298
700	*ExtractSGT*	2(3), 4(15), 6(2)
	SeismogramSynthesis	348, 347, 346
800	*ExtractSGT*	4(11), 6(4), 8(5)
	SeismogramSynthesis	397, 396, 395
900	*ExtractSGT*	4(8), 6(9), 8(3)
	SeismogramSynthesis	447, 446, 445
1000	*ExtractSGT*	4(6), 6(6), 8(3), 10(5)
	SeismogramSynthesis	497, 496, 495, 494

Tabela 6.3: *Quantidade de tarefas por nível da aplicação CyberShake.*

6.3 A aplicação Epigenomics

O projeto Epigenomics foi criado pelo centro do genoma da *University Southern California (USC)* e pelo *Pegasus Team*. Atualmente o centro do genoma da *USC* está envolvido no mapeamento do estado epigenético de células humanas sobre uma grande escala genômica.

O sistema Epigenomics é basicamente um *pipeline* de processamento de dados que usa o *Pegasus Workflow Management System* para automatizar a execução de várias operações em sequência do

genoma. Através de um sistema de análise genética, Illumina-Solexa[7], a sequência de DNA gerada é dividida em vários conjuntos de processos que são operados em paralelo.

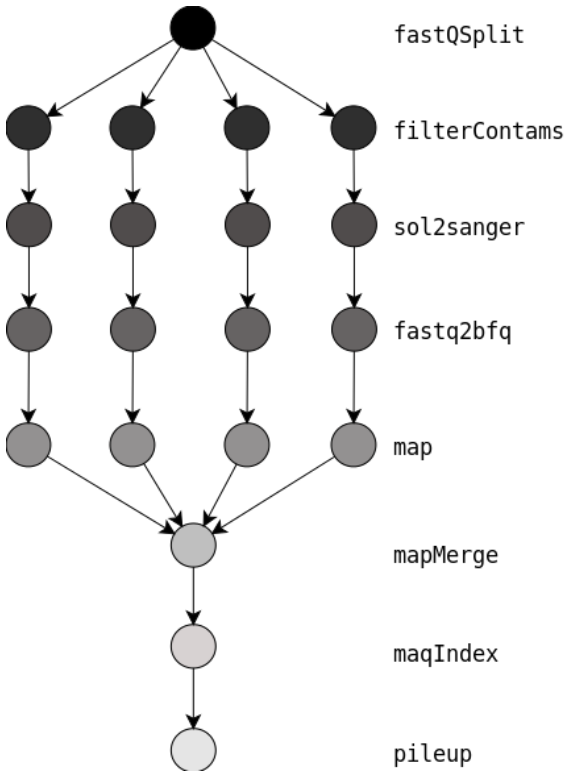


Figura 6.3: *Estrutura da aplicação Epigenomics, cada vértice do grafo representa uma tarefa e cada nível representa uma fase no processamento da sequência do DNA na aplicação Epigenomics.*

A estrutura da aplicação Epigenomics é mostrada na Figura 6.3. Os dados de entrada da aplicação Epigenomics são obtidos pelo sistema de análise genética, esses dados são divididos pela tarefa pertencente ao nível *fastQSplit*, o número de divisões gerado depende dos dados de entrada. O seguinte nível no processamento do DNA é o *filterContams*, nesse nível são filtrados os dados com ruído. Os dados depois de serem filtrados são convertidos em um formato entendível pelo sistema no nível *sol2sanger*, depois desse processo são binarizados no nível *fastq2bfq*. O processo *map* prepara aos dados para serem juntados no nível *mapMerge*. Finalmente o *maqIndex* processa a saída gerada pelo *mapMerge* e termina no *pileup*, tendo um formato de saída específico para a sequência de DNA.

Como podemos observar na Figura 6.3, os níveis *filterContams*, *sol2sanger*, *fastq2bfq* e *map* sempre possuem mesma quantidade de tarefas. Por outro lado, os níveis maqIndex e *pileup* sempre possuem somente uma tarefa. Portanto, na Tabela 6.4 é mostrado o número de tarefas nos níveis *fastQSplit*, *filterContams* e *mapMerge*. Uma convenção usada nesta estrutura é que nem sempre o número de tarefas é exatamente o mesmo número como acontece nas anteriores estruturas, em alguns casos tem três tarefas a menos, este fato é por causa da estrutura, mas serão considerados como se fossem os valores gerais das estruturas.

[7] http://www.illumina.com/, último acesso em 16 de maio de 2011.

Número de tarefas	Nível	Número de tarefas
47	*fastQSplit*	2
	filterContams	10
	mapMerge	3
100	*fastQSplit*	1
	filterContams	24
	mapMerge	1
199	*fastQSplit*	2
	filterContams	48
	mapMerge	3
297	*fastQSplit*	7
	filterContams	70
	mapMerge	8
399	*fastQSplit*	18
	filterContams	90
	mapMerge	19
497	*fastQSplit*	19
	filterContams	114
	mapMerge	20
597	*fastQSplit*	27
	filterContams	135
	mapMerge	28
699	*fastQSplit*	12
	filterContams	168
	mapMerge	13
799	*fastQSplit*	2
	filterContams	198
	mapMerge	3
897	*fastQSplit*	3
	filterContams	222
	mapMerge	4
997	*fastQSplit*	7
	filterContams	245
	mapMerge	8

Tabela 6.4: *Quantidade de tarefas por nível da aplicação Epigenomics.*

6.4 A aplicação LIGO

O *Laser Interferometer Gravitational Wave Observatory* (LIGO) está tentando detectar ondas gravitacionais produzidas por vários eventos no universo. O *workflow* LIGO é usado para analisar os dados obtidos a partir da coalescência de sistemas binários compactos tais como estrelas de nêutrons e os buracos negros binários.

Os dados de tempo-frequência de qualquer evento para cada um dos três detectores LIGO é dividido em blocos menores para análise. A estrutura da aplicação LIGO é mostrada na Figura 6.4. Temos seis níveis, no primeiro nível *tmpltBank* são identificadas famílias de ondas que pertencem ao espaço, os dados de entrada no sistema são gerados pelos detectores do LIGO. A saída do *tmpltBank* são bancos de parâmetros de onda que são usadas para serem filtradas no processo

Inspiral. Os gatilhos produzidos por múltiplas tarefas no *Inspiral* são testados pelo processo *Thinca*. A saída dos dados no *Thinca* é a entrada no *TrigBank* que gera modelos de bancos fora dos gatilhos. Esses modelos de bancos são usados pelo quinto nível *Inspiral*. Finalmente são testados novamente por outro processo *Thinca*.

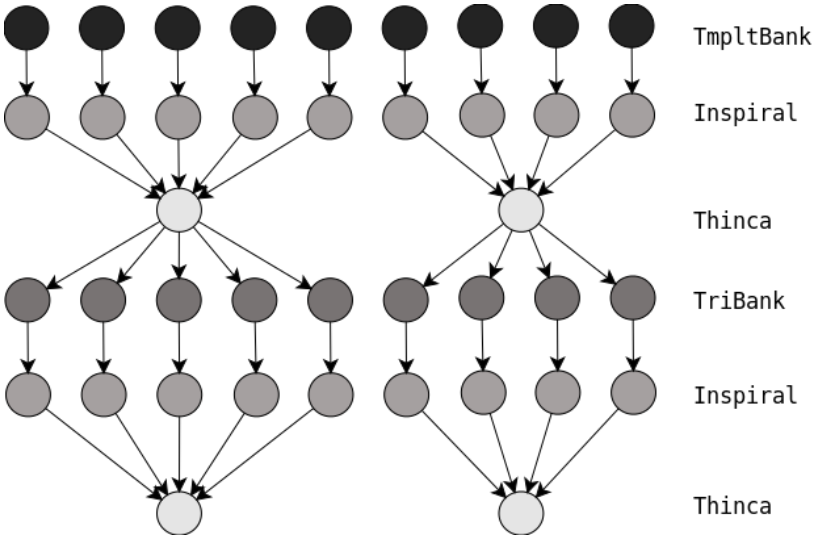


Figura 6.4: *Estrutura da aplicação LIGO, cada circulo representa uma tarefa e cada nível representa uma fase na geração de dados e processamento da na aplicação LIGO.*

Na estrutura do sistema LIGO vemos que a o primeiro nível *tmpltBank* e o segundo nível (primeiro) *Inspiral* possuem a mesma quantidade de tarefas. De forma similar acontece com o quarto e quinto nível, *TrigBank* e (segundo) *Inspiral*, respectivamente. No caso do terceiro nível (primeiro) Thinca e sexto (segundo) *Thinca* possuem a mesma quantidade de tarefas. Portanto na Tabela 6.5 são apresentados as quantidades de tarefas nos níveis *tmpltBank*, *TrigBank* e primeiro Thinca. Essas quantidades são apresentadas como intervalos, por exemplo, $10 - 12$, significa que os valores variam desde 10 até 12 tarefas.

6.5 Considerações Finais do Capítulo

Neste capítulo foram apresentadas quatro aplicações que fazem uso da computação em grade. As estruturas das aplicações mostradas são complexas, especialmente para o escalonamento. Assim, o intuito é entender como os algoritmos de escalonamento se comportam com cada uma dessas aplicações.

Cada aplicação possui uma quantidade de tarefas que é determinada pelos dados de entrada, esse número de tarefas é dado a seguir: 50, 100, 200, 300, ..., 1000.

O projeto Pegasus gerou e disponibilizou esse conjunto de especificações de cada uma dessas aplicações para interesses de pesquisa.

Número de tarefas	**Nível**	**Número de tarefas**
50	*tmpltBank*	$10 - 12$
	Thinca	$1 - 2$
	TrigBank	$12 - 13$
100	*tmpltBank*	$21 - 24$
	Thinca	$2 - 4$
	TrigBank	$23 - 26$
200	*tmpltBank*	$43 - 49$
	Thinca	$2 - 8$
	TrigBank	$47 - 51$
300	*tmpltBank*	$65 - 73$
	Thinca	$2 - 13$
	TrigBank	$70 - 77$
400	*tmpltBank*	$87 - 98$
	Thinca	$3 - 16$
	TrigBank	$93 - 101$
500	*tmpltBank*	$109 - 122$
	Thinca	$2 - 24$
	TrigBank	$117 - 130$
600	*tmpltBank*	$134 - 146$
	Thinca	$2 - 25$
	TrigBank	$138 - 154$
700	*tmpltBank*	$159 - 169$
	Thinca	$2 - 31$
	TrigBank	$160 - 181$
800	*tmpltBank*	$181 - 198$
	Thinca	$2 - 28$
	TrigBank	$191 - 207$
900	*tmpltBank*	$204 - 222$
	Thinca	$2 - 34$
	TrigBank	$217 - 231$
1000	*tmpltBank*	$227 - 246$
	Thinca	$2 - 31$
	TrigBank	$242 - 261$

Tabela 6.5: *Quantidade de tarefas por nível da aplicação Ligo.*

Capítulo 7

Simulador

Um dos aspectos fundamentais no desenvolvimento de aplicações e novas estratégias em sistemas distribuídos e especificamente na computação em grade é a sua avaliação. Neste cenário, geralmente são usadas ferramentas de simulação, principalmente porque a realização de experimentos reais em recursos reais não é apropriada pelos seguintes motivos:

- Aplicações reais devem ser executadas por longos períodos de tempo, e não são viáveis para executar um grande número de experimentos de simulação neles;
- A utilização de recursos reais torna difícil a exploração de uma grande variedade de configurações de recursos;
- Variações na carga de recursos ao longo do tempo tornam difícil a obtenção de resultados reprodutíveis.

7.1 Simuladores para Grades

Mesmo a simulação na computação em grade sendo mais complexa pelas características já comentadas, foram desenvolvidas soluções para a simulação de computação em larga escala, algumas se focando principalmente nas grades computacionais.

7.1.1 Bricks

O simulador Bricks [TMN+99][1] é um sistema de avaliação de desempenho de algoritmos de escalonamento para sistemas computacionais de alto desempenho, entre eles grades computacionais. O Bricks simula diversos comportamentos de sistemas de computação distribuída, focado especialmente para avaliar o comportamento de topologia de sistemas cliente-servidor, estratégias de processamento para redes, recursos e algoritmos de escalonamento.

O simulador Bricks possui mecanismos de coleta de informações dos recursos computacionais por meio de monitoração e predição da disponibilidade dos recursos; essa informação é fundamental para os algoritmos de escalonamento.

A arquitetura do Bricks é composta de um "ambiente de computação global" (*Global Computing Environment*) o qual é responsável pela monitoração, predição dos recursos e escalonamento de ta-

[1] http://ninf.apgrid.org/bricks/, último acesso em 08 de junho de 2011.

refas. O segundo componente é a "unidade de escalonamento" (*Scheduling Unit*), o qual compreende os usuários que submetem tarefas, os recursos que executam essas tarefas e a rede que os interligam.

O simulador Bricks emprega uma linguagem script para modelar a plataforma da grade. As desvantagens para o uso do simulador Bricks são principalmente a pouca documentação a respeito da ferramenta e o fato do código fonte não encontra-se disponível.

7.1.2 Optorsim

O simulador OptorSim [BCC+03][2] é um simulador para grades computacionais criado para testar algoritmos de escalonamento que usam replicação dinâmica para melhorar a eficiência na execução de aplicações. Neste cenário, o termo "replicação" significa criação de cópias de tarefas nos recursos disponíveis na grade.

A arquitetura do Optorsim foi construída assumindo que a grade possui diversos nós. Cada nó deve fornecer recursos computacionais e de armazenamento de dados para as tarefas submetidas. Cada nó é formado por zero ou mais "elementos computacionais" CE (*Computing Elements*) e zero ou mais "elementos de armazenamento" SE (*Storage Elements*). Os CEs executam tarefas que usam os dados armazenados em arquivos nos SEs.

Um componente denominado "Agente de Recursos" (*Resource Broker*) tem a função de controlar o escalonamento das tarefas nas CEs. O "gerenciador de réplicas" (*Replica Manager*) tem as funções das decisões sobre os movimentos dos dados entre nós.

A modelagem da plataforma da grade no Optorsim é feita mediante um arquivo de configuração, nesse arquivo é especificado cada componente descrito anteriormente.

7.1.3 GridSim

O simulador GridSim [BM02][3] é um *toolkit* que permite modelagem e simulação de entidades em sistemas de computação paralela e distribuída, como usuários, aplicações, recursos e *resource brokers* ou escalonadores, para a modelagem e avaliação de algoritmos de escalonamento.

O simulador GridSim fornece um mecanismo global para a simulação de diferentes classes de recursos heterogêneos, usuários, aplicações, e *resource brokers*. Pode ser usado para simular aplicações de escalonamento para diferentes domínios de sistemas de computação distribuída como aglomerados ou grades.

A modelagem da plataforma da grade é feita descrevendo-se diretamente no código fonte do simulador. Este simulador fornece uma API (*Application Programming Interface*) para declarar os diferentes tipos de recursos e as interconexões de rede. Portanto, descrever a plataforma no código diretamente torna difícil a execução do mesmo código sobre diferentes plataformas.

7.1.4 SimGrid

O simulador SimGrid [CLQ08][4] é um *toolkit* que fornece importantes funcionalidades para a simulação de aplicações distribuídas em ambientes heterogêneos. O objetivo do projeto SimGrid é facilitar a pesquisa de escalonamento de aplicações paralelas e distribuídas sobre plataformas de arquiteturas paralelas e distribuídas.

[2] http://sourceforge.net/projects/optorsim/, último acesso em 08 de junho de 2011.
[3] http://sourceforge.net/projects/gridsim/, último acesso em 08 de junho de 2011.
[4] http://simgrid.gforge.inria.fr/, último acesso em 08 de junho de 2011.

A arquitetura do simulador SimGrid é descrita em forma de camadas, como é mostrado na Figura 7.1. A primeira camada fornece ambientes de programação ao usuário. Na segunda camada se encontra implementado todo o mecanismo do SimGrid para a simulação da plataforma. Finalmente na terceira camada estão implementadas as estruturas de dados, a portabilidade do simulador e os registros de eventos (*logging*).

A modelagem da plataforma é feita mediante a especificação de um arquivo externo ao simulador. Assim, é possível especificar diferentes tipos de plataformas para grades sem ter a necessidade de modificar o código.

O simulador é código livre e é implementado na linguagem *C*, mas fornece como alternativa o uso da linguagem *Java*. O projeto continuamente disponibiliza melhoras do simulador possuindo uma comunidade ativa.

O simulador usado no presente trabalho é o SimGrid, pelas qualidades comentadas anteriormente. Além disso, ele possui exatamente o que é preciso para nossos propósitos. Na seguinte seção será apresentado o SimGrid com maiores detalhes.

7.2 O Simulador SimGrid

O projeto SimGrid foi iniciado em 1999 com o objetivo de permitir o estudo de estratégias de escalonamento sobre plataformas heterogêneas. A versão estável atual do simulador é 3.5, liberada no dia 02 de dezembro de 2010. O projeto está em constante desenvolvimento, corrigindo possíveis erros e melhorando o código disponibilizado no seu repositório, *SVN*[5].

O simulador SimGrid fornece várias *APIs* de programação e um conjunto de funcionalidades abstratas que permitem um fácil uso para simulações sobre arquiteturas de sistemas paralelos e distribuídos.

7.2.1 Arquitetura

A arquitetura do simulador é formada por três camadas: "ambientes de programação" (*Programmation environments layer*), "Núcleo de simulação" (*Simulation kernel*) e "Base" (*Base layer*), como é apresentado na Figura 7.1.

O usuário pode desenvolver sua aplicação fazendo uso de algum dos ambientes de programação situados na primeira camada. O projeto SimGrid disponibiliza além da biblioteca, gerada na instalação, um conjunto de exemplos e aplicações desenvolvidos e compartilhados por usuários.

7.2.2 Componentes do SimGrid

O simulador SimGrid, como foi citado anteriormente, possui três camadas, cada uma destas camadas é dividida em componentes, os quais são fundamentais para a simulação. A seguir, serão detalhados os componentes de cada camada.

Camada: Ambientes de Programação

O SimGrid fornece diversos ambientes de programação, construídos sobre um único núcleo. Cada ambiente objetiva um alvo específico e constitui um paradigma diferente. Para escolher o

[5] svn://scm.gforge.inria.fr/svn/simgrid.

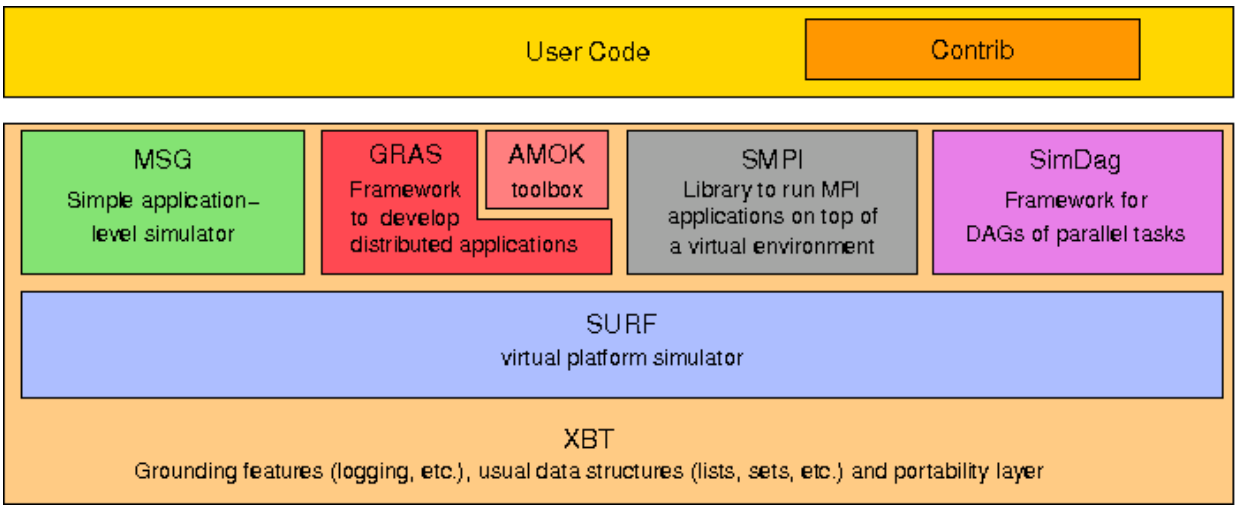


Figura 7.1: *Componentes do simulador SimGrid.*

mais adequado, o desenvolvedor deve avaliar se o seu problema é coberto pelas características do ambiente. Na Figura 7.2 são mostrados os componentes da camada de ambientes de programação; cada componente é detalhado a seguir.

- **MSG** (*Meta-SimGrid*): foi o primeiro ambiente de programação disponibilizado e de uso mais difundido. Ele é usado para modelar aplicações como processos sequenciais concorrentes (*Concurrent Sequential Processes*);
- **SMPI** (*Simulated MPI*): destinado para simulações de códigos MPI. Simulações do comportamento de uma aplicação MPI usando técnicas de emulação;
- **GRAS** (*Grid Reality And Simulation*): neste ambiente é possível a execução de aplicações reais com o intuito de analisá-las e testá-las. Acima deste ambiente tem um *toolkit* chamado *Advanced Metacomputing Overlat Kit* (AMOK) que implementa em alto nível diversos serviços necessários para a execução das aplicações distribuídas;
- **SimDag**: é um ambiente dedicado à simulação de aplicações paralelas, por meio do modelamento de grafos acíclicos dirigidos (*Directed Acyclic Ggraphs - DAG*. Com este modelo, é possível especificar relações de dependência entre tarefas de uma aplicação paralela.

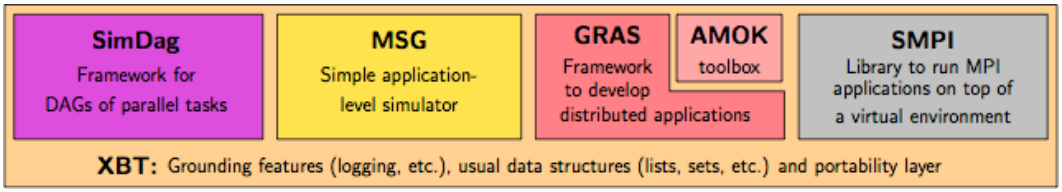


Figura 7.2: *Camada: Ambientes de Programação (*Programmation environments*).*

Camada: Núcleo de simulação

As funcionalidades para simular uma plataforma virtual são fornecidas pelo módulo chamado *SimUlation aRtiFact* (SURF). É de muito baixo nível e não é destinado para ser utilizado como tal pelos usuários finais. Em vez disso, ele serve de base para a camada de nível superior.

Uma das principais características do SURF é a capacidade de mudar de forma transparente o modelo utilizado para descrever a plataforma. Isto facilita bastante a comparação dos vários modelos existentes na literatura.

Camada: Base

A camada Base da ferramenta é constituída pelo *eXtended Bundle of Tools* (XBT). É uma biblioteca portátil, fornecendo alguns recursos como registro de *logs*, lançamento de exceções e suporte a configurações.

O módulo *XBT* também possui as seguintes estruturas de dados: *Dynar*: matriz dinâmica genérica, *Fifo*: fila genérica, *Dict*: dicionário genérico, *Heap*: um *heap* genérico, *Set*: conjunto de dados genéricos, e *Swag*: tipo de dado baseado em listas encadeadas.

7.2.3 Implementação e Documentação

O SimGrid é disponibilizado como software livre, funciona em modo texto, e está disponível para ambientes Linux, Windows e MacOS. É implementado na linguagem de programação C.

Algumas estratégias de otimização são utilizadas na manipulação de *traces*, eles podem ser compartilhados pelos recursos; conjuntos grandes de *traces* são carregados em memória apenas quando são necessários e depois de serem usados, são descartados.

7.2.4 Modelagem da Plataforma da Grade e os Workloads

Uma importante característica do SimGrid é a forma como são especificadas a plataforma computacional e as aplicações. Ambas as especificações são feitas mediante arquivos em formato XML (*Extensible Markup Language*).

No caso das plataformas, o SimGrid já possui etiquetas definidas para a representação de cada um dos componentes necessários[6]. É possível especificar todo o conjunto de recursos da grade, recursos computacionais, aglomerados, assim como a estrutura de interconexão entre eles. Nos recursos computacionais que executam tarefas são modelados pelo seu poder computacional, número de cores, entre outras características. Enquanto que os *links* de comunicação são modelados pela sua largura de banda e latência. Entre dois recursos pode haver mais de um *link*, e um dado *link* pode ser utilizado tanto para enviar quanto para receber tarefas [FQS08].

No caso do ambiente de tarefas, é usado também um formato específico. No ambiente de programação *MSG*, o arquivo especifica os processos que serão executados em cada recurso e argumentos necessários e opcionais pertencentes a cada processo.

No caso do ambiente de programação *SimDag*, é possível criar todas as tarefas e as dependências diretamente do código. Além disso, o simulador SimGrid possui funções que facilitam ao desenvolvedor a criação de tarefas. O simulador possui atualmente duas funções chamadas `SD_daxload()` e `SD_dotload()`. Essas funções dão suporte respectivamente para dois formatos preestabelecidos, o formato DAX (*Directed Acyclic Graph in XML*) [BCD+08][7], e o formato DOT[8], que é um formato para descrever aplicações modeladas por DAGs.

[6] http://simgrid.gforge.inria.fr/simgrid.dtd, último acesso em 08 de junho de 2011.
[7] https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, último acesso em 08 de junho de 2011.
[8] http://www.graphviz.org/doc/info/lang.html, último acesso em 08 de junho de 2011.

7.3 Considerações Finais do Capítulo

Neste capítulo foram apresentados alguns simuladores para sistemas distribuídos, especificamente para computação em grade. Dentre os simuladores o melhor avaliado e mais adequado para os propósitos do presente trabalho foi o SimGrid, principalmente porque ele possui as características suficientes para fazer as simulações tanto do ambiente da grade quanto as aplicações.

O simulador SimGrid possui três camadas: (a) Ambientes de Programação, (b) Núcleo de simulação e (c) Base. O usuário faz uso da camada de Ambientes de Programação para simular sua aplicação, fazendo uso do componente adequado. Neste trabalho o componente usado foi o *SimDag*, cujo propósito principal é a simulação de aplicações paralelas modeladas por DAGs. O modelamento das arquiteturas para grade é feito mediante a camada Núcleo de simulação, nesta camada se encontra as funcionalidades para simular a plataforma virtual. Na camada base se encontra os tipos e estruturas de dados registro de *logs* entre outros.

No próximo capítulo serão apresentados os algoritmos de escalonamento para grades, tanto para tarefas independentes quanto para tarefas dependentes. Será feito uma breve descrição dos algoritmos para tarefas independentes e uma descrição aprofundada dos algoritmos para tarefas dependentes que é o ponto central do presente trabalho.

Capítulo 8

Algoritmos de Escalonamento

Nos Capítulos 4 e 3 pudemos perceber que, apesar de haver algoritmos de escalonamento sofisticados disponíveis na literatura, a maioria dos escalonadores para aglomerados e *middlewares* para grades usam uma fila simples, por exemplo, uma fila de tipo FIFO (*First In First Out*). Alguns usam técnicas de replicação no escalonamento, para tentar diminuir o tempo de término das tarefas em processamento. Porém, a maioria deles é extensível, portanto, existe a possibilidade de implementar algoritmos de escalonamento especializados.

A utilização de um algoritmo de tipo FIFO é adequada somente em arquiteturas homogêneas, arquiteturas representadas por $\langle P||C_{max}\rangle$, onde P representa o tipo de máquinas (recursos) idênticas e o C_{max} representa o *Makespan*, minimização do tempo de término das aplicações. Como já vimos, o problema de escalonamento é NP-Completo [Pin08]. O problema de escalonamento se torna mais desafiador devido a algumas características únicas pertencentes à computação em grade, tais como, a sua heterogeneidade e alta dinamicidade. Infelizmente, algoritmos de escalonamento de sistemas paralelos e distribuídos tradicionais, como é o caso do FIFO, o qual é usado em sistemas homogêneos e dedicados, podem não ser bem adequados para essas novas características [DA06].

Os tipos de aplicações avaliadas neste trabalho serão: aplicações com tarefas dependentes e aplicações com tarefas independentes. Na seguinte seção, serão apresentados os algoritmos mais relevantes de acordo com a literatura.

8.1 Algoritmos de Escalonamento para Tarefas Independentes

No contexto de aplicações executadas nas grades computacionais, aplicações com tarefas independentes podem ser escalonadas em qualquer ordem. Este tipo de aplicações é referenciado na literatura como aplicações (BoT) [CPC+03].

8.1.1 O Algoritmo WQR

O algoritmo *Workqueue with Replication* (WQR) [dSCB03] foi criado para solucionar o problema de obtenção de informações sobre a aplicação e os recursos da grade.

O algoritmo WQR em sua fase inicial é similar à heurística de escalonamento tradicional *Workqueue* [dSCB03], onde as tarefas são escolhidas em uma ordem arbitrária e enviadas ao processador quando estes estiverem disponíveis.

As heurísticas WQR e *Workqueue* passam a diferir no momento em que um processador se torna

disponível e não há mais nenhuma tarefa pendente para executar. Neste momento, *Workqueue* já terminou seu trabalho e apenas aguarda a finalização de todas as tarefas. Porém, o WQR inicia sua fase de replicação para as tarefas que ainda estão executando. Note que, na fase de replicação, quando uma tarefa original finaliza sua execução antes que suas réplicas, estas são interrompidas. Caso contrário, quando alguma réplica finaliza primeiro, a tarefa original e as demais réplicas dela são interrompidas.

O algoritmo WQR alcança bons níveis de desempenho sem a utilização de informações dinâmicas sobre os processadores, conexões de rede ou mesmo sobre o tempo de execução das tarefas, considerando aplicações que não processam dados de forma intensiva. Há, porém, um efeito colateral no uso da replicação na heurística WQR: as réplicas que são interrompidas geram um uso desnecessário de ciclos de CPU, mesmo que haja interrupção [dSCB03]. Por outro lado, uma das principais vantagens da replicação neste cenário é que indiretamente já se trabalha a parte de tolerância a falhas.

8.1.2 O Algoritmo XSufferage

Outra heurística popular para tarefas independentes é chamado *Sufferage* [CLZB00]. A lógica por detrás do *Sufferage* é que uma tarefa deveria ser alocada a certa máquina e, se isso não acontece, a tarefa vai sofrer mais. Isto significa que cada tarefa possui um valor de prejuízo ou sofrimento chamado *sufferage*, que é definido pela diferença entre os processadores que oferecem o melhor *Minimum completion time* (MCT) e o segundo melhor MCT. Tarefas com maior valor de *sufferage* tomam precedência.

O *XSufferage* [CLZB00] é uma extensão de heurística de escalonamento *Sufferage*. A ideia básica da heurística *Sufferage* é determinar quanto cada tarefa seria prejudicada se não fosse escalonada no processador que a executaria de forma mais eficiente. Portanto, o *Sufferage* prioriza as tarefas de acordo com o valor que mede o prejuízo de cada uma. A principal diferença entre *Sufferage* e *XSufferage* é o método usado para calcular o valor do *sufferage*. No *XSuffrage* o valor do *sufferage* não é calculado só com o MCT, mas sim considerando o nível de *tempo de conclusão mínimo* dos aglomerados (*cluster-level MCTs*), ou seja, *tempo de conclusão mínimo* pelo cálculo do mínimo sobre todas as máquinas em cada aglomerado. Assim, para cada tarefa é calculado o valor do *sufferage* no aglomerado. Esse valor é a diferença entre o melhor e o segundo melhor valor *cluster-level MCTs*.

Portanto, o algoritmo de escalonamento *XSufferage* usa informações sobre os níveis do cluster, os quais precisam estar disponíveis no momento em que o algoritmo vai alocar as tarefas. Algumas das informações que ele precisa são:

- disponibilidade de CPU;
- disponibilidade da rede;
- tempo de execução das tarefas.

Estas informações precisam ser conhecidas antes do escalonamento, isto com o alvo de fazer os cálculos sobre qual processador oferece o melhor MCT. Assim, com essas informações em consideração, é possível decidir qual tarefa deve ser escalonada em qual processador. Um ponto importante a ser observado é que o algoritmo considera somente os recursos livres no momento em que vai

escalonar uma tarefa, pois, caso contrário, sempre o recurso mais rápido e com a melhor conexão de rede receberia todas as tarefas.

8.1.3 O Algoritmo Storage Affinity

O algoritmo *Storage Affinity* [SNCBL04] leva em conta o fato dos dados de entrada da aplicação serem frequentemente reutilizados em execuções sucessivas.

O método de escalonamento é definido sobre o conceito fornecido pelo próprio nome do algoritmo. A *afinidade* é o valor da afinidade entre uma tarefa e uma máquina, a qual determina quão próxima da máquina a tarefa se encontra. A semântica do termo *próximo* está associada à quantidade de *bytes* da entrada da tarefa que já está armazenada remotamente em uma dada máquina, assim, quanto mais *bytes* da entrada da tarefa estiverem armazenados na máquina, mais próxima a tarefa estará da máquina, pois possui mais afinidade no armazenamento (*storage affinity*).

Além de aproveitar a reutilização de dados, o *Storage Affinity*, também realiza replicação de tarefas, obtendo vantagem na dificuldade de obtenção de informações dinâmicas sobre a grade, e sobre o tempo de execução das tarefas. A ideia é que as réplicas tenham a chance de serem submetidas a processadores mais rápidos do que aqueles associados a tarefas originais, reduzindo o tempo de execução da aplicação.

Cirne *et. al.* [SNCBL04] comparam os algoritmos *Storage Affinity*, o *XSufferage* e o *WQR*. Nesse mesmo trabalho, o algoritmo *XSufferage* e o *Storage Affinity* apresentam melhor desempenho que o *WQR*. O algoritmo *XSufferage* supera ao *Storage Affinity* somente quando a granularidade da aplicação é grande, no entanto, se a granularidade em tipos de aplicações BoT fosse reduzida, o makespan do algoritmo *Storage Affinity* superaria ao *XSufferage* em 42%.

8.2 Algoritmos de Escalonamento para Tarefas Dependentes

Os algoritmos de escalonamento mostrados nesta seção são algoritmos que representam aplicações com tarefas dependentes. As heurísticas neste tipo de aplicações são classificadas em várias categorias, principalmente pelas técnicas que usam. Essas técnicas são descritas a seguir:

- *List-scheduling*: a lista ordenada de tarefas é construída através da atribuição de prioridade para cada tarefa. As tarefas são selecionadas pela ordem de suas prioridades e cada tarefa selecionada é escalonada em um recurso que minimiza uma função custo predefinido;
- *Clustering*: esta técnica consiste em criar grupos de tarefas (*clusters* de tarefas) que têm dependência de dados entre si. Essas tarefas são escalonadas no mesmo recurso;
- *Duplication-based*: consiste na duplicação de tarefas em mais de um recurso. Assim, cada tarefa pode ser escalonada em dois ou mais recursos distintos.

8.2.1 Problema de Escalonamento para Tarefas Dependentes

Uma aplicação com tarefas dependentes é representada ou modelada por um grafo acíclico dirigido (DAG).

Seja, $G = (V, E)$, um grafo acíclico dirigido, onde:

- V é o conjunto de tarefas v;

- E é o conjunto de relações de precedência entre as tarefas.

Neste contexto cada aresta $(i, j) \in E$ representa uma restrição de precedência, portanto, a tarefa t_i deve terminar sua execução antes que a tarefa t_j comece [THW02].

A tarefa que não tem pais é chamada *tarefa de entrada* e a tarefa que não tem filhos é chamada *tarefa de saída*. Os algoritmos de escalonamento de tarefas neste trabalho requerem somente uma tarefa de entrada e uma tarefa de saída. No caso de existir mais de uma tarefa de saída, estas serão conectadas com custo de comunicação zero para uma pseudotarefa de saída, também com custo zero de computação.

Também, consideramos o seguinte:

- R de r um conjunto recursos heterogêneos conectados;
- *Dados* $v \times v$ uma matriz de comunicação de dados, onde $dados_{i,k}$ é o custo de comunicação, custo necessário para ir desde a tarefa t_i para a tarefa t_k;
- W é uma matriz $v \times r$ de custo de computação, na qual cada $w_{i,j}$ representa o tempo de execução estimado para completar a tarefa t_i no recurso r_j.

Antes do escalonamento, as tarefas geralmente são etiquetadas com a média dos seus custos de execução. A média do custo de execução de uma tarefa t_i é apresentada na Equação 8.1:

$$\overline{w_i} = \sum_{j=1}^{r} w_{i,j}/r \tag{8.1}$$

O custo de transferir dados de um recurso para outro é representado pela matriz B de tamanho $r \times r$. A Equação 8.2 define o custo de comunicação da aresta (i, k), onde $c_{i,k}$ é o custo para transferir dados da tarefa t_i para a tarefa t_k, escalonadas nos recursos r_m e r_n, respectivamente.

$$c_{i,k} = L_m + \frac{dados_{i,k}}{B_{m,n}} \tag{8.2}$$

No caso que as tarefas t_i e t_k sejam escalonadas no mesmo recurso, o valor do $c_{i,k}$ será zero, pois o custo da comunicação dentro de um mesmo recurso é desprezível em comparação com o custo da comunicação entre recursos. Antes do escalonamento, a média dos custos da comunicação, definida pela Equação 8.3, é utilizada para etiquetar as arestas.

$$\overline{c_{i,k}} = \overline{L} + \frac{dados_{i,k}}{\overline{B}} \tag{8.3}$$

onde, $\overline{B}$ representa a média do custo para transferir dados entre recursos e $\overline{L}$ representa a média da latência dos recursos.

Também é necessário definir as funções *earliest execution start time* $EST(t_i, r_j)$ e o *earliest execution finish time* $EFT(t_i, r_j)$, que são derivados de um escalonamento parcial. Para a tarefa de entrada $t_{entrada}$, o valor de EST é:

$$EST(t_{entrada}, r_j) = 0 \tag{8.4}$$

Os valores do EFT e do EST das outras tarefas são calculados recursivamente, iniciando da tarefa de entrada, como é mostrado nas equações 8.5 e 8.6. Para calcular o EFT de uma tarefa t_i, todas as tarefas predecessoras da tarefa t_i devem ter sido escalonadas.

$$EST(t_i, r_j) = \max\left\{avail[j], \max_{t_m \in pred(t_i)}(AFT(t_m) + c_{m,i})\right\} \tag{8.5}$$

$$EFT(t_i, r_j) = w_{i,j} + EST(t_i, r_j) \tag{8.6}$$

onde $prec(t_i)$ é o conjunto de tarefas predecessoras imediatas da tarefa t_i e $avail[j]$ representa o tempo em que o recurso r_j estará disponível para executar uma tarefa, depois de terminar de executar as tarefas atualmente escalonadas no mesmo.

Depois que uma tarefa t_m é escalonada no recurso r_j, o *earliest start time* e o *earliest finish time* de t_m sobre o recurso r_j é igual ao tempo de início atual (*actual start time*) $AST(t_m)$ e o tempo de término atual (*actual finish time*) $AFT(t_m)$ da tarefa t_m, respectivamente. Depois de todas as tarefas do grafo estarem escalonadas, o comprimento do escalonamento (ou seja, o tempo de conclusão total) será o tempo de término da tarefa de saída t_{saida}, ou seja:

$$makespan = AFT(t_{saida}) \tag{8.7}$$

8.2.2 Atributos do Grafo usado pelos Algoritmos de Escalonamento

As tarefas são ordenadas nos algoritmos pelas suas prioridades do escalonamento, que são baseados na classificação das variáveis *upward rank* ou $rank_u$ e *downward rank* ou $rank_d$.

A variável $rank_u$ de uma tarefa t_i é recursivamente definida pela Equação 8.8.

$$rank_u(t_i) = \overline{w_i} + \max_{t_j \in succ(t_i)}(\overline{c_{i,j}} + rank_u(t_j)) \tag{8.8}$$

onde, $succ(t_i)$ é o conjunto de sucessores imediatos da tarefa t_i, $\overline{c_{i,j}}$ é o custo de comunicação médio da aresta (i, j), e $\overline{w_i}$ é o custo de computação médio da tarefa t_i. Cada valor $rank_u$ é calculado recursivamente percorrendo o grafo de tarefas de trás para frente, a partir da tarefa de saída. Para a tarefa de saída t_{saida}, o valor de $rank_u$ é igual a:

$$rank_u(t_{saida}) = \overline{w_{saida}} \tag{8.9}$$

A variável $rank_u(t_i)$ representa o comprimento do "caminho crítico" desde a tarefa t_i para a tarefa de saída, incluindo o custo de computação da tarefa t_i.

Da mesma forma, a variável $rank_d$ de uma tarefa t_i é recursivamente definida pela Equação 8.10:

$$rank_d(t_i) = \max_{t_j \in pred(t_i)}\left\{rank_d(t_j) + \overline{w_j} + \overline{c_{j,i}}\right\} \tag{8.10}$$

onde $pred(t_i)$ é o conjunto de tarefas predecessoras imediatas de t_i. As variáveis $rank_d$ são calculados recursivamente percorrendo o grafo de tarefas para frente, começando na tarefa de entrada do grafo. Para a tarefa de entrada $t_{entrada}$, o valor $rank_d$ é igual a zero.

A variável $rank_d(t_i)$ é a distância mais longa desde a tarefa de entrada até a tarefa de t_i, excluindo o custo de computação da tarefa em si [THW02].

Depois que uma tarefa t_m é escalonada no recurso r_j, o *earliest start time* e o *earliest finish time* de t_m sobre o recurso r_j é igual ao tempo de início atual (*actual start time*) $AST(t_m)$ e o tempo de término atual (*actual finish time*) $AFT(t_m)$ da tarefa t_m, respectivamente.

Depois de todas as tarefas do grafo estarem escalonadas, o comprimento do escalonamento (ou seja, o tempo de conclusão total) será o tempo de término da tarefa de saída t_{saida}. Na Equação 8.11 é definido o *makespan* no caso de ter uma tarefa de saída.

$$makespan = AFT(t_{saida}) \tag{8.11}$$

Se a aplicação tem múltiplas tarefas finais e a convenção de inserir uma pseudotarefa de saída não é aplicada, o comprimento do escalonamento (que também é chamado *makespan*). Na Equação 8.11 é definido o *makespan* no caso de ter várias tarefas de saída.

$$makespan = \max\{AFT(t_{saida})\} \tag{8.12}$$

Nesta parte, os algoritmos de escalonamento escolhidos para serem avaliados são:

- HEFT: este algoritmo usa a técnica de *list scheduling*, é o melhor representante e um dos mais usados pela sua simplicidade na implementação. A partir dele foram feitos diferentes algoritmos de escalonamento para tarefas paralelas;
- CPOP: também usa a técnica de *list scheduling*. Ele agrupa os nós do caminho crítico em um mesmo recurso;
- PCH: inicialmente proposto para trabalhar no *middlware* Xavantes [CMB04], depois foi generalizado no trabalho de Luiz F. Bittencourt*et. al.* [BMCB06] , usa as técnicas de *list scheduling* e *clustering*.

8.2.3 O Algoritmo HEFT

O algoritmo de escalonamento *Heterogeneous Earliest Finish Time* [THW02] utiliza a técnica de *list scheduling* para escalonar tarefas dependentes em sistemas heterogêneos.

Este algoritmo possui duas fases, a fase de atribuir prioridade nas tarefas, e a fase de seleção de recursos para as tarefas em ordem não crescente das prioridades e escalonar cada tarefa selecionada no recurso que minimiza o tempo de finalização da tarefa.

Priorização de tarefas: nesta fase é calculada a "prioridade" de cada tarefa. O valor dessa "prioridade" é igual à variável $rank_u$, cujo cálculo é baseado na média dos custos de computação e custos de comunicação. Depois de computadas as prioridades de cada tarefa é gerada uma lista das tarefas em ordem não crescente. A ordem não crescente de valores $rank_u$ fornece uma ordem topológica das tarefas, que é uma ordem linear que preserva as restrições de precedência.

Seleção de recursos: na fase de seleção de recursos é considerada a possível alocação de uma tarefa t_i a um recurso r_j que minimize o tempo de término (*earliest finish time*). Aqui, o escalonador seleciona a tarefa t_i da lista com maior prioridade, depois para cada recurso $r \in R$ é calculado o EST e EFT de cada tarefa t_i, e finalmente a tarefa é alocada ao recurso r_j que minimiza o EFT da tarefa t_i.

O pseudocódigo do algoritmo HEFT é representado no algoritmo 8.1.

Algoritmo 8.1 O Algoritmo HEFT

1: Calcular o custo de computação das tarefas e o custo de comunicação das arestas com os valores das médias.
2: Calcular $rank_u$ para todas as tarefas percorrendo o grafo, começando na tarefa de saída $task_{saida}$.
3: Ordenar as tarefas em uma lista de escalonamento baseado na ordem não crescente dos valores $rank_u$.
4: **while** (Existem tarefas não escalonadas na lista) **do**
5: Selecionar a primeira tarefa t_i, da lista para o escalonamento.
6: **for** cada recurso r_j no conjunto de recursos ($r_j \in R$) **do**
7: Calcular o valor $EFT(t_i, r_j)$ usando a técnica *scheduling list.*
8: **end for**
9: Alocar a tarefa t_i ao recurso r_j que minimiza o EFT da tarefa t_i.
10: **end while**

8.2.4 O Algoritmo CPOP

O algoritmo de escalonamento *Critical Path On a Processor* [THW02], similar ao algoritmo HEFT, possui as fases de atribuir prioridade às tarefas e seleção de recursos. Ele usa um atributo diferente para estabelecer as prioridades de tarefas e uma estratégia diferente para determinar o melhor recurso para cada tarefa selecionada.

Priorização de tarefas: na fase de atribuir prioridade às tarefas, tanto a variável $rank_u$ quanto a variável $rank_d$ são calculadas para todas as tarefas, usando a média de computação e a média de comunicação.

O algoritmo CPOP usa o caminho crítico (*critical path*) de uma aplicação, o comprimento desse caminho crítico é representado por $|CP|$. A prioridade de cada tarefa é atribuída com a soma de variáveis $rank_u$ e $rank_d$. O $|CP|$ é igual à prioridade da tarefa de entrada. Inicialmente, a tarefa de entrada é a tarefa selecionada e marcada como uma tarefa do caminho crítico. O sucessor imediato (da tarefa selecionada) que tem o valor de maior prioridade é selecionado e é marcado como uma tarefa do caminho crítico. Este processo é repetido até que o nó de saída seja atingido. Em caso de empate, o primeiro sucessor imediato que tem a maior prioridade é selecionado.

É mantida uma fila de prioridade (com a chave de $rank_u + rank_d$) para todas as tarefas prontas em qualquer instante. Um *heap* binário é utilizado para implementar a fila de prioridade, que tem complexidade de tempo $O(log\ v)$ para inserção e exclusão de uma tarefa e $O(1)$ para recuperar a tarefa com a maior prioridade. Em cada etapa, a tarefa com a maior valor $rank_u + rank_d$ é selecionada da fila de prioridade.

Seleção de recursos: o recurso de caminho crítico (*critical-path processor*), (PCP), é o que minimiza os custos de cálculo cumulativo de tarefas no caminho crítico. Se a tarefa selecionada está no caminho crítico, então é escalonada no recurso de caminho crítico, caso contrário, ela é atribuída a um recurso que minimiza o tempo de terminar mais cedo a execução da tarefa. Em ambos os casos, é considerada uma política de escalonamento baseada em inserção. O algoritmo 8.2 representa a execução do Algoritmo CPOP.

Algoritmo 8.2 O Algoritmo CPOP

1: Calcular o custo de computação das tarefas e o custo de comunicação das arestas com os valores das médias.
2: Calcular $rank_u$ para todas as tarefas percorrendo o grafo ascendentemente, começando na tarefa saída $task_{saida}$.
3: Calcular $rank_d$ para todas as tarefas percorrendo o grafo descendentemente, começando na tarefa de entrada $task_{entrada}$.
4: Calcular a prioridade $prioridade(t_i) = rank_u(t_i) + rank_d(t_i)$ para cada tarefa t_i no grafo.
5: $|CP| = prioridade(t_{entrada})$, onde $t_{entrada}$ é a tarefa de entrada.
6: $SET_{CP} = \{t_{entrada}\}$, onde SET_{CP} é o conjunto de tarefas no caminho crítico.
7: $t_k \leftarrow t_{entrada}$.
8: **while** t_k não seja a tarefa saída **do**
9: Selecionar t_j, onde $((t_j \in succ(t_k))$ e $(prioridade(t_j) == |CP|))$.
10: $SET_{CP} = SET_{CP} \cup t_j$.
11: $t_k \leftarrow t_j$.
12: **end while**
13: Selecionar o recurso de caminho crítico (PCP) que minimiza. $\sum_{t_i \in SET_{CP}} w_{i,j}, \forall r_j \in R$.
14: Inicia a fila de prioridade com a tarefa de entrada.
15: **while** (Existem tarefas não escalonadas na fila de prioridade) **do**
16: Selecionar a tarefa de maior prioridade t_i, da fila de prioridade.
17: **if** $t_i \in SET_{CP}$ **then**
18: Alocar a tarefa t_i no PCP.
19: **else**
20: Alocar a tarefa t_i no recurso r_j que minimiza o $EFT(t_i, r_j)$.
21: **end if**
22: Atualizar a fila de prioridade com os sucessores da tarefa t_i, se eles se tornam tarefas prontas.
23: **end while**

8.2.5 O Algoritmo PCH

O algoritmo *Path Clustering Heuristic* (PCH) [BMCB06] é um algoritmo de escalonamento de tarefas dependentes que utiliza as técnicas heurísticas: *list scheduling* e *clustering*. Esta heurística possui duas fases: (a) seleção de tarefas e agrupamento e (b) Seleção de recursos. Na etapa do agrupamento é feito *clusters* de tarefas. Neste contexto, *cluster* é chamado a um agrupamento de tarefas que o algoritmo constrói. A criação de *clusters* tem como finalidade minimizar o *overhead* de comunicação.

Nesta heurística a variável $rank_u$ é levemente modificada. Na Equação 8.13 é apresentada a modificação.

$$rank_u(t_i) = w_i + \max_{t_j \in succ(t_i)} (c_{i,j} + rank_u(t_j)) \tag{8.13}$$

Seleção de tarefas e agrupamento: nesta fase, o algoritmo seleciona tarefas que formarão cada *cluster*, o qual será escalonado no mesmo recurso. A tarefa inicial que será incluída em um *cluster* $cluster_k$ é a tarefa não escalonada com maior prioridade ($rank_u$). A partir dessa tarefa, o algoritmo faz uma busca em profundidade no grafo (DAG), selecionando novas tarefas para serem adicionadas ao *cluster*. A próxima tarefa selecionada para compor $cluster_k$ é a tarefa $t_s \in suc(t_i)$ com o maior $rank_u(t_s) + EST_{t_s}$. Essa soma representa o tamanho do maior caminho que inicia em $t_{entrada}$, passa por t_s e termina em t_{saida}. Além da variável $rank_u$, o EST foi redefinido. Na Equação 8.14 é mostrada a modificação feita no EST:

$$EST(t_i, r_j) = \max\{avail[j], \max_{t_m \in pred(t_i)} (c_{m,i} + w_m + EST_m)\} \tag{8.14}$$

A heurística no inicio do escalonamento, seleciona a primeira tarefa (tarefa de entrada) para adicioná-la ao cluster $cluster_0$. O algoritmo realiza a busca em profundidade, iniciando com a tarefa de entrada, seleciona a tarefa sucessora t_s que tem o maior $rank_{t_s} + EST_{t_s}$ e adicionando-a ao cluster $cluster_0$. A busca em profundidade continua até que a tarefa de saída seja atingida. Nesta primeira parte do escalonamento, é importante ressaltar que, o primeiro *cluster* contém o caminho crítico do grafo inicial. Para a formação do próximo *cluster*, o algoritmo seleciona a tarefa t_i não escalonada com maior prioridade, adicionando-a ao *cluster* $cluster_k$. Partindo dessa tarefa o algoritmo efetua uma busca em profundidade de forma análoga à realizada durante a formação do primeiro *cluster*, porém a busca cessa quando atinge uma tarefa sem sucessores não escalonados, incluindo-a no *cluster*. Então o *cluster* é escalonado e os atributos do grafo recalculados.

No algoritmo 8.3 é apresentado o pseudocódigo da fase descrita acima. A complexidade do algoritmo é $O(n)$.

Algoritmo 8.3 Gerar_agrupamento

1: $t \Leftarrow$ tarefa não escalonada com a maior prioridade, isto é, maior $rank_u$.
2: $cluster \Leftarrow cluster \cup t$.
3: **while** (t tem sucessores não escalonados) **do**
4: $t_{succ} \Leftarrow t_i$ de t com prioridade $rank_{t_i} + EST_{t_i}$.
5: $cluster \Leftarrow cluster \cup t_{succ}$.
6: $t \Leftarrow t_{succ}$.
7: **end while**
8: **return** *cluster*.

Seleção de recursos: nesta fase o algoritmo procura o recurso mais adequado para o *cluster* de tarefas $cluster_k$, gerado na fase anterior. O critério para saber qual recurso é o apropriado é mediante o cálculo do EST do sucessor da última tarefa do *cluster* $cluster_k$.

O calculo do EST do sucessor da última tarefa é feito primeiramente calculando o Earliest Finish Time (EFT) de cada tarefa do *cluster*. O valor EFT da última tarefa do *cluster* $cluster_k$, será somado com o valor EST da tarefa sucessora, e baseado nisso, um recurso será selecionado.

Esse cálculo em cada recurso determina qual recurso terminará a execução do *cluster* $cluster_k$ em menor tempo. Um fator importante a ser levado em conta é que se o recurso contém tarefas da aplicação que esta sendo escalonada, é necessário antes, ordenar as tarefas para que suas precedências não sejam violadas e então efetuar o cálculo dos $EFTs$.

O algoritmo 8.4 mostra essa estratégia de seleção de recursos.

Algoritmo 8.4 Seleciona_melhor_recurso

1: **for all** $r \in recursos$ **do**
2: $escalonamento \Leftarrow$ Insere *cluster* em escalonamento.
3: calcula $EST(escalonamento)$.
4: $tempo_r \Leftarrow EST(sucessor(cluster))$.
5: **end for**
6: **return** recurso r com menor tempo r.

Path Clustering Heuristic: o algoritmo PCH, em Luiz F. Bittencourt *et. al.* [Bit10] e [BMCB06] é composta por um conjunto de fases: (a) seleção de tarefas e agrupamento, (b) seleção de recursos e (c) escalonamento de controladores. A fase (c), seleção de controladores, acontece no *middleware* chamado *Xavantes*, isto porque o algoritmo foi proposto e testado nesse *middleware*. Assim neste cenário, é usada uma estrutura chamada de controlador.

Mas, no nosso cenário de simulações vamos usar as fases de seleção de tarefas e agrupamento, e seleção de recursos. Assim, estas fases vão formar o Path Clustering Heuristic (PCH), que é mostrado no algoritmo 8.5.

O primeiro passo do algoritmo é fazer o cálculo dos atributos iniciais das tarefas. Em seguida, o algoritmo inicia a iteração sobre as tarefas do grafo, criando os *clusters* e escalonando-as até que não restem tarefas não escalonadas. A cada *cluster* escalonado, o algoritmo recalcula os atributos, a prioridade ($rank_u$) EST e EFT das tarefas.

Algoritmo 8.5 O Algoritmo PCH

1: Atribui o DAG G ao sistema homogêneo virtual.
2: Computa os atributos iniciais das tarefas de G.
3: **while** existem tarefas não escalonadas **do**
4: $cluster \Leftarrow gera_agrupamento()$.
5: $recurso \Leftarrow$ seleciona_melhor_recurso($cluster$).
6: Escalona *cluster* em *recurso*.
7: Recalcula, prioridade ($rank_u$), $ESTs$ e $EFTs$.
8: **end while**

8.3 Considerações Finais do Capítulo

Neste capítulo, foram apresentados os algoritmos de escalonamento para tipos de aplicações tanto para tarefas independentes quanto tarefas dependentes.

No caso de aplicações com tarefas independentes: o *WQR*, o *XSufferage* e o *Storage Affinity* são comparados no trabalho [SNCBL04]. O algoritmo *Storage Affinity* e o *WQR* são usados no escalonador *OurGrid*.

Uma avaliação comparativa de 20 diferentes heurísticas é encontrada em [CJSZ08]; entre essas heurísticas, a heurística *Heterogeneous Earliest Finish Time* (HEFT), que é frequentemente citada e utilizada, tem a vantagem de ser simples e produzir escalonamentos de boa qualidade com um *makespan* menor na maioria dos cenários. A heurística *Critical Path On a Processor* (CPOP) foi proposta no mesmo artigo que o HEFT [THW02], apresentando bom desempenho. Finalmente, *Path Clustering Heuristic* (PCH) foi proposto a partir destes algoritmos. No próximo capítulo, é apresentada a comparação de algoritmos para tarefas dependentes usando aplicações e arquiteturas reais.

Capítulo 9

Resultados Experimentais

Neste Capítulo são apresentados e analisados os diferentes resultados experimentais obtidos da implementação e simulação dos algoritmos de escalonamento para tarefas dependentes: (a) *Heterogeneous Earliest Finish Time*, (b) *Critical Path on a Processor* e (c) *Path Clustering Heuristic*, esses algoritmos foram descritos no Capítulo 8.

A simulação dos algoritmos é feita usando o simulador SimGrid, já descrito no Capítulo 7, especificamente, o ambiente de programação do simulador chamado *SimDag*. Este ambiente aceita a modelagem de *aplicações* e arquiteturas mediante formatos específicos.

As aplicações usadas nas simulações foram descritas no Capítulo 6, no entanto cabe ressaltar que essas aplicações foram modeladas em um formato especificado pelo projeto Pegasus, disponibilizados para interesses de pesquisa e baseadas em execuções reais executadas sobre grades.

As arquiteturas para computação em grade usadas na simulação dos algoritmos são especificações de arquiteturas reais existentes para computação em grade: (a) DAS-3, (b) Grid5000 e (c) GridPP; descritas no Capítulo 5. Além dessas arquiteturas foi gerada uma arquitetura que chamaremos *Small Grid*, isso com o intuito de observar o comportamento dos algoritmos com uma pequena quantidade de recursos computacionais. Esta arquitetura é apresentada na Figura 9.1.

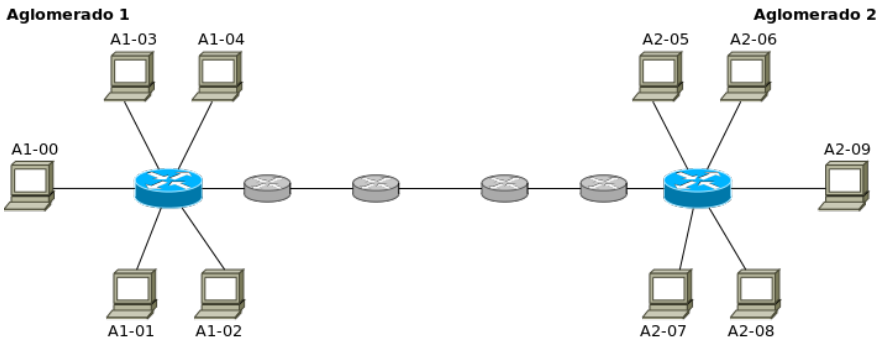


Figura 9.1: *Representação da arquitetura* Small Grid *usada nos experimentos. Cada roteador representa um aglomerado e enlace para o outro. O total de recursos computacionais é dez.*

Na Tabela 9.1 são mostrados os poderes computacionais dos recursos do *Small Grid*. Temos duas abordagens: na primeira: a abordagem homogênea, onde todos os processadores possuem o mesmo poder computacional de $5GFlops$ e; a segunda: a abordagem heterogênea, onde os processadores possuem diferente poder computacional. Nas duas abordagens o poder computacional total é $50GFlops$.

Id	Poder Computacional (*GFlops*)	
	Homogêneo	Heterogêneo
A1-00	5,00	1,00
A1-01	5,00	2,00
A1-02	5,00	3,00
A1-03	5,00	4,00
A1-04	5,00	5,00
A2-05	5,00	5,00
A2-06	5,00	6,00
A2-07	5,00	7,00
A2-08	5,00	8,00
A2-09	5,00	9,00

Tabela 9.1: *Conjunto de recursos computacionais da arquitetura* Small Grid. *A primeira fila representa o* Id dos recursos*, a segunda e a terceira fila representam os poderes computacionais dos recursos da arquitetura homogênea e heterogênea, respectivamente.*

9.1 Análise das Aplicações

Para cada aplicação temos um conjunto de "traços de execução" (*traces*), os quais são divididos por instâncias, por exemplo, o conjunto de execuções da aplicação Montage com diferentes tamanhos nos dados de entrada. Um traço de execução, neste trabalho, é dado por um conjunto de tarefas que pertencem a uma aplicação. Esse conjunto é descrito como um grafo acíclico dirigido que precisa por cada tarefa: (i) o tempo de execução necessário para processar essa tarefa e (ii) a ordem em que essa tarefa será executada.

Para analisar o conjunto de aplicações distinguimos dois tipos: (i) Aplicação Regular e (ii) Aplicação Irregular. Justificamos esse fato como segue: dada uma aplicação podemos agrupar aos conjuntos de traços de execução pelo número de tarefas que possui, por exemplo, os traços de execução da aplicação Montage com 50, 100, 200, ..., 1000 tarefas. A soma dos tempos de execução das tarefas de um traço é denominada "carga do trabalho" (*workload*).

Aplicações regulares são aquelas nas quais a diferença das cargas do trabalho de diferentes traços de execução produzidas com o mesmo número de tarefas é baixo; de outra forma a aplicação é irregular. A ideia de agrupar pelo número de tarefas é observar se os traços de execução dependem dos dados de entrada, esses traços com um determinado número de tarefas devem corresponder às execuções com dados de entrada do mesmo tamanho. Desta forma, se a execução depende do conjunto de dados de entrada, podemos dizer que a aplicação é regular. Um exemplo clássico neste caso é a multiplicação de matrizes. No entanto, no caso do problema da satisfabilidade (SAT) onde os dados de entrada não influem no comportamento da estrutura da aplicação é um exemplo de aplicação de tipo irregular.

Os principais motivos para esta distinção são: em aplicações regulares é possível comparar escalabilidade no escalonamento dos algoritmos, acrescentando gradualmente tarefas. Esse tipo de comparação pode não ter sentido em aplicações irregulares, porque neste caso a carga de trabalho pode não necessariamente ser incrementada com o número de tarefas.

Para identificar uma aplicação regular, denotamos: T_A como os "traços de execução" de uma aplicação com n tarefas.

Definição A carga de trabalho $W(T_A)$ de um traço de execução de uma aplicação T_A de tamanho n é dado por:

$$W(T_A) = \sum_{i=1}^{n} w_i \tag{9.1}$$

onde, w_i é o tempo de execução da tarefa i.

Definição Dada uma aplicação A, se $T_{A,n}$ é o conjunto de traços de tamanho n, a média da carga do trabalho $W(A,n)$ de uma aplicação de cada instância de tamanho n é dado por:

$$W(A,n) = \frac{1}{|T_{A,n}|} \sum_{T \in T_{A,n}} W(T) \tag{9.2}$$

Definição Dada uma aplicação A, chamamos a aplicação de irregular se $\exists n, m$ com $n < m$ tal que $W(A,n) > W(A,m)$.

Essa definição pode também ser obtida calculando o desvio padrão sobre a carga de trabalho, dado um número de tarefas fixo. Se esse desvio padrão é grande, é muito provável que a aplicação seja irregular.

Na Figura 9.2 são mostrados os valores $\log(\frac{W(A,n)}{W(A,50)})$ para cada aplicação. Nesta figura podemos entender que a aplicação Epigenomics apresenta um comportamento diferente, dado que possui pelo menos dois valores i, j tal que $W(A,i) > W(A,j)$, com $i < j$. Portanto, concluímos que essa aplicação é irregular.

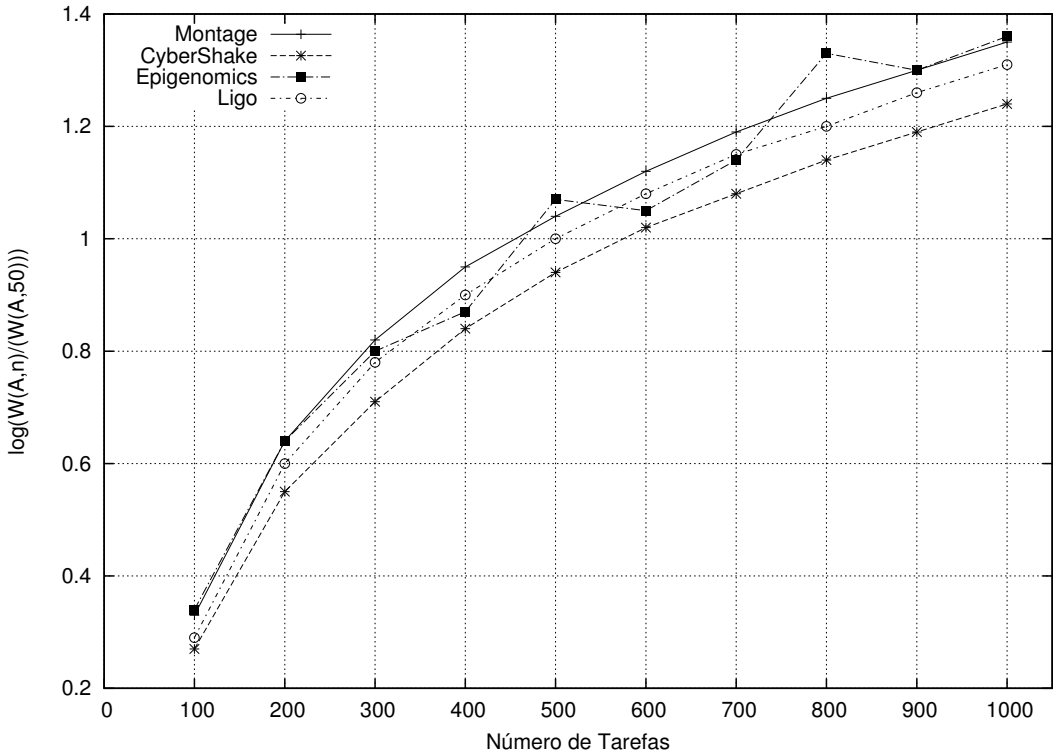


Figura 9.2: *Representação dos comportamentos das aplicações.*

Essas observações foram calculadas analisando o desvio padrão das cargas de trabalho. Portanto

para uma aplicação com tamanho fixo i, denotamos:

$$\sigma(A,i) = \sqrt{\frac{\sum_{j=1}^{|T_{A,i}|}(w_{i,j} - W(A,i))^2}{|T_{A,i}| - 1}} \tag{9.3}$$

calculando esse valor para Epigenomics, foi obtido o $\sigma(A, 50) = 35947.96$ e para Montage, $\sigma(A, 50) = 1.46$. Portanto, em aplicações irregulares o valor σ será alto.

9.1.1 Análise das Estruturas das Aplicações

No Capítulo 6 foram descritas as aplicações e as estruturas de cada uma. Nesta seção vamos analisar com maior detalhe essas estruturas, para isso consideramos duas métricas: (i) Caminho Crítico e (ii) Sincronização Crítica. Caminho Crítico é usualmente usado para denotar o conjunto de tarefas críticas do grafo que devem ser executadas em sequência. A Sincronização Crítica corresponde ao número máximo de sincronização que acontece sobre uma tarefa no grafo.

Todas as aplicações que foram usadas neste trabalho têm o comprimento do caminho crítico constante em todos os traços de execução. Assim, as aplicações Montage, CyberShake, Epigenomics e Ligo, possuem **nove**, **quatro**, **oito** e **seis** tarefas no caminho crítico, respectivamente.

No caso da sincronização crítica, na aplicação Montage, acontece em dois níveis, o *mConcatFit* e o *mImgTbl*, o número de tarefas a serem sincronizadas depende da quantidade total de tarefas no traço de execução, esse número é mostrado na Tabela 6.2.

A aplicação CyberShake, também possui dois níveis onde acontece a sincronização crítica, o *ZipSeis* e o *ZipPSA*. A quantidade de tarefas para sincronizar são as mesmas em ambos os casos e essa quantidade é igual ao número de tarefas no nível *SeismogramSynthesis*, apresentado na Tabela 6.3.

Na aplicação Epigenomics a sincronização crítica acontece somente no nível *mapMerge*, como mostrado na Figura 6.3 e o número de tarefas para serem sincronizadas são quatro por cada tarefa de sincronização crítica, a quantidade de tarefas de sincronização crítica é apresentada na Tabela 6.4.

Finalmente na aplicação Ligo, sincronização crítica acontece em dois níveis, nas duas ocorrências do *Thinca*, como é mostrado na Figura 6.4, neste caso o número de tarefas que serão sincronizadas varia de acordo à quantidade de tarefas nos níveis *Inspiral*.

9.2 Resultados Experimentais

Na Tabela 9.2 são descritas as principais questões que direcionam as nossas simulações. Essas questões são organizadas por critérios de avaliação selecionados. Além disso, é indicada por cada critério, a métrica que será usada e descreve como serão feitos os experimentos.

O primeiro critério é o **Desempenho**: se o nosso conjunto de experimentos são exemplos representativos de paralelismo, podemos pensar que o algoritmo com melhor desempenho deveria ser escolhido como padrão em um sistema de recomendação, se estivermos sobre plataformas semelhantes.

Também consideramos a **Escalabilidade**: este critério de avaliação servirá para entender como os algoritmos de escalonamento se comportam na medida em que o número de tarefas é aumentado.

Critério	Métricas	Questões & Configurações
Desempenho	A soma total dos *Makespans*	Em diferentes grades se quer encontrar o algoritmo com melhor desempenho por grade e por aplicação (tipo de aplicação), uma noção geral do algoritmo com o melhor desempenho.
Escalabilidade	Média do *Makespan* pelo número de tarefas e nós da grade	A avaliação é feita para aplicações regulares sobre todas as grades. Com o intuito de encontrar como é o desempenho do algoritmo de escalonamento sobre diferentes números de tarefas e processadores.
Adaptabilidade	Taxa entre o total do *Makespan* por grade e por aplicação	A avaliação é feita por aplicação sobre as grades. O intuito é identificar quais algoritmos são mais adaptativos sobre diferentes arquiteturas.
Distribuição da Carga do Trabalho	Número de tarefas por nós da grade e tempo necessário para a comunicação entre elas	Esta avaliação é feita para todas as grades em uma aplicação. O intuito é entender qual algoritmo é o melhor distribuidor da carga de trabalho, dependendo da heterogeneidade.

Tabela 9.2: *Critérios para a comparação dos algoritmos.*

O terceiro critério que será considerado a **Adaptabilidade**: o nosso interesse é entender se um algoritmo de escalonamento mantém um bom desempenho sobre diferentes plataformas para grade.

O último critério é a **Distribuição da carga do trabalho**: o objetivo aqui é entender porque um algoritmo tem ou não tem bom desempenho dependendo da aplicação e da arquitetura da grade.

9.2.1 Resultados

Nos experimentos foram simuladas: (i) as instâncias de cada traço de execução (das aplicações), (ii) as especificações das arquiteturas (baseada na especificação estabelecida pelo projeto SimGrid) e (iii) os algoritmos de escalonamento para tarefas dependentes.

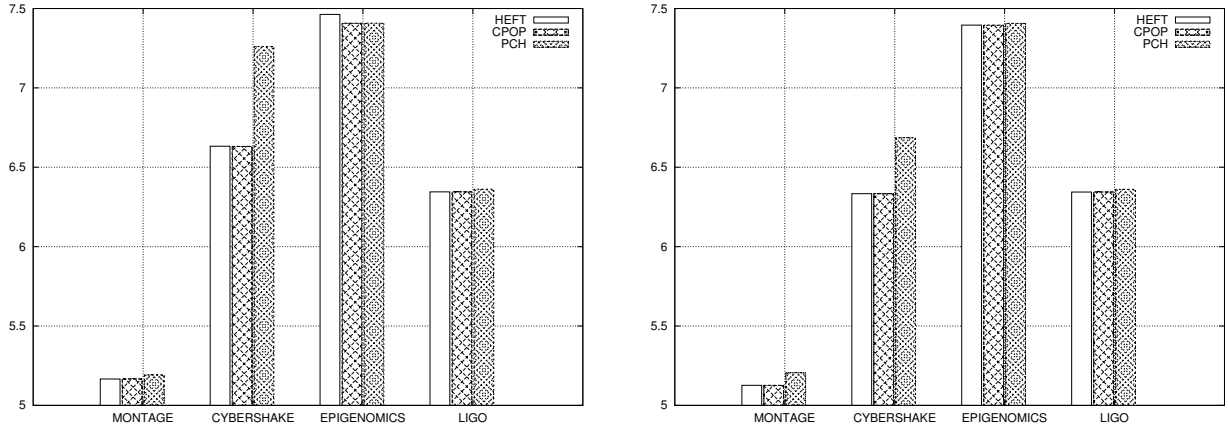
Nesta subseção é apresentada a análise das simulações feitas, de acordo com os critérios apresentados na Tabela 9.2.

Dividimos cada critério em dois cenários: (i) o cenário da arquitetura *Small Grid*, onde temos duas instâncias de uma pequena grade com 10 recursos computacionais, uma instância homogênea e outra heterogênea; e (ii) o cenário das grades de larga escala, DAS-3, Grid5000 e GridPP.

Desempenho

Nas Figuras 9.3 e 9.4 são representados os resultados do critério **Desempenho**. O eixo y representa a medida em escala logarítmica da soma total dos *Makepans* que cada algoritmo obteve sobre cada arquitetura. O eixo x representa as aplicações. O intuito desse critério é encontrar o algoritmo com melhor desempenho por grade e por aplicação.

Nas Figuras 9.3a e 9.3b são mostrados os resultados de desempenho para a arquitetura *Small Grid*. Na arquitetura *Small Grid* homogênea, os algoritmos HEFT e CPOP apresentam um desem-



(a) Small Grid *Homogênea* **(b)** Small Grid *Heterogênea*

Figura 9.3: *Resultados do desempenho avaliado na arquitetura* Small Grid.

penho bom em todas as aplicações com exceção de Epigenomics, nesse caso, o algoritmo CPOP ganha por uma porcentagem de $11,9\%$. O algoritmo PCH não apresentou um bom desempenho em nenhuma aplicação, mas foi muito próximo aos outros, com uma exceção na aplicação CyberShake, apresentando um desempenho muito ruim. Esse fato é devido à estrutura da aplicação, dado que possui um considerável número de tarefas de sincronização crítica e isso não é bem apropriado pelo agrupamento que o algoritmo PCH cria quando existe pequena quantidade de recursos.

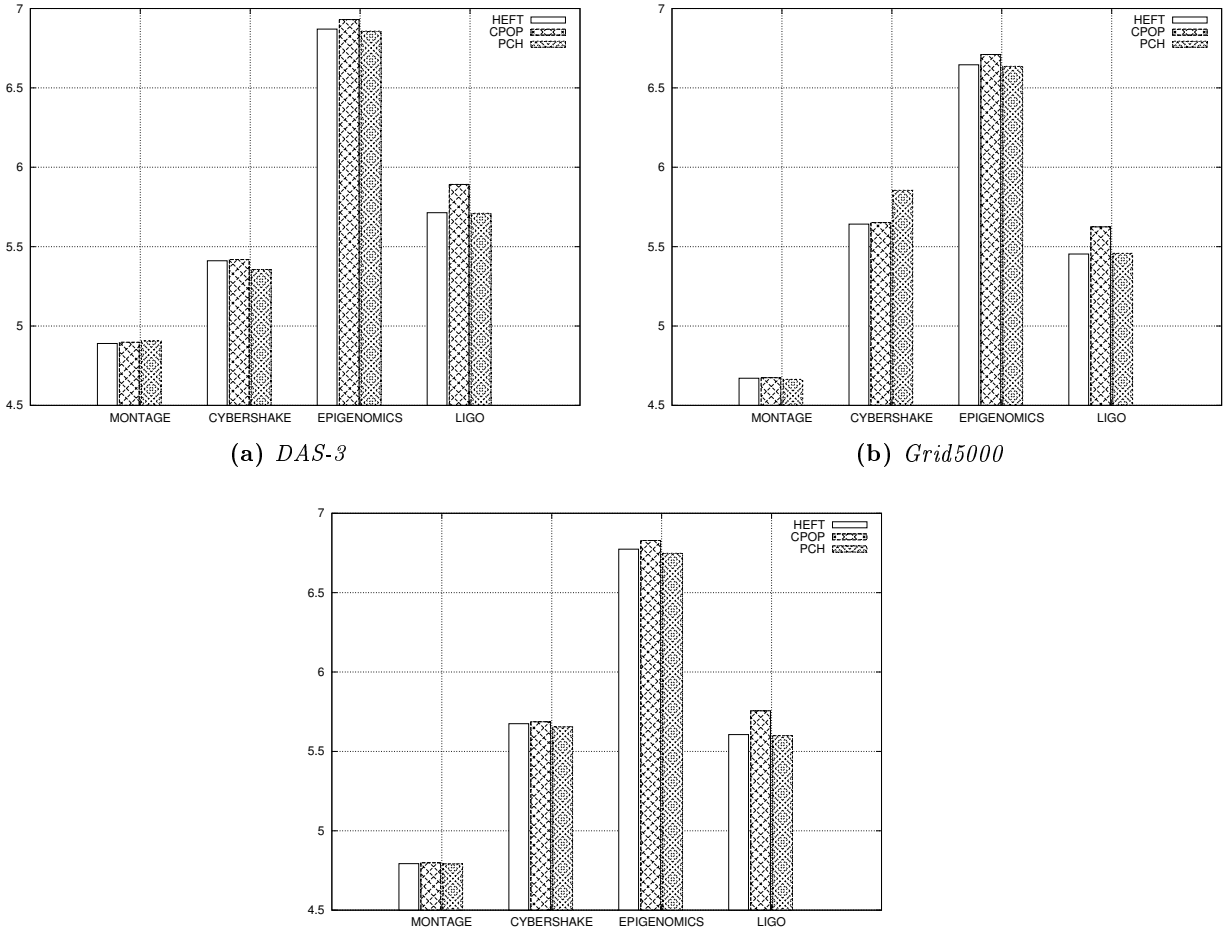
Na arquitetura heterogênea, Os algoritmos HEFT e CPOP apresentam um bom desempenho, com uma diferença média 0.07%, quase desprezível. Entretanto, o algoritmo PCH apresentou um comportamento similar à arquitetura homogênea, isso confirma o fato que o comportamento do algoritmo PCH não apresenta um bom desempenho sobre aplicações com grande número de tarefas críticas e recursos limitados.

Os resultados das simulações das arquiteturas de larga escala são mostradas nas Figuras 9.4a, 9.4b e 9.4c.

No caso da arquitetura DAS-3, mostrado na Figura 9.4a, o algoritmo PCH destacou, mostrando o melhor desempenho nas aplicações: CyberShake, Epigenomics e Ligo, esse fato confirma novamente que acrescentando o número de recursos e a largura de banda, o agrupamento de tarefas é bem mais adequado. O algoritmo HEFT apresenta melhor desempenho somente na aplicação Montage, no entanto, sempre se mantém quase próximo ao algoritmo que apresentou melhor desempenho, com uma diferença quase desprezível. No caso do algoritmo CPOP, obteve o pior desempenho nas aplicações: CyberShake, Epigenomics e Ligo, mostrando um desempenho muito ruim na aplicação Ligo.

Na a arquitetura Grid5000, mostrado na Figura 9.4b . Tanto o algoritmo HEFT quanto o algoritmo PCH obtiveram melhor desempenho. No caso do algoritmo HEFT foi melhor nas aplicações CyberShake e Ligo, enquanto que o algoritmo PCH foi melhor para Montage e Epigenomics. Cabe ressaltar o fato que o algoritmo PCH obteve o pior desempenho na aplicação CyberShake, esse comportamento foi similar para a arquitetura *Small Grid*.

Finalmente, na arquitetura GridPP, mostrado na Figura 9.4c, o algoritmo PCH atingiu melhor desempenho em todas as aplicações. Cabe dizer que a diferença entre esse algoritmo e o segundo melhor algoritmo, nesse caso o algoritmo HEFT, é quase desprezível.



(a) *DAS-3*

(b) *Grid5000*

(c) *GridPP*

Figura 9.4: *Resultados do desempenho avaliado nas arquiteturas de larga escala.*

De forma geral, para a arquitetura *Small Grid*, o algoritmo CPOP obteve melhor desempenho em 75% dos casos, enquanto o algoritmo HEFT obteve o melhor desempenho em 25% dos casos. Por outro lado, o algoritmo PCH obteve o pior desempenho em $87,5\%$ dos casos e o algoritmo HEFT em $12,5\%$ dos casos.

Com respeito às arquiteturas de grande porte, o algoritmo HEFT foi o melhor algoritmo em 25% dos casos e o algoritmo PCH foi o melhor em 75% dos casos. O pior desempenho foi obtido pelo algoritmo CPOP com $83,3\%$ dos casos e $16,6$ pelo algoritmo PCH.

O critério **Desempenho** da uma visão geral do comportamento dos algoritmos, como foi mostrado em casos onde o cenário seja muito variado, isto é, a plataforma (ou arquitetura) seja cambiante e os tipos de aplicações sejam muito deferentes, esse critério pode dar uma noção de qual algoritmo pode ser o mais adequado no uso de maneira geral.

Escalabilidade

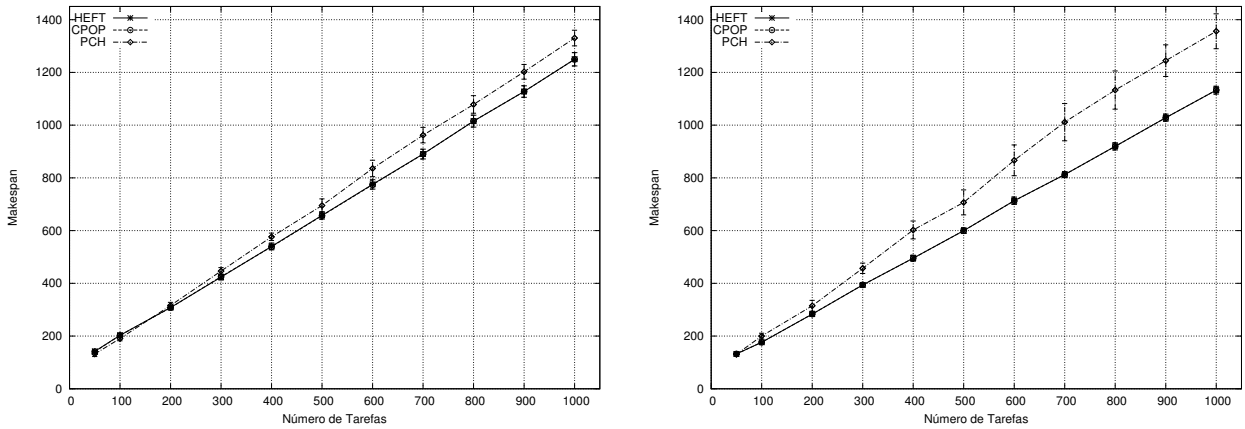
A análise do critério de **Escalabilidade** é dividida por aplicações, focado especificamente para as aplicações regulares, dado que para aplicações irregulares é bem mais difícil medir esse critério.

Nas Figuras 9.5 e 9.6 são mostrados os resultados de simular os algoritmos de escalonamento com a aplicação Montage, nas Figuras 9.7 e 9.9 são mostrados os resultados da aplicação CyberShake e

nas Figuras 9.10 e 9.11 são mostrados os resultados da aplicação Ligo. Onde, o eixo y representa o *Makepan* (tempo de termino da execução da ultima tarefa) e o eixo x representa o número total de tarefas. A aplicação Epigenomics é irregular, portanto, não é adequada para analisar escalabilidade, pois, nesse tipo de aplicações, os resultados não representam informação significativa.

Nas Figuras 9.5 e 9.6 são apresentados os resultados da simulação da aplicação **Montage** sobre cada uma das arquiteturas.

No caso da arquitetura *Small Grid* os resultados são apresentados nas Figuras 9.5a e 9.5b para a arquitetura Homogênea e Heterogênea, respectivamente. Os algoritmos HEFT e CPOP tanto na instância homogênea quanto na heterogênea apresentam um desempenho muito parecido com uma diferença quase desprezível, na medida em que o número de tarefas é maior, a gráfica cresce progressivamente. Por outro lado, o algoritmo PCH somente apresentou um desempenho parecido aos algoritmos HEFT e CPOP com 50 tarefas, no entanto, em medida que as tarefas foram acrescentadas, o desempenho do algoritmo piorava em comparação aos outros algoritmos.



(a) Small Grid *Homogênea* **(b)** Small Grid *heterogênea*

Figura 9.5: *Resultados da simulação da aplicação Montage avaliada sobre a arquitetura* Small Grid.

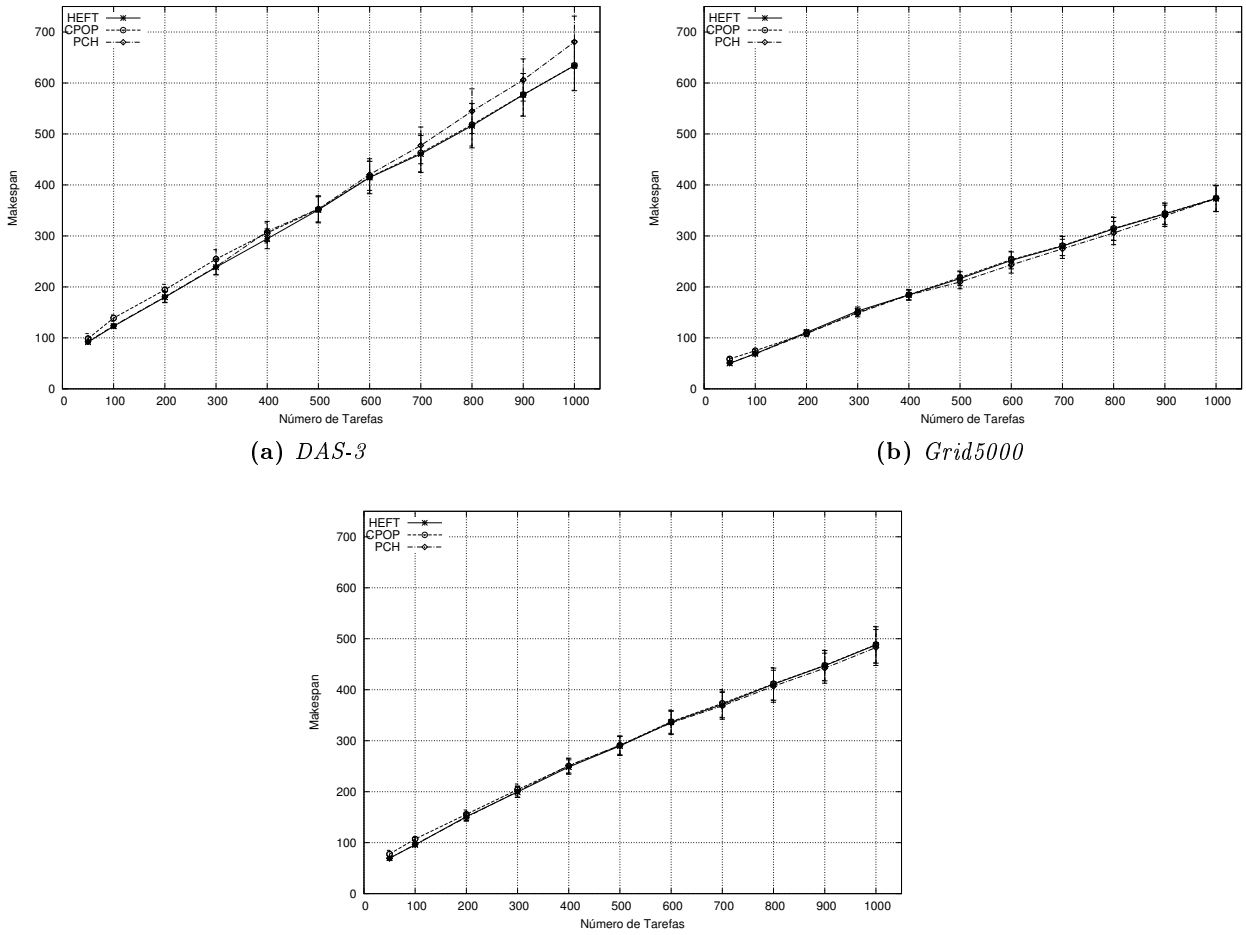
A comparação dos algoritmos executando a aplicação Montage sobre arquiteturas de larga escala são apresentadas nas Figuras 9.6a, 9.6b e 9.6c.

Na arquitetura DAS-3, mostrado na Figura 9.6a, o algoritmo HEFT apresentou melhor desempenho em todos os casos. Entretanto, o algoritmo CPOP apresentou um desempenho ruim para 50, 100, 200, 300 e 400 tarefas e para 500, 600, 700, 800, 900 e 1000 o algoritmo obteve melhoria, mostrando um desempenho próximo ao algoritmo HEFT. No caso do algoritmo PCH apresentou um comportamento um tanto instável, para quantidade de tarefas pequenas como 50, 100, 200 e 300 o algoritmo apresentou um desempenho quase similar ao algoritmo HEFT, mas a partir de 400 tarefas o algoritmo começa apresentar um desempenho ruim, este fato pode ser explicado porque a arquitetura DAS-3 possui 272 recursos computacionais.

Na arquitetura Grid5000, mostrado na Figura 9.6b, os algoritmos apresentam um comportamento muito similar, com pequenas diferenças favoráveis ao algoritmo PCH nas instâncias com 500, 600, 700, 800 e 900 tarefas.

No caso da arquitetura GridPP, mostrado na Figura 9.6c, os algoritmos apresentam um comportamento similar, com diferenças quase desprezíveis.

Os resultados da simulação dos algoritmos de escalonamento com a aplicação Montage sobre as



(a) *DAS-3*

(b) *Grid5000*

(c) *GridPP*

Figura 9.6: *Resultados da simulação da aplicação Montage avaliada sobre as arquiteturas de grande porte.*

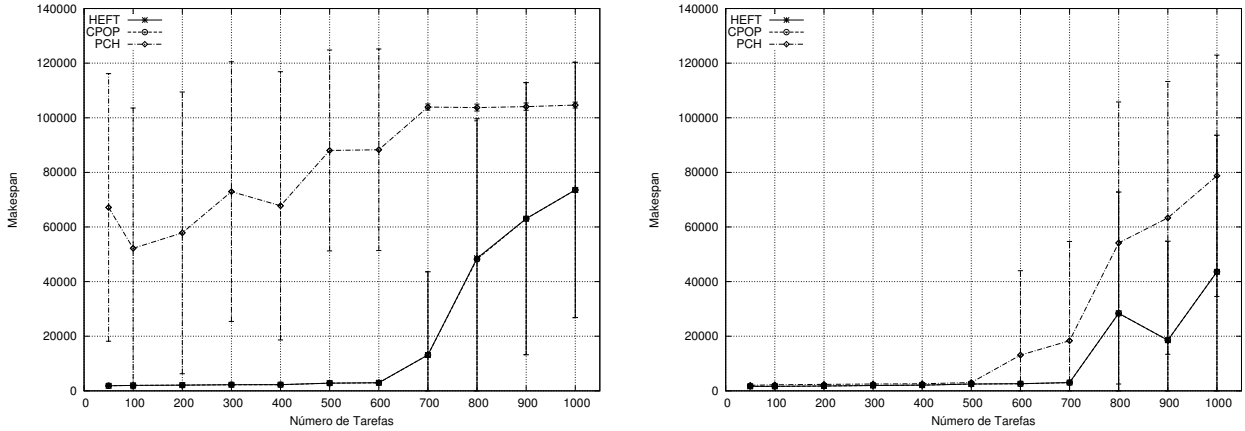
arquiteturas de larga escala, confirma que a técnica usada pelo algoritmo PCH, de agrupar tarefas, tem um efeito positivo quando se possui os recursos suficientes, no caso de ter aplicações com grande número de tarefas de sincronização crítica. Por outro lado, podemos destacar o comportamento do algoritmo HEFT, que apesar da sua simplicidade no escalonamento apresenta um bom desempenho na maioria dos casos.

Nas Figuras 9.7 e 9.9 são apresentados os resultados da simulação da aplicação **CyberShake**.

Na arquitetura *Small Grid* homogênea, mostrada na Figura 9.7a, podemos ver que o algoritmo PCH apresentou um comportamento muito instável para 50, 100, 200, 300, 400, 500 e 600 tarefas, vide Figura 9.7a, a instabilidade é mostrada por um desvio padrão grande. Esse mesmo comportamento acontece com os algoritmos HEFT e CPOP para 700, 800, 900 e 1000 tarefas. Nessa figura podemos entender quão complexo é escalonar uma estrutura como da aplicação CyberShake. Nesse caso o algoritmo PCH apresentou pior desempenho em comparação aos outros algoritmos.

Na Figura 9.7b apresenta um resultado estável nos três algoritmos até 500 tarefas. Depois desse número os algoritmos apresentam uma instabilidade piorando no comportamento e nos desvios padrões. Novamente, o algoritmo PCH apresentou um desempenho muito ruim em relação aos outros algoritmos.

O fato da instabilidade e do grande desvio padrão nos resultados nessas figuras é principalmente por um escalonamento não apropriado por parte dos algoritmos, o qual em alguns casos apresenta



(a) Small Grid *Homogênea* **(b)** Small Grid *heterogênea*

Figura 9.7: *Resultados da simulação da aplicação CyberShake avaliada sobre a arquitetura* Small Grid.

Makespans razoáveis e em outros *Makepans* muito grandes, como é mostrado na Figura 9.8. Por exemplo, podemos ver que no caso da primeira simulação, os algoritmos HEFT e CPOP atingiram um *Makespan* razoável, no entanto, o algoritmo PCH não.

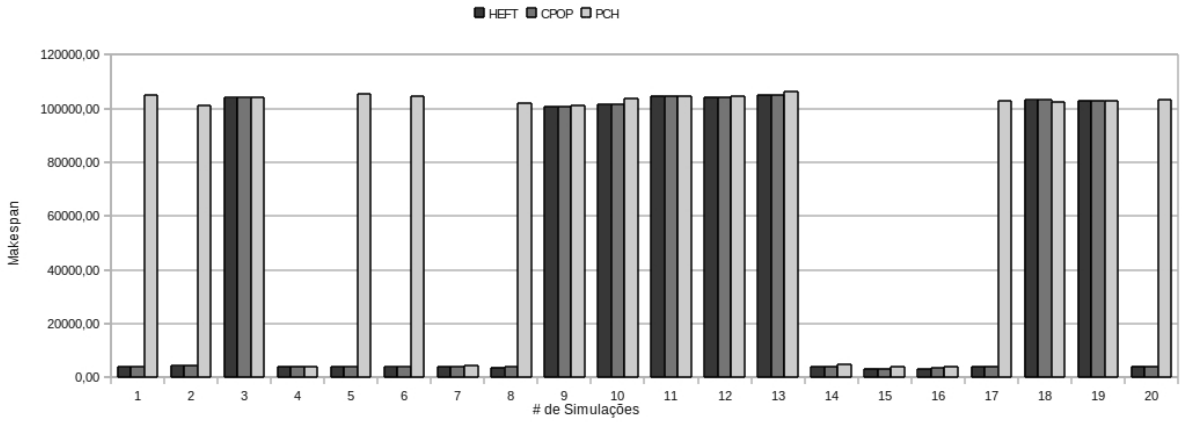


Figura 9.8: *Resultados das simulações de 20 instâncias da Aplicação CyberShake na arquitetura* Small Grid *heterogênea.*

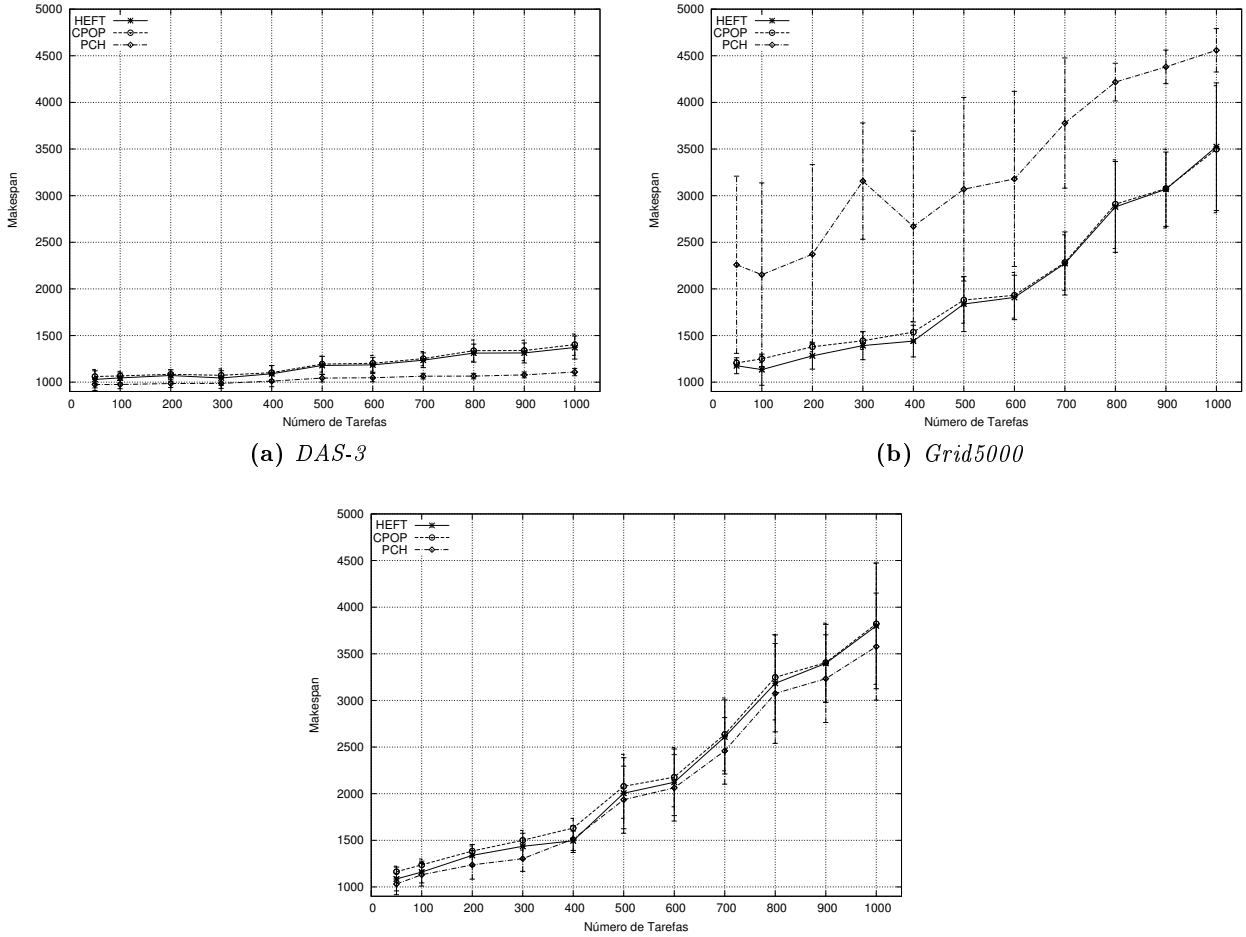
Os resultados das simulações das arquiteturas DAS-3, Grid5000 e GridPP são apresentadas nas Figuras 9.9a, 9.9b e 9.9c, respectivamente.

No caso da arquitetura DAS-3, mostrado na Figura 9.9a, o algoritmo que apresentou melhor desempenho em todos os casos foi o PCH, mesmo com a complexidade da estrutura da aplicação CyberShake o escalonamento do PCH atingiu bom desempenho, enquanto o algoritmo que apresentou pior desempenho foi o algoritmo CPOP. Aqui, nesta simulação podemos ver quão instáveis são esses algoritmos, e quão estável é o algoritmo HEFT, dado que o HEFT conserva seu comportamento obtendo o melhor ou o segundo melhor desempenho em relação aos outros algoritmos.

Na arquitetura Grid5000, mostrado na Figura 9.9b, o algoritmo PCH mostrou um comportamento instável, apresentando o pior desempenho em todos os casos, por outro lado os algoritmos HEFT e CPOP apresentam um desempenho quase similar, no entanto, o algoritmo HEFT melhora o desempenho em comparação com o algoritmo CPOP com uma mínima diferença.

Na arquitetura GridPP, como é mostrado na Figura 9.9c, o algoritmo PCH apresentou o desem-

penho melhor, enquanto o algoritmo HEFT foi segundo melhor e o algoritmo CPOP obteve o pior desempenho.



(a) *DAS-3*

(b) *Grid5000*

(c) *GridPP*

Figura 9.9: *Resultados da simulação da aplicação CyberShake avaliada sobre as arquiteturas de grande porte.*

Nos resultados apresentados na simulação dos algoritmos com a aplicação CyberShake sobre as diferentes arquiteturas, novamente podemos entender que a técnica de agrupamento não é adequada quando se possuem pequenas quantidades de recursos, como foram mostradas nos casos da arquitetura *Small Grid*. Mesmo sendo a estrutura da aplicação CyberShake complexa para quantidades de recursos maiores o agrupamento é bem mais adequado, prova disso é quando o algoritmo PCH obteve melhor desempenho nas arquiteturas DAS-3 e GridPP.

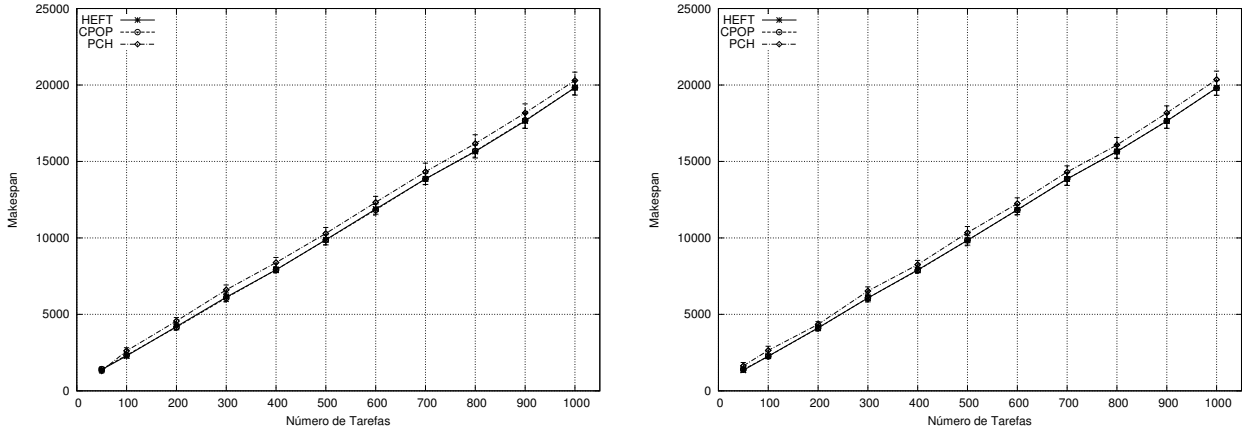
Nas Figuras 9.10 e 9.11 são apresentados os resultados da simulação da aplicação **Ligo**.

Na arquitetura *Small Grid* homogênea, como é mostrado na Figura 9.10a, os algoritmos apresentam um desempenho bem similar, com uma diferença mínima não favorável ao algoritmo PCH. Os algoritmos HEFT e CPOP apresentam um comportamento quase similar.

Na arquitetura *Small Grid* heterogênea, mostrada na Figura 9.10b, o comportamento dos algoritmos é quase próximo à arquitetura *Small Grid* homogênea. Os algoritmos apresentam um comportamento similar. Mantém-se a pequena diferença não favorável do algoritmo PCH.

Os resultados para as arquiteturas de larga escala são apresentados na Figura 9.11.

No caso da arquitetura DAS-3, mostrado na Figura 9.11a, o algoritmo CPOP apresentou um



(a) Small Grid *Homogênea* **(b)** Small Grid *heterogênea*

Figura 9.10: *Resultados da simulação da aplicação Ligo avaliada sobre a arquitetura* Small Grid.

comportamento instável com um desempenho muito ruim em comparação aos outros algoritmos. Tanto o algoritmo HEFT quanto o algoritmo PCH apresentaram um desempenho quase similar, com um ganho pequeno por parte do algoritmo PCH em alguns casos.

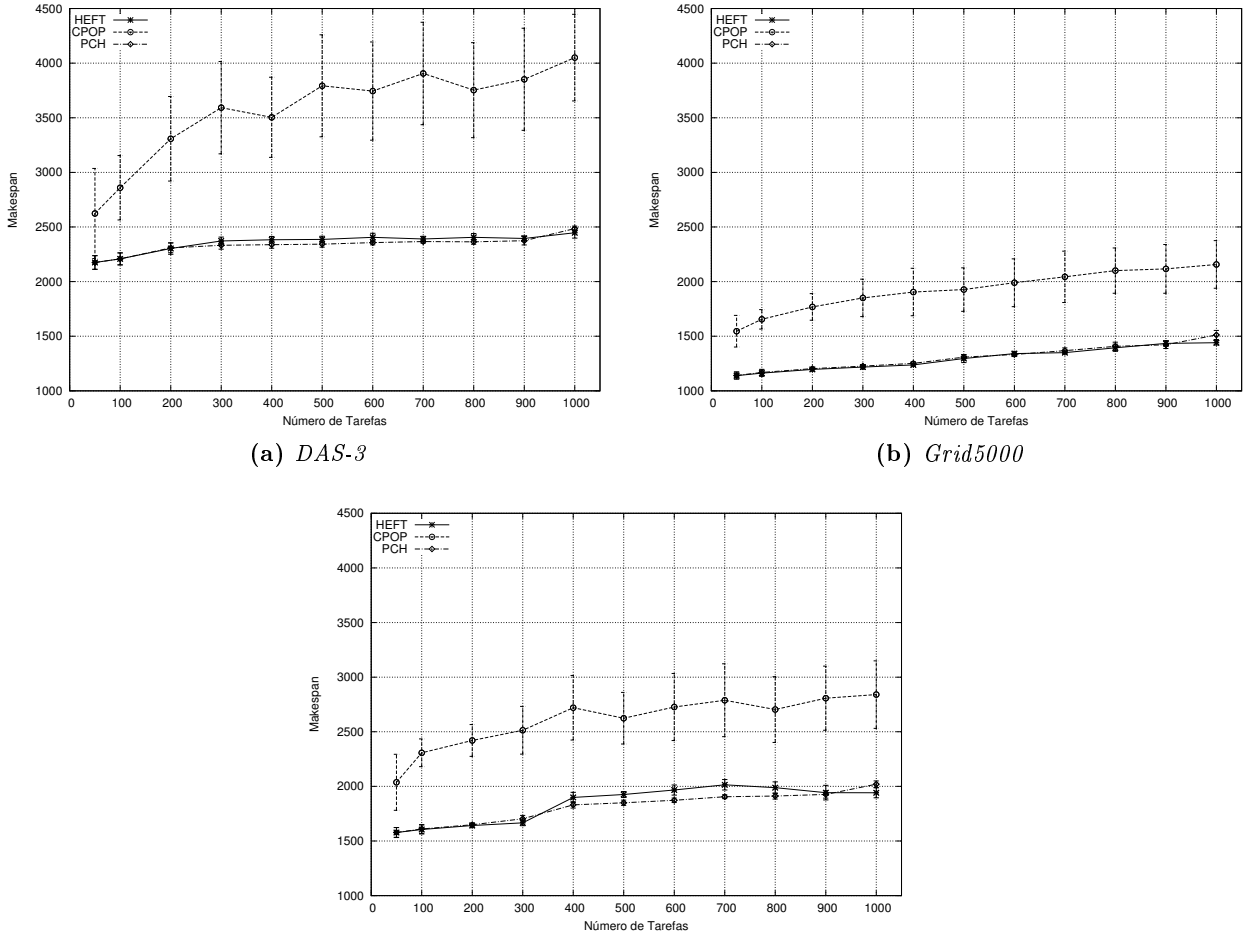
No caso da arquitetura Grid5000, mostrado na Figura 9.11b, o comportamento do algoritmo CPOP foi similar que no caso anterior, não apresentou bom desempenho em relação aos outros algoritmos. Enquanto, os algoritmos HEFT e PCH mostraram um desempenho bem similar, com uma diferença minimamente considerável na aplicação com 1000 tarefas, onde o algoritmo HEFT obteve melhor desempenho.

No caso da arquitetura GridPP, mostrado na Figura 9.11c, novamente o algoritmo CPOP apresentou um desempenho ruim em relação aos algoritmos HEFT e PCH. No entanto, os algoritmos HEFT e CPOP apresentam um desempenho similar, com pequenas diferenças a partir de 400 tarefas.

O critério de **Escalabilidade** apresenta um maior detalhe do comportamento dos algoritmos de escalonamento. Podemos destacar que o desempenho do algoritmo PCH não é bom em cenários onde a quantidade de recursos é pequena, isso pode ser observado nos casos das avaliações sobre a arquitetura *Small Grid*, onde em quase todas as simulações o algoritmo apresentou um desempenho muito ruim, no entanto, para arquiteturas de larga escala o algoritmo melhorou consideravelmente. O algoritmo CPOP apresentou bom desempenho somente nos casos da arquitetura *Small Grid*, nas arquiteturas de larga escala, o algoritmo não apresentou um bom desempenho, prova disso é o caso da aplicação Ligo, onde o algoritmo mostrou um desempenho baixo em relação aos outros algoritmos, esse fato é devido à grande dependência das tarefas com o caminho crítico, dado que o algoritmo CPOP escalona as tarefas do caminho crítico no melhor processador, essa abordagem nem sempre será adequada em aplicações com estruturas similares à aplicação Ligo. O algoritmo HEFT continua mantendo sua característica de algoritmo estável.

Adaptabilidade

A avaliação do critério de **Adaptabilidade** é apresentada nas Tabelas 9.3, 9.4 e 9.5, para os algoritmos HEFT, CPOP e PCH, respectivamente. Esses valores são baseados sobre o total do *Makepan* por grade e aplicação, esse valor é o *Makepan* obtido na grade i dividido pelo *Makepan*



(a) *DAS-3*

(b) *Grid5000*

(c) *GridPP*

Figura 9.11: *Resultados da simulação da aplicação Ligo avaliada sobre as arquiteturas de grande porte.*

obtido na grade j, assim, $c(i,j) = \frac{Makespan(i)}{Makespan(j)}$, onde i e j são os números de processadores e $i < j$. As arquiteturas para grade consideradas neste critério são DAS-3, Grid5000 e GridPP.

A Tabela 9.3, apresenta quão adaptável é o algoritmo HEFT por cada aplicação se é mudada de uma arquitetura para outra. Para a aplicação Montage, como é mostrado na segunda coluna, mudar a arquitetura DAS-3 para a arquitetura Grid5000, é conveniente, dado que a soma total dos *Makespans* obtidos na arquitetura DAS-3 é maior que a soma dos *Makespans* obtidos sobre arquitetura Grid5000. Por outro lado, como é mostrado na terceira coluna, mudar da arquitetura Grid5000 para a arquitetura GridPP, não é conveniente, dado que a soma total dos *Makespans* no Grid5000 é menor que a soma dos *Makespans* obtidos sobre a arquitetura GridPP. Na aplicação CyberShake, mudar da arquitetura DAS-3, seja para a arquitetura Grid5000 ou GridPP, não é conveniente, como são mostrados os valores na tabela. A aplicação CyberShake usando o algoritmo HEFT obtém um melhor *Makespan* se é executado sobre a arquitetura DAS-3. Para as aplicações Epigenomics e Ligo, como mostram os valores da segunda coluna, ir da arquitetura DAS-3 para a arquitetura Grid5000, é bem conveniente, no entanto, ir da arquitetura Grid5000 para GridPP não.

A Tabela 9.4, mostra os valores de adaptabilidade do algoritmo CPOP. Os valores na tabela possuem uma relação próxima aos valores de adaptabilidade do algoritmo HEFT (mostrado na Tabela 9.3). Podemos observar que para as aplicações Montage, Epigenomics e Ligo, a mudança da arquitetura DAS-3 para a arquitetura Grid5000, é conveniente, dado que os valores mostram que

	(Das3, G5k)	(G5k, Gpp)
Montage	1,65	0,76
CyberShake	0,59	0,93
Epigenomics	1,68	0,74
Ligo	1,82	0,70

Tabela 9.3: *Adaptabilidade do algoritmo HEFT.*

a execução na arquitetura Grid5000 retorna melhores valores de *Makespans*, por outro lado, para essas três aplicações, a mudança da arquitetura Grid5000 para a GridPP, não é conveniente, dado que caso seja mudada a arquitetura os valores de *Makespans* serão maiores. No caso da aplicação CyberShake, como podemos ver, os valores mostram que essa aplicação é bem melhor executada na arquitetura DAS-3, a mudança para as arquiteturas Grid5000 ou GridPP implica um aumento no *Makespan*.

	(Das3, G5k)	(G5k, Gpp)
Montage	1,67	0,75
CyberShake	0,59	0,92
Epigenomics	1,66	0,76
Ligo	1,85	0,74

Tabela 9.4: *Adaptabilidade do algoritmo CPOP.*

Finalmente, na Tabela 9.5, apresenta os valores de adaptabilidade do algoritmo PCH. Para as aplicações Montage, Epigenomics e Ligo, similar aos outros algoritmos, mudar da arquitetura DAS-3 para a arquitetura Grid5000, gera um beneficio, entretanto, mudar da arquitetura Grid5000 para GridPP, gera não gera beneficio. A diferença está na aplicação CyberShake, que é bem executada sobre a arquitetura DAS-3, mas entre as arquiteturas Grid5000 e GridPP, mudar da Grid5000 para a GridPP é conveniente, pois gera um *Makespan* menor.

	(Das3, G5k)	(G5k, Gpp)
Montage	1,75	0,75
CyberShake	0,32	1,59
Epigenomics	1,67	0,77
Ligo	1,79	0,72

Tabela 9.5: *Adaptabilidade do algoritmo PCH.*

Nas tabelas apresentadas de adaptabilidade, podemos ver que os algoritmos HEFT e CPOP, possuem os mesmos valores, isso implica, que para um cenário onde a arquitetura seja cambiante o uso de um algoritmo ou do outro não traz um grande beneficio. De forma similar o algoritmo PCH, possui valores similares, possuindo uma diferença na aplicação CyberShake.

Distribuição da Carga do Trabalho

A avaliação da **Distribuição da Carga do Trabalho**, feita para todas as grades e cada aplicação, é o critério onde tentamos ver qual algoritmo é o melhor distribuidor do conjunto de tarefas sobre os recursos, assim, teremos uma noção de qual algoritmo faz um melhor uso dos recursos.

Para poder visualizar como foi feito o escalonamento das tarefas sobre os recursos usamos a ferramenta de visualização Jedule [HHS10], a qual toma dados gerados na simulação feita pelo simulador SimGrid (usado neste trabalho) e gera como saída uma figura que apresenta a distribuição das tarefas sobre cada processador pertencente à grade.

Nas Figuras 9.12, 9.13 e 9.14 são mostradas as distribuições da carga do trabalho dos três algoritmos de escalonamento, o HEFT, o CPOP e o PCH, respectivamente, na aplicação Montage com 50 tarefas sobre a arquitetura DAS-3. Podemos ver que tanto o algoritmo HEFT quanto o algoritmo CPOP distribuíram as tarefas em recursos próximos e de uma forma uniforme. Entanto, o algoritmo PCH fez uso de alguns recursos distantes aos recursos onde a maioria das tarefas foram escalonadas.

Nas Figuras 9.15, 9.16 e 9.17 são mostrados os resultados da distribuição de tarefas para uma aplicação Montage com 500 tarefas, sobre a arquitetura DAS-3. Podemos ver que a distribuição de tarefas usando o algoritmo HEFT, novamente tem um parecido à distribuição feita pelo algoritmo CPOP. A distribuição feita pelos algoritmos é uniforme e bem estruturada, podemos ver que a estrutura da distribuição é bem parecida à estrutura da aplicação. No caso do algoritmo PCH, a distribuição que o algoritmo faz, é bem mais espalhada usando recursos distantes uns de outros.

Nas Figuras 9.18, 9.19 e 9.20 são mostrados os resultados da distribuição de tarefas para uma aplicação Montage com 1000 tarefas, sobre a arquitetura DAS-3. Confirmamos que os algoritmos HEFT e CPOP distribuem as tarefas de forma parecida, com uma diferença nas tarefas do caminho crítico, que no caso do algoritmo CPOP são escalonamento no melhor processador da grade.

Podemos observar que a atribuição das tarefas tem um parecido à estrutura da aplicação, para cada algoritmo. Além disso, vemos que a distribuição feita pelo algoritmo HEFT tem um parecido à distribuição feita pelo algoritmo CPOP, enquanto que a distribuição feita pelo algoritmo PCH difere. O algoritmo que usa maior quantidade de recursos e com maior Makespan é geralmente o PCH, esse fato é pelas estruturas das aplicações e pelo algoritmo que forma agrupamentos das tarefas, ao existir as tarefas de sincronização críticas os grupos de tarefas formadas pelo algoritmo são grupos com duas tarefas na maioria dos casos. No caso do algoritmo CPOP, que escalona o caminho crítico no processador que ofereça melhor tempo no término às tarefas que pertencem ao caminho crítico, esse critério cria uma dependência das outras tarefas, esse fato é mais bem apresentado na distribuição da aplicação Ligo.

Todas os resultados da distribuição de tarefas, de todas as simulações feitas neste trabalho estão disponíveis em http://www.ime.usp.br/~alvaroma/research.html#workload.

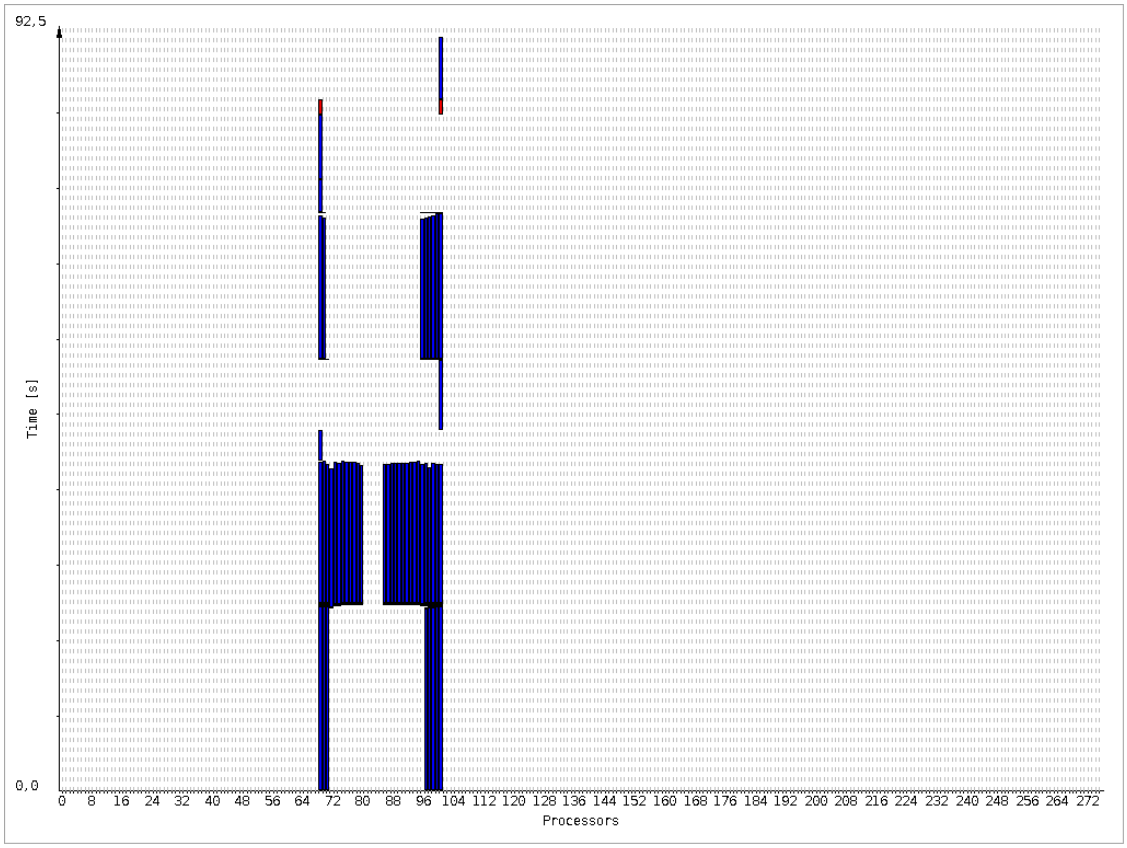


Figura 9.12: *HEFT: Montage de 50 Tarefas sobre DAS-3.*

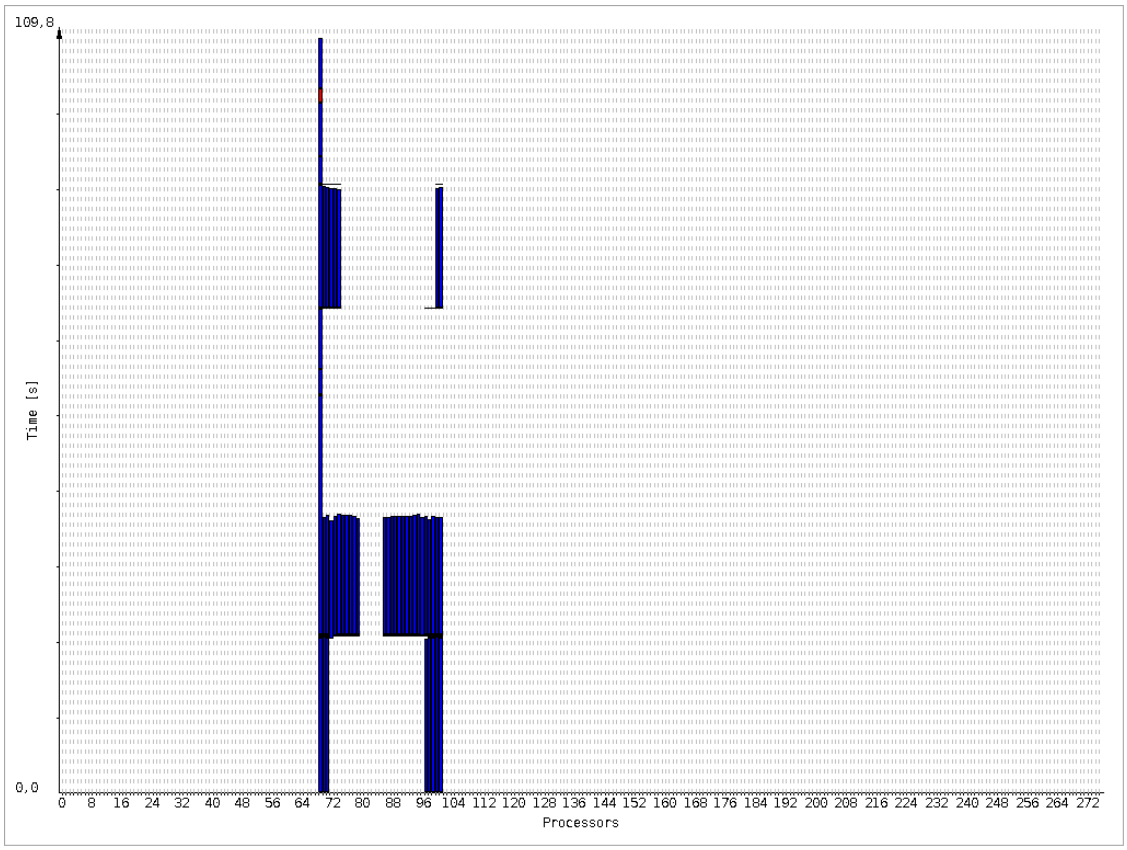


Figura 9.13: *CPOP - Montage de 50 Tarefas sobre DAS-3.*

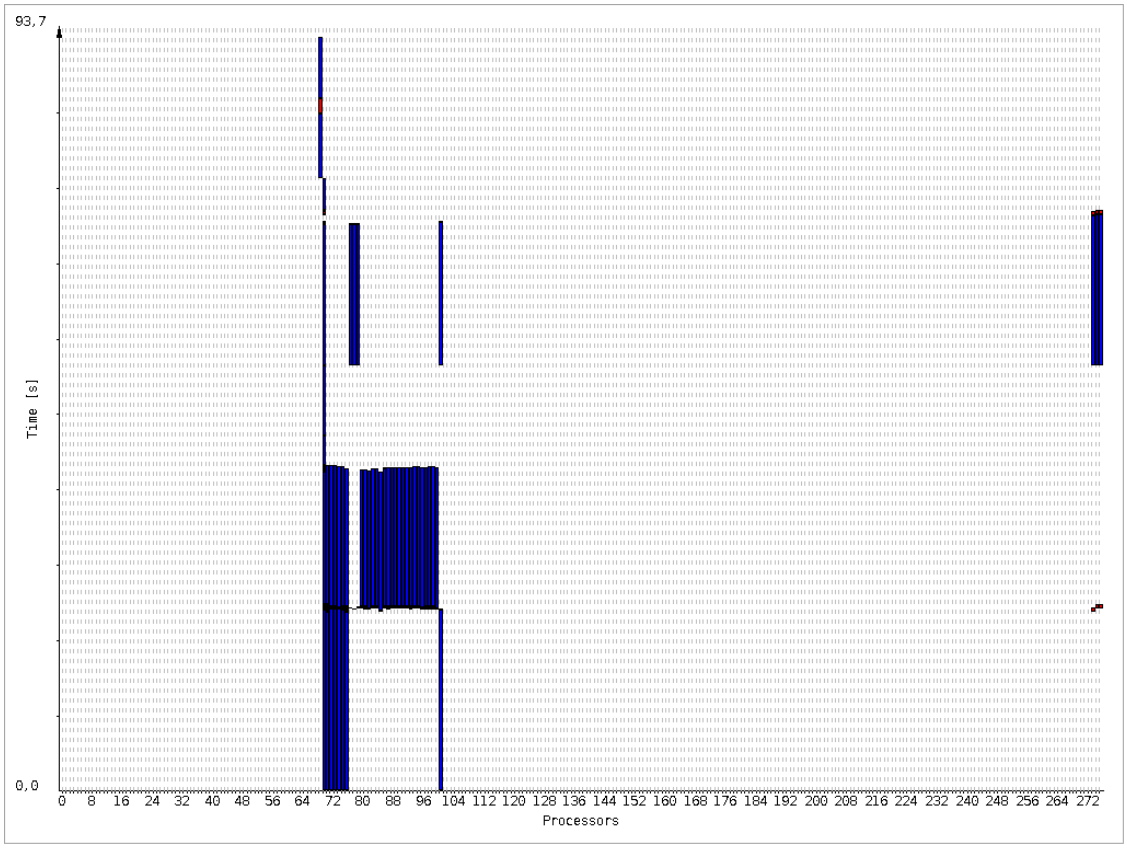


Figura 9.14: *PCH - Montage de 50 Tarefas sobre DAS-3.*

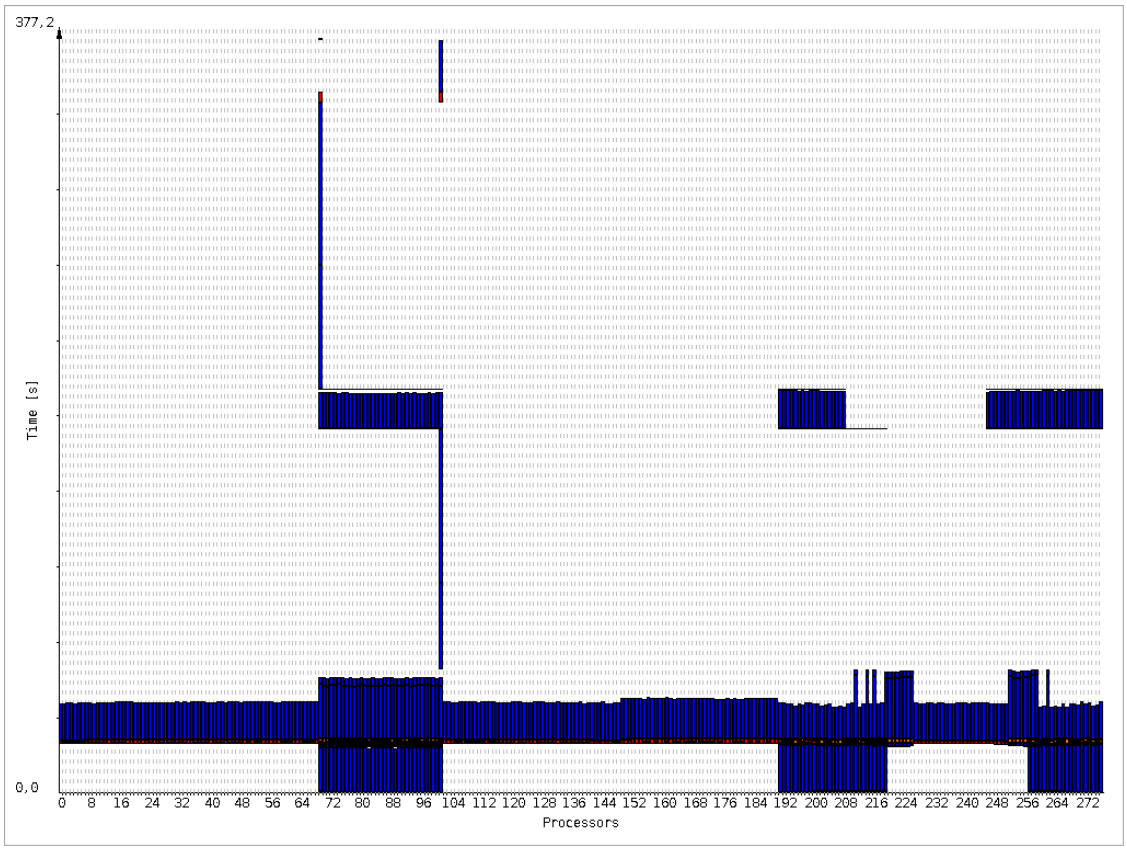


Figura 9.15: *HEFT: Montage de 500 Tarefas sobre DAS-3.*

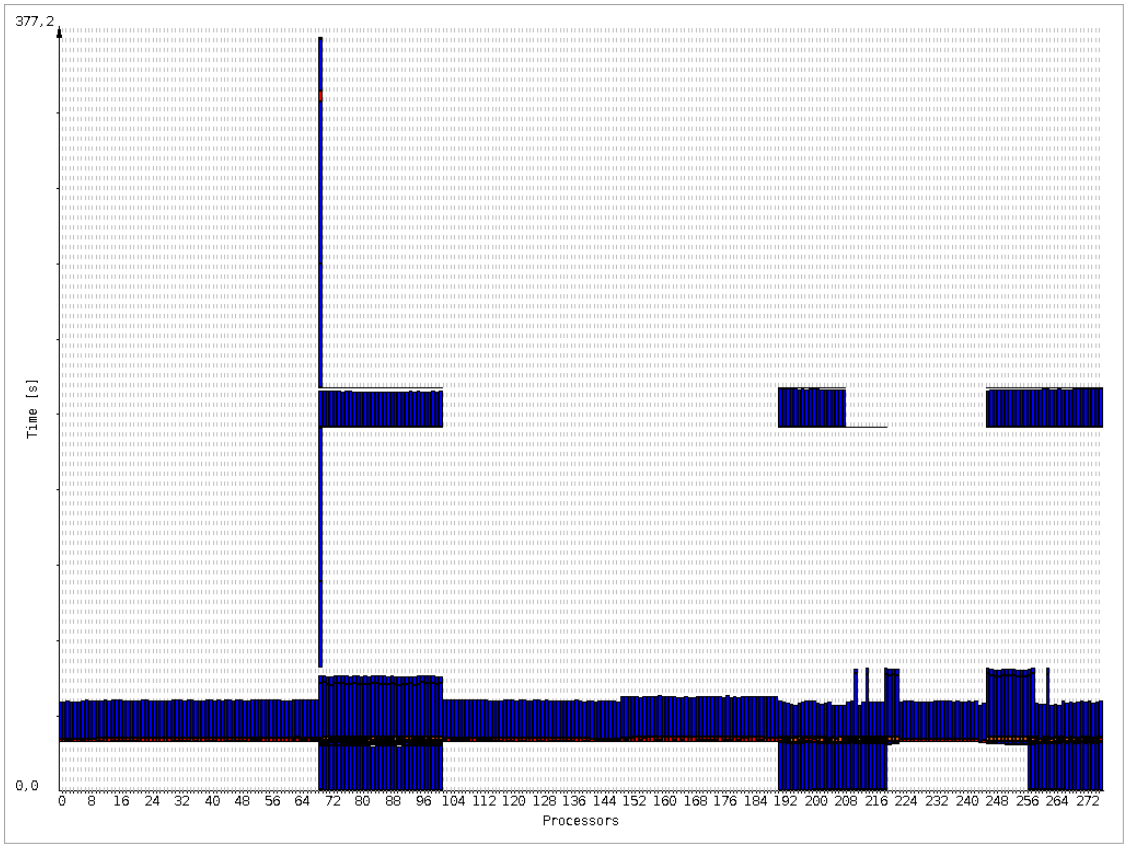


Figura 9.16: *CPOP: Montage de 500 Tarefas sobre DAS-3.*

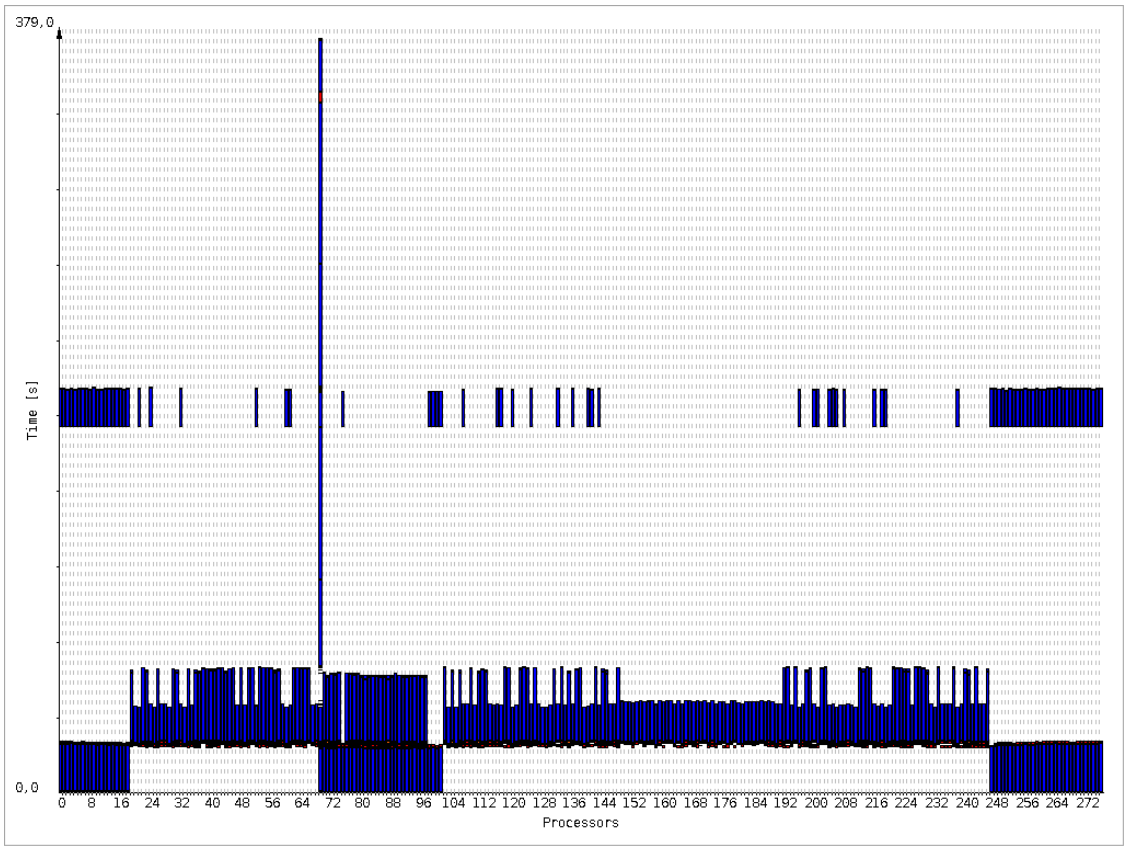


Figura 9.17: *PCH: Montage de 500 Tarefas sobre DAS-3.*

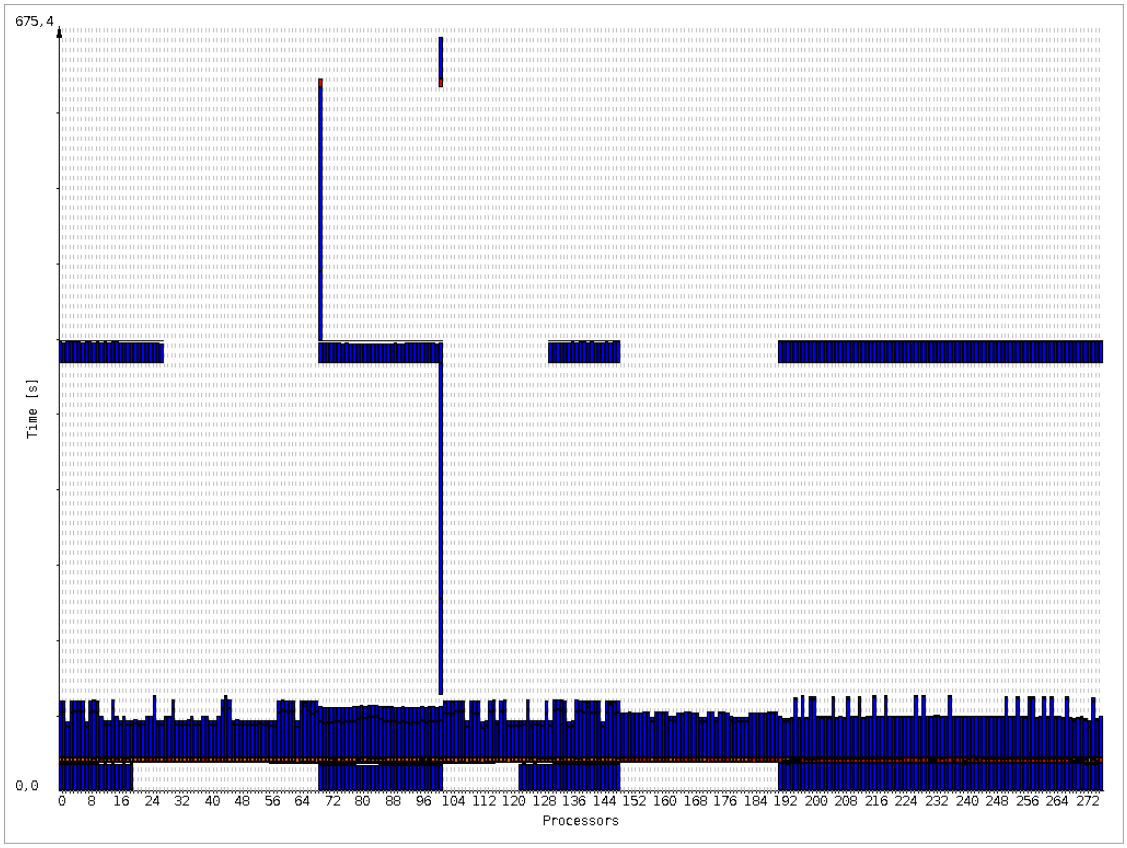


Figura 9.18: *HEFT: Montage de 1000 Tarefas sobre DAS-3.*

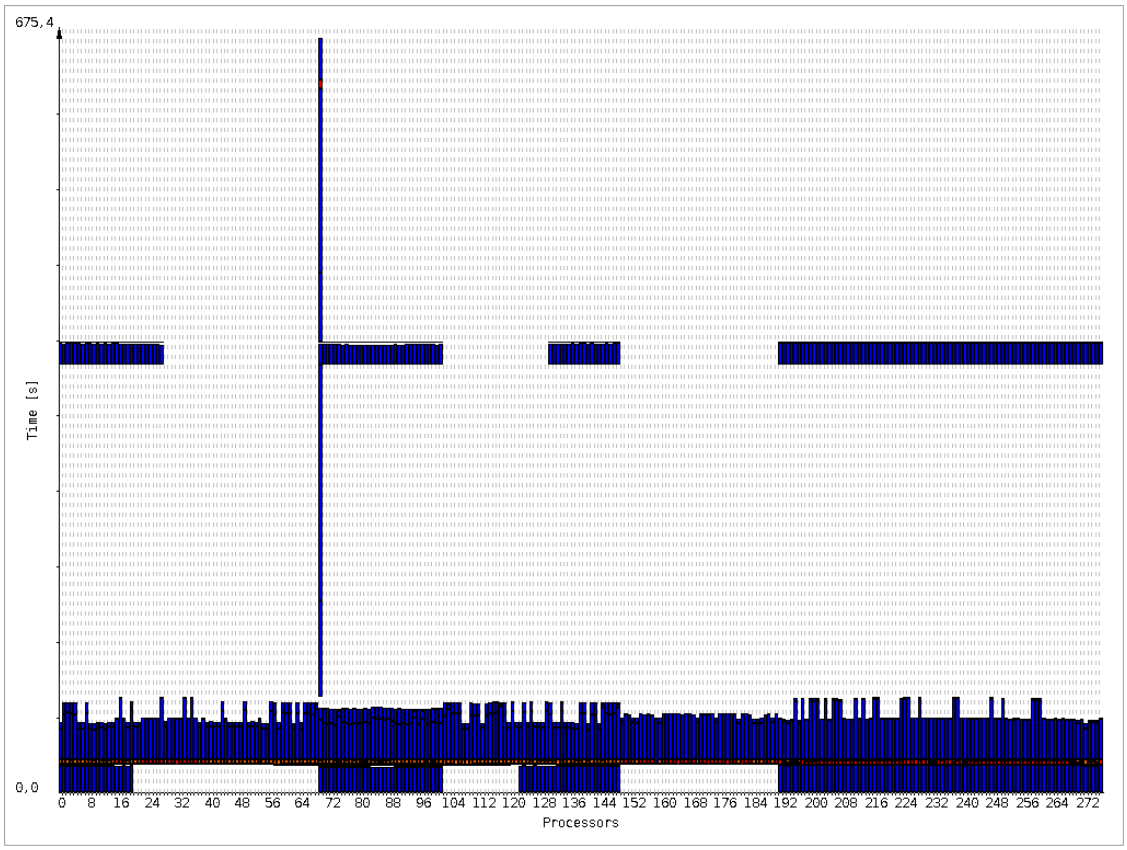


Figura 9.19: *CPOP: Montage de 1000 Tarefas sobre DAS-3.*

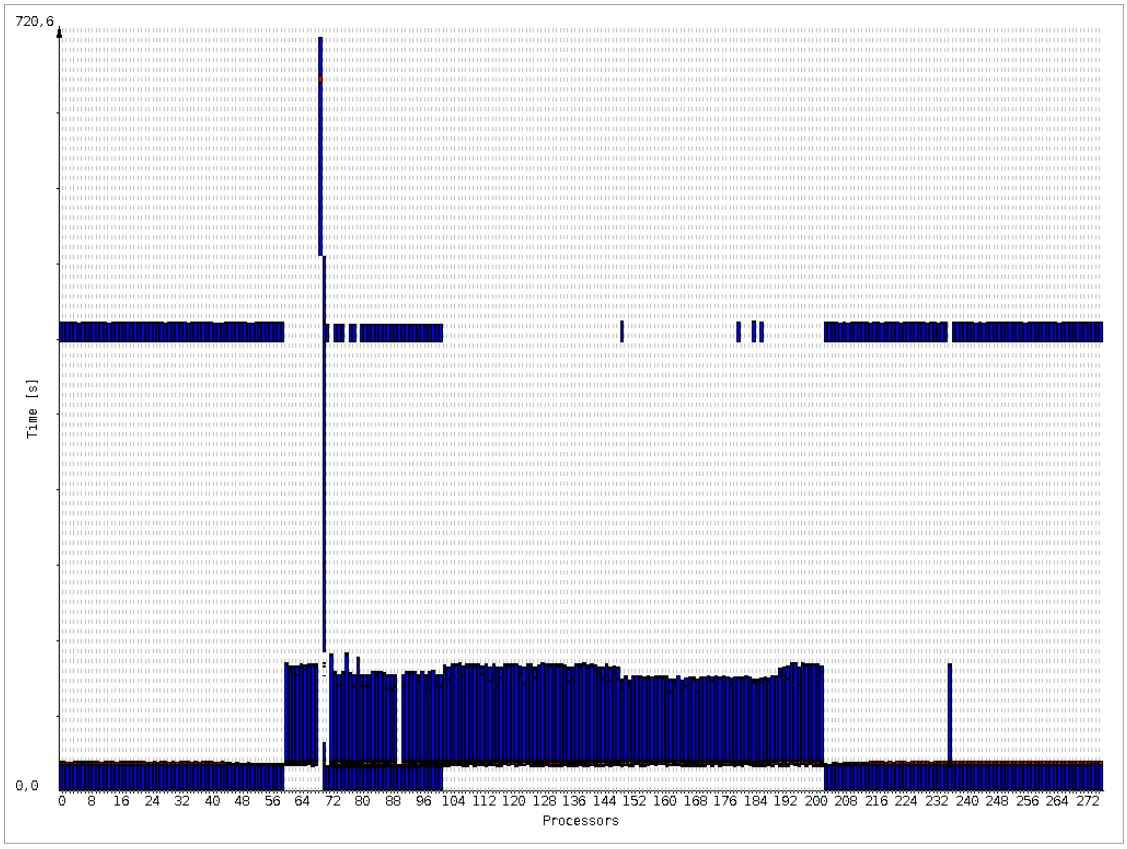


Figura 9.20: *CPOP: Montage de 1000 Tarefas sobre DAS-3.*

Capítulo 10

Conclusões

10.1 Considerações Finais

A computação em grade é uma alternativa para a execução de aplicações paralelas ou distribuídas que demandam alto poder computacional. Essas aplicações são compostas por diversas tarefas que, a depender do tipo de aplicação, podem se comunicar durante a fase de execução. No escopo da computação em grade, o escalonamento é um grande desafio, para atingir um bom desempenho no tempo de execução de aplicações.

No entanto, na literatura foram propostos diferentes algoritmos de escalonamento. A escolha de um algoritmo de escalonamento que tenha as características necessárias para obter um desempenho bom em um determinado cenário é indispensável. Neste trabalho foi proposta uma metodologia para a comparação dos algoritmos de escalonamento que segue quatro critérios.

(a) **Desempenho**: usando a métrica da soma total dos *Makepans* sobre diferentes grades o algoritmo que oferece o melhor desempenho por grade e por aplicação, uma noção geral;

(b) **Escalabilidade**: usando a métrica da média dos *Makespans* do número de tarefas e nós da grade; essa avaliação é feita para aplicações regulares sobre todas as grades. O intuito é encontrar como é o desempenho do algoritmo de escalonamento sobre diferente número de tarefas e processadores;

(c) **Adaptabilidade**: usando a métrica de taxa entre o total dos *Makespans* por grade e por aplicação. Essa avaliação é feita por aplicação sobre todas as grades com o intuito de identificar qual algoritmo é o mais adaptativo;

(d) **Distribuição da Carga do Trabalho**: usando a métrica do número de tarefas por nós da grade e tempo necessário para a comunicação entre elas. O intuito é entender qual algoritmo é o melhor distribuidor na carga do trabalho, dependendo da heterogeneidade.

Para fazer uma boa recomendação de algoritmos de escalonamento em ambientes de grade computacional é importante entender o tipo de aplicação. As aplicações podem ser classificadas em dois tipos: (i) Aplicação Regular e (ii) Aplicação Irregular. No caso das aplicações irregulares não é possível medir escalabilidade, pois neste tipo de aplicações dado um conjunto de entrada, não é possível saber qual será o número de tarefas que terá a aplicação, um exemplo clássico neste tipo é o problema de satisfabilidade (SAT). Também é necessário ter um conhecimento prévio das

características da grade: poder computacional, largura de banda, número de processadores, entre outros.

Assim, foram avaliados os algoritmos para grades computacionais: *Path Clustering Heuristic* (PCH)[BM06], *Critical Path on a Processor* (CPOP) [THW02] e *Heterogeneuos Earliest Finish Time* (HEFT) [THW02]. Na avaliação dos algoritmos o algoritmo HEFT possui um bom desempenho na maioria dos casos, apresentando uma estabilidade, independente do tipo de aplicação ou da arquitetura da grade; enquanto que os outros algoritmos, CPOP e PCH, apresentaram um desempenho bom sobre determinadas circunstâncias.

No caso do algoritmo CPOP possui uma dependência sobre a estrutura da aplicação e da arquitetura, dado que escalona as tarefas do caminho crítico, sobre o processador que ofereça melhor tempo no término, esse escalonamento cria uma dependência das outras tarefas que não pertencem ao caminho crítico.

O algoritmo PCH agrupa as tarefas e escalona cada grupo no processador que ofereça o melhor tempo de término. Esse critério perde sentido em tipos de aplicações com tarefas de sincronização crítica, isto porque, nesse tipo de aplicações, os grupos criados possuem somente uma ou duas tarefas escalonados muitas vezes sobre diferentes processadores, característica que contradiz o alvo do algoritmo.

10.2 Sugestões para Pesquisas Futuras

A comparação de algoritmos de escalonamento é um problema complexo, assim, depois deste estudo muitas questões ainda estão em aberto. Apesar da abordagem proposta neste trabalho, o estudo pode ser melhorado. A seguir, descrevemos algumas questões importantes, as quais precisam ser pesquisadas como continuação do trabalho.

- Os experimentos feitos no trabalho foram feitos utilizando um grupo limitado de aplicações (quatro). Sugerimos ter um maior conjunto de aplicações, tanto em tamanho quanto em forma da estrutura. Como uma alternativa existe o uso de um gerador randômico de grafos de aplicações.

- Com respeito às arquiteturas, a variação nos tipos de arquitetura, desde largura de banda, capacidade dos processadores, até o nível de considerar arquiteturas com vários núcleos, este tipo de experimentos não foi abordado pelo fato do simulador ainda não suportar este tipo de arquiteturas.

- Das três técnicas de escalonamento avaliadas, nenhuma faz um estudo do comportamento do melhor processador, somente eles são baseados no processador que oferece o melhor tempo de término da tarefa ou do grupo de tarefas, sem considerar que esse processador pode ser o processador que tenha uma comunicação muito ruim em relação aos demais. Assim, recomendá-se desenvolver um algoritmo que considere esses aspectos, pois em tipos de aplicações com grande número de tarefas de sincronização crítica aumenta o *Makespan*.

Apêndice A

Exemplo de Especificação das Arquiteturas

Neste exemplo é apresentada a arquitetura do projeto DAS-3. As especificações das outras arquiteturas estão disponíveis em http://www.ime.usp.br/~alvaroma/research.html.

```
1 <?xml version='1.0'?>
2 <!--
3       Url: http://www.cs.vu.nl/das3/
4       Adaptation from: Platform description for DAS-3 grid.
5       Update to work on SimGrid 3.5
6        By: Alvaro Henry Mamani Aliaga
7
8       Total nodes: 272 in 5 sites
9 -->
10 <!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
11 <platform version="3">
12 <AS id="AS_das3" routing="Floyd">
13
14   <cluster id="vu" prefix="vrije-universiteit-" suffix=".das3.nl"
15    radical="0-85" power="2.40E9" bw="1.25E9" lat="1.0E-4"
16    bb_bw="1.25E9" bb_lat="1.0E-4"></cluster>
17
18   <cluster id="lu" prefix="leiden-university-" suffix=".das3.nl"
19    radical="0-32" power="2.60E9" bw="1.25E9" lat="1.0E-4"
20    bb_bw="1.25E9" bb_lat="1.0E-4"></cluster>
21
22   <cluster id="uva" prefix="university-of-amsterdam-" suffix=".das3.nl"
23    radical="0-41" power="2.20E9" bw="1.25E9" lat="1.0E-4"
24    bb_bw="1.25E9" bb_lat="1.0E-4"></cluster>
25
26   <cluster id="tud" prefix="delpft-university-of-technology-" suffix=".das3.nl"
27    radical="0-68" power="2.40E9" bw="1.25E9" lat="1.0E-4"
28    bb_bw="1.25E9" bb_lat="1.0E-4"></cluster>
29
30   <cluster id="uva-mn" prefix="multimedia-consortium-" suffix=".das3.nl"
31    radical="0-46" power="2.40E9" bw="1.25E9" lat="1.0E-4"
32    bb_bw="1.25E9" bb_lat="1.0E-4"></cluster>
33
34   <AS id="network" routing="Floyd" >
```

```
35
36      <router id="VU-switch_sw"/>
37      <router id="LU-switch_sw"/>
38      <router id="UvA-switch_sw"/>
39      <router id="UvA-MN-switch_sw"/>
40      <router id="TUD-switch_sw"/>
41
42      <router id="VU-switch_gw"/>
43      <router id="LU-switch_gw"/>
44      <router id="UvA-switch_gw"/>
45      <router id="UvA-MN-switch_gw"/>
46      <router id="TUD-switch_gw"/>
47
48      <router id="amsterdam1-switch"/>
49      <router id="amsterdam2-switch"/>
50
51      <link id="LU-amsterdam2-bb" bandwidth="1.00E11" latency="1.00E-4"/>
52        <link id="VU-amsterdam2-bb"  bandwidth="1.00E11" latency="1.00E-4"/>
53        <link id="TUD-amsterdam1-bb" bandwidth="1.00E11" latency="1.00E-4"/>
54        <link id="amsterdam1-amsterdam2-bb" bandwidth="1.00E11"
55        latency="1.00E-4"/>
56        <link id="UvA-MN-amsterdam1-bb" bandwidth="1.00E11" latency="1.00E-4"/>
57        <link id="UvA-amsterdam1-bb" bandwidth="1.00E11" latency="1.00E-4"/>
58
59      <link id="VU-switch_sw_gw" bandwidth="1.00E11" latency="1.00E-4"/>
60      <link id="LU-switch_sw_gw" bandwidth="1.00E11" latency="1.00E-4"/>
61      <link id="UvA-switch_sw_gw" bandwidth="1.00E11" latency="1.00E-4"/>
62      <link id="UvA-MN-switch_sw_gw" bandwidth="1.00E11" latency="1.00E-4"/>
63      <link id="TUD-switch_sw_gw" bandwidth="1.00E11" latency="1.00E-4"/>
64
65      <route src="VU-switch_sw"  dst="amsterdam2-switch">
66        <link_ctn id="VU-amsterdam2-bb"/>
67      </route>
68      <route src="amsterdam2-switch"  dst="VU-switch_sw">
69        <link_ctn id="VU-amsterdam2-bb"/>
70      </route>
71      <route src="LU-switch_sw"  dst="amsterdam2-switch">
72        <link_ctn id="LU-amsterdam2-bb"/>
73      </route>
74      <route src="amsterdam2-switch"  dst="LU-switch_sw">
75        <link_ctn id="LU-amsterdam2-bb"/>
76      </route>
77
78      <route src="amsterdam2-switch"  dst="amsterdam1-switch">
79        <link_ctn id="amsterdam1-amsterdam2-bb"/>
80      </route>
81      <route src="amsterdam1-switch"  dst="amsterdam2-switch">
82        <link_ctn id="amsterdam1-amsterdam2-bb"/>
83      </route>
84
85      <route src="UvA-switch_sw"  dst="amsterdam1-switch">
86        <link_ctn id="UvA-amsterdam1-bb"/>
87      </route>
```

```
88      <route src="amsterdam1-switch"  dst="UvA-switch_sw">
89        <link_ctn id="UvA-amsterdam1-bb"/>
90      </route>
91      <route src="UvA-MN-switch_sw"  dst="amsterdam1-switch">
92        <link_ctn id="UvA-MN-amsterdam1-bb"/>
93      </route>
94      <route src="amsterdam1-switch"  dst="UvA-MN-switch_sw">
95        <link_ctn id="UvA-MN-amsterdam1-bb"/>
96      </route>
97      <route src="TUD-switch_sw"  dst="amsterdam1-switch">
98        <link_ctn id="TUD-amsterdam1-bb"/>
99      </route>
100     <route src="amsterdam1-switch"  dst="TUD-switch_sw">
101       <link_ctn id="TUD-amsterdam1-bb"/>
102     </route>
103
104     <!--gw sw-->
105     <route src="VU-switch_gw" dst="VU-switch_sw">
106       <link_ctn id="VU-switch_sw_gw"/>
107     </route>
108     <route src="VU-switch_sw" dst="VU-switch_gw">
109       <link_ctn id="VU-switch_sw_gw"/>
110     </route>
111
112     <route src="LU-switch_gw" dst="LU-switch_sw">
113       <link_ctn id="LU-switch_sw_gw"/>
114     </route>
115     <route src="LU-switch_sw" dst="LU-switch_gw">
116       <link_ctn id="LU-switch_sw_gw"/>
117     </route>
118
119     <route src="UvA-switch_gw" dst="UvA-switch_sw">
120       <link_ctn id="UvA-switch_sw_gw"/>
121     </route>
122     <route src="UvA-switch_sw" dst="UvA-switch_gw">
123       <link_ctn id="UvA-switch_sw_gw"/>
124     </route>
125
126     <route src="UvA-MN-switch_gw" dst="UvA-MN-switch_sw">
127       <link_ctn id="UvA-MN-switch_sw_gw"/>
128     </route>
129     <route src="UvA-MN-switch_sw" dst="UvA-MN-switch_gw">
130       <link_ctn id="UvA-MN-switch_sw_gw"/>
131     </route>
132
133     <route src="TUD-switch_gw" dst="TUD-switch_sw">
134       <link_ctn id="TUD-switch_sw_gw"/>
135     </route>
136     <route src="TUD-switch_sw" dst="TUD-switch_gw">
137       <link_ctn id="TUD-switch_sw_gw"/>
138     </route>
139
140   </AS>
```

```
141
142  <link id="vu_gw" bandwidth="1.00E11" latency="1.00E-4"/>
143  <link id="lu_gw" bandwidth="1.00E11" latency="1.00E-4"/>
144  <link id="uva_gw" bandwidth="1.00E11" latency="1.00E-4"/>
145  <link id="uva-mn_gw" bandwidth="1.00E11" latency="1.00E-4"/>
146  <link id="tud_gw" bandwidth="1.00E11" latency="1.00E-4"/>
147
148  <ASroute src="vu" dst="network"
149      gw_src="vrije-universiteit-vu_router.das3.nl"
150      gw_dst="VU-switch_gw" >
151    <link_ctn id="vu_gw"/>
152  </ASroute>
153  <ASroute src="network" dst="vu"
154      gw_src="VU-switch_gw"
155      gw_dst="vrije-universiteit-vu_router.das3.nl" >
156    <link_ctn id="vu_gw"/>
157  </ASroute>
158
159  <ASroute src="lu" dst="network"
160      gw_src="leiden-university-lu_router.das3.nl"
161      gw_dst="LU-switch_gw" >
162    <link_ctn id="lu_gw"/>
163  </ASroute>
164  <ASroute src="network" dst="lu"
165      gw_src="LU-switch_gw"
166      gw_dst="leiden-university-lu_router.das3.nl" >
167    <link_ctn id="lu_gw"/>
168  </ASroute>
169
170  <ASroute src="uva" dst="network"
171      gw_src="university-of-amsterdam-uva_router.das3.nl"
172      gw_dst="UvA-switch_gw" >
173    <link_ctn id="uva_gw"/>
174  </ASroute>
175  <ASroute src="network" dst="uva"
176      gw_src="UvA-switch_gw"
177      gw_dst="university-of-amsterdam-uva_router.das3.nl" >
178    <link_ctn id="uva_gw"/>
179  </ASroute>
180
181  <ASroute src="tud" dst="network"
182      gw_src="delpft-university-of-technology-tud_router.das3.nl"
183      gw_dst="TUD-switch_gw" >
184    <link_ctn id="tud_gw"/>
185  </ASroute>
186  <ASroute src="network" dst="tud"
187      gw_src="TUD-switch_gw"
188      gw_dst="delpft-university-of-technology-tud_router.das3.nl">
189    <link_ctn id="tud_gw"/>
190  </ASroute>
191
192  <ASroute src="uva-mn" dst="network"
193      gw_src="multimedia-consortium-uva-mn_router.das3.nl"
```

```
194         gw_dst="UvA-MN-switch_gw" >
195       <link_ctn id="uva-mn_gw"/>
196     </ASroute>
197     <ASroute src="network" dst="uva-mn"
198         gw_src="UvA-MN-switch_gw"
199         gw_dst="multimedia-consortium-uva-mn_router.das3.nl">
200       <link_ctn id="uva-mn_gw"/>
201     </ASroute>
202   </AS>
203 </platform>
```

Referências Bibliográficas

[AKW05] David P. Anderson, Eric Korpela e Rom Walton. High-performance task distribution for volunteer computing. Em *e-Science and Grid Computing, 2005. First International Conference on*, 2005. Citado na(s) página(s) 19, 20

[And04] David P. Anderson. Boinc: A system for public-resource computing and storage. Em *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, páginas 4–10, Washington, DC, USA, 2004. IEEE Computer Society. Citado na(s) página(s) 19

[And07] David P. Anderson. Local scheduling for volunteer computing. Em *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, páginas 1–8, 2007. Citado na(s) página(s) 20

[Bat10] Daniel Macêdo Batista. *Escalonamento de Tarefas Dependentes para Grades Robustos às Incertezas das Informações de Entrada.* Tese de Doutorado, Unicamp - Universidade Estadual de Campinas. Instituto de Computação, Campinas, São Paulo, Brasil, Fevereiro 2010. Citado na(s) página(s) 7

[BBA+04] G. Bruce Berriman, Ewa Deelman B., John Good A., Joseph Jacob C., Daniel S. Katz, Carl Kesselman C., Anastasia Laity, Thomas A. Prince, Gurmeet Singh e Mei hu Su. Montage: A grid enabled engine for delivering custom science-grade mosaics on demand. Em *SPIE*, 2004. Citado na(s) página(s) 35

[BBH+00] Henri Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, Jason Maassen, Rob van Nieuwpoort, John Romein, Luc Renambot, Tim Rühl, Ronald Veldema, Kees Verstoep, Aline Baggio, Gerco Ballintijn, Ihor Kuz, Guillaume Pierre, Maarten van Steen, Andy Tanenbaum, Gerben Doornbos, Desmond Germans, Hans Spoelder, Evert-Jan Baerends, Stan van Gisbergen, Hamideh Afsermanesh, Dick van Albada, Adam Belloum, David Dubbeldam, Zeger Hendrikse, Bob Hertzberger, Alfons Hoekstra, Kamil Iskra, Drona Kandhai, Dennis Koelma, Frank van der Linden, Benno Overeinder, Peter Sloot, Piero Spinnato, Dick Epema, Arjan van Gemund, Pieter Jonker, Andrei Radulescu, Cees van Reeuwijk, Henk Sips, Peter Knijnenburg, Michael Lew, Floris Sluiter, Lex Wolters, Hans Blom, Cees de Laat e Aad van der Steen. The distributed asci supercomputer project. *SIGOPS Oper. Syst. Rev.*, 34:76–96, 2000. Citado na(s) página(s) 27

[BCC+03] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger e Floriano Zini. Optorsim - a grid simulator for studying dynamic data replication stra-

tegies. *International Journal of High Performance Computing Applications*, páginas 403–416, 2003. Citado na(s) página(s) 44

[BCC+06] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lantéri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, Talbi El-Ghazali e Iréa Touche. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006. Citado na(s) página(s) 28

[BCC+09] Dave Britton, Tony Cass, Pete Clarke, Jeremy Coles, Dave Colling, Tony Doyle, Neil Geddes, John Gordon, Roger Jones e David Kelsey. Gridpp- the uk grid for particle physics, 2009. Citado na(s) página(s) 31

[BCD+08] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su e Karan Vahi. Characterization of scientific workflows. *The 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS08), in conjunction with Supercomputing (SC08) Conference.*, Novembro 2008. Citado na(s) página(s) 35, 47

[BFH03] Fran Berman, Geoffrey Fox e Tony Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, New York, NY, USA, primeira edição, 2003. Citado na(s) página(s) 1

[BHK+00] Brett Bode, David M. Halstead, Ricky Kendall, Zhou Lei e David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. Em *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, páginas 27–27, 2000. Citado na(s) página(s) 14, 16

[Bit10] Luiz F. Bittencourt. *Algoritmos para Escalonamento de Tarefas Dependentes Representadas por Grafos Acíclicos Direcionados em Grades Computacionais.* Tese de Doutorado, Unicamp - Universidade Estadual de Campinas. Instituto de Computação, Campinas, São Paulo, Brasil, Fevereiro 2010. Citado na(s) página(s) 58

[BM02] Rajkumar Buyya e Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience (CCPE)*, 14(13):1175–1220, 2002. Citado na(s) página(s) 44

[BM06] Luiz F. Bittencourt e Edmundo R. M. Madeira. A dynamic approach for scheduling dependent tasks on the Xavantes grid middleware. Em *MCG '06: Proceedings of the 4th international workshop on Middleware for grid computing.* ACM, 2006. Citado na(s) página(s) 1, 82

[BMCB06] Luiz F. Bittencourt, Edmundo R. M. Madeira, Fábio R. L. Cicerre e Luiz E. Buzato. Uma heuristica de agrupamento de caminhos para escalonamento de tarefas em grades computacionais. Em *Anais SBRC 2006*, páginas 83–98, Curitiba, Brazil, 2006. Citado na(s) página(s) 54, 57, 58

[CBA+06] Waldredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro B. Costa, Alisson Andrade, Reynaldo Novaes e Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006. Citado na(s) página(s) 6, 22, 23

[CCD+05] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet e Olivier Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. 2005. Citado na(s) página(s) 28

[CDCG+05] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron e O. Richard. A batch scheduler with high level components. Em *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2, páginas 776–783, 2005. Citado na(s) página(s) 9, 10

[CDF+05] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri e Oleg Lodygensky. Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3):417–437, 2005. Citado na(s) página(s) 24

[CHR09] Benoit Claudel, Guillaume Huard e Olivier Richard. Taktuk, adaptive deployment of remote executions. Em *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, páginas 91–100, 2009. Citado na(s) página(s) 9

[CJSZ08] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou e Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. páginas 63–74, 2008. Citado na(s) página(s) 59

[CK88] Thomas L. Casavant e Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988. Citado na(s) página(s) 1, 6

[CLQ08] Henri Casanova, Arnaud Legrand e Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. Em *10th IEEE International Conference on Computer Modeling and Simulation*. IEEE Computer Society Press, March 2008. Citado na(s) página(s) 44

[CLZB00] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov e Francine Berman. Heuristics for scheduling parameter sweep applications in grid environments. Em *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, páginas 349–363, 2000. Citado na(s) página(s) 1, 50

[CMB04] Fábio R. L. Cicerre, Edmundo R. M. Madeira e Luiz E. Buzato. A hierarchical process execution support for grid computing. Em *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, páginas 87–92. ACM, 2004. Citado na(s) página(s) 54

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício A. B. Silva, Carla O. Barros e Cirano Silveira. Running Bag-of-Tasks applications on computational grids: the Mygrid approach. *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 0:407–416, 2003. Citado na(s) página(s) 22, 49

[CWT+04] Clovis Chapman, Paul Wilson, Todd Tannenbaum, Matthew Farrellee, Miron Livny, John Brodholt e Wolfgang Emmerich. Condor services for the global grid: interoperability between Condor and OGSA. Em *Proceedings of 2004 UK e-Science All Hands Meeting*, páginas 870–877, Nottingham, UK, August 2004. Citado na(s) página(s) 12

[DA06] Fangpeng Dong e Selim G. Akl. Scheduling algorithms for grid computing: State of art and open problems. Relatório técnico, School of Computing, Queen's University, Kingston, Ontario, 2006. Citado na(s) página(s) 49

[Dan05] Mario Dantas. *Computação Distribuida De Alto Desempenho. Redes, Clusters e Grids Computacionais*. Axcel Books, Rio de Janeiro, RJ, Brasil, primeira edição, 2005. Citado na(s) página(s) 5, 6

[dSCB03] Daniel Paranhos da Silva, Walfredo Cirne e Francisco Vilar Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. Em *Euro-Par*, páginas 169–180, 2003. Citado na(s) página(s) 1, 23, 49, 50

[FGNC01] Gilles Fedak, Cécile Germain, Vincent Neri e Franck Cappello. Xtremweb: a generic global computing system. Em *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, páginas 582–587, 2001. Citado na(s) página(s) 24

[FK04] Ian Foster e Carl Kesselman. *Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, segunda edição, 2004. Citado na(s) página(s) 1, 5

[FKNT02] Ian Foster, Carl Kesselman, Jeffrey M. Nick e Steven Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002. Citado na(s) página(s) 1

[Fos05] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. Em *NPC 2005: IFIP International Conf. on Network and Parallel Computing*, páginas 2–13, Beijing, China, 2005. Citado na(s) página(s) 6

[FQS08] Marc-Eduard Frincu, Martin Quinson e Frédéric Suter. Handling very large platforms with the new simgrid platform description formalism. Relatório Técnico 0348, Institut National de Recherche en Informatique et en Automatique, INRIA, France, 2008. Citado na(s) página(s) 27, 47

[FTF+02] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny e Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002. Citado na(s) página(s) 11

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger e Germano C. Bezerra. Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency - Practice and Experience*, 16(5):449–459, 2004. Citado na(s) página(s) 6, 20, 21

[GW97] Andrew S. Grimshaw e William A. Wulf. The legion vision of a worldwide virtual computer. *Communications of ACM*, 40(1):39–45, 1997. Citado na(s) página(s) 6

[Hen95] Robert L. Henderson. Job scheduling under the portable batch system. Em *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, páginas 279–294, 1995. Citado na(s) página(s) 13, 15

[HHS10] Sascha Hunold, Ralf Hoffmann e Frédéric Suter. Jedule: A Tool for Visualizing Schedules of Parallel Applications. Em *Proceedings of the 1st International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)*, 2010. Citado na(s) página(s) 75

[LLM88] Michael Litzkow, Miron Livny e Matthew Mutka. Condor - a hunter of idle workstations. Em *Proceedings of the 8th International Conference of Distributed Computing Systems*, páginas 104–111, June 1988. Citado na(s) página(s) 11

[Pin08] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, New York, NY, USA, terceira edição, 2008. Citado na(s) página(s) 2, 49

[RLS98] R. Raman, M. Livny e M. Solomon. Matchmaking: distributed resource management for high throughput computing. *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, páginas 140–146, July 1998. Citado na(s) página(s) 12

[SNCBL04] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro e Aliandro Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. Em *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, páginas 210–232, 2004. Citado na(s) página(s) 1, 23, 51, 59

[Sta06] Garrick Staples. Torque resource manager. Em *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006. Citado na(s) página(s) 14

[THW02] Haluk Topcuouglu, Salim Hariri e Min-you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002. Citado na(s) página(s) 1, 52, 53, 54, 55, 59, 82

[TMN+99] Atsuko Takefusa, Satoshi Matsuoka, Hidemoto Nakada, Kento Aida e Umpei Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, páginas 97–104, 1999. Citado na(s) página(s) 43

[TTL05] Douglas Thain, Todd Tannenbaum e Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. Citado na(s) página(s) 11, 13

[WET03] Gropp William, Lusk Ewing e Sterling Thomas. *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge , Massachusetts London, England, segunda edição, 2003. Citado na(s) página(s) 13

Índice Remissivo

Aglomerado, 5
Aplicações
    para Grades, 35

Boinc, 19
    arquitetura, 19
    escalonamento, 20
Bricks, 43

Clustering, 51
Computação em Grade, 1
    fundamentos, 5
Condor, 11
    arquitetura, 11
    Escalonamento, 12
CPOP, 54
Custo de
    computação, 52
    comunicação, 52
    execução, 52

DAG, 46, 51
DAS, 27
DAS-3, 27
    arquitetura, 28
DAX, 47
DOT, 47
downward, 53
Duplication-Based, 51

EFT, 52
Escalonador, 9
Escalonamento
    de Tarefas, 7
    dinâmico, 7
    estático, 6
    fundamentos, 6
    taxonomia, 6
EST, 52

FIFO, 49

GRAS, 46
Grid5000, 28
    arquitetura, 29
GridPP, 31
    arquitetura, 31
GridSim, 44

HEFT, 54

InteGrade
    escalonamento, 21
Integrade, 20
    arquitetura, 21

List Scheduling, 51

Makespan, 1, 6, 49, 54
Maui, 16
Middleware, 5
MSG, 46

OAR, 9
    arquitetura, 9
    escalonamento, 10
Optorsim, 44
OurGrid, 22
    arquitetura, 22
    escalonamento, 23

PBS, 13
    arquitetura, 14
PCH, 55
Plataformas, 27

Resultados
    experimentais, 64

Saco de Tarefas, 7, 22
SimDag, 46
SimGrid, 45
    arquitetura, 45
    componentes, 45
Simulador, 43
    Bricks, 43
    GridSim, 44
    Optorsim, 44
    SimGrid, 44
SMPI, 46
Storage Affinity, 23

Tarefas
    Dependentes, 7
    Independentes, 7
Torque, 14
    Arquitetura, 15
    Escalonamento, 15

upward, 53

Workqueue, 23
Workqueue with Replication, 23

XtremWeb, 24
    arquitetura, 24
    escalonamento, 24