Universidade Federal Fluminense

Fernanda Gonçalves de Oliveira Passos

Provendo Autonomia às Aplicações da e-Ciência

NITERÓI

2014

Universidade Federal Fluminense

Fernanda Gonçalves de Oliveira Passos

Provendo Autonomia às Aplicações da e-Ciência

Tese de **Doutorado** apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Orientador:
Prof. Eugene Francis Vinod Rebello, Ph.D.

NITERÓI

2014

Fernanda Gonçalves de Oliveira Passos

Provendo Autonomia às Aplicações da e-Ciência

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Aprovada em 5 de Setembro de 2014.

Prof. Eugene Francis Vinod Rebello, Ph.D. / IC-UFF
(Orientador)

Prof.ª Alba Cristina Magalhães Alves de Melo, Ph.D. / CIC-UnB

Prof. Philippe Olivier Alexandre Navaux, Ph.D. / INF-UFRGS

Prof.ª Lúcia Maria de Assumpção Drummond, D.Sc. / IC-UFF

Prof.ª Maria Cristina Silva Boeres, Ph.D. / IC-UFF

Niterói
2014

"Look up at the stars and not down at your feet. Try to make sense of what you see, and wonder about what makes the Universe exist. Be curious."

Stephen Hawking

Aos meus admiráveis avós que tive e tenho o prazer de conviver nesta vida:
Lauzina, Gracinda e Rafael.

Agradecimentos

Nesta etapa de minha vida, desde quando iniciei meu doutorado, muitas pessoas importantes me transmitiram alguma forma de ajuda ou inspiração. Gostaria, então, de expor meus agradecimentos aqui.

Primeiramente, agradeço a minha família. Minha amável mãe, Estela, sempre me ensinou a ser dedicada, a lutar pelos meus objetivos e a insistir nas minhas pretensões, mesmo diante das dificuldades. Os melhores conselhos e motivações de minha vida vieram dela. Meu adorável pai, Aloisio, é meu grande exemplo de honestidade, determinação e generosidade. Sempre educou-me, protegeu-me e ajudou-me em tudo que podia. Minha irmã, Rosana, que, desde quando eu era criança, sempre me fez companhia e me ajudou nas dificuldades do meu dia a dia. Meus avós queridos, que sempre cuidaram de mim e minha irmã com muita dedicação e carinho. Meu amado esposo, Diego, que me deu todo apoio e inspiração para me dedicar a este trabalho.

Gostaria de agradecer, também, a todos os professores e funcionários do Instituto de Computação que passaram por minha vida acadêmica. Todos podem ter certeza que contribuíram, através de conhecimento e apoio, para a realização desta tese. Um especial agradecimento ao meu orientador Vinod Rebello, que incansavelmente colaborou para o desenvolvimento deste trabalho e me proporcionou conhecimentos além da expectativa. Ao funcionário Carlos Eduardo da Cunha, por gentilmente ajudar na utilização do *cluster* Oscar, ferramenta fundamental para as avaliações experimentais de minha tese.

Agradeço, ainda, aos membros dos projetos EasyGrid e EUBrazilOpenBio e aos alunos do SGCLAB que contribuíram com conselhos e no suporte das infraestruturas. Ao Carlos Henrique Nicodemus, por auxiliar na configuração de outro *cluster* utilizado. Ao Rafael Amaral, pela colaboração no projeto EUBrazilOpenBio e na configuração de alguns experimentos. Ao Alexandre Sena e Felipe Ribeiro, por cooperarem para a implementação e avaliação de uma das aplicações estudadas.

Finalmente, agradeço ao apoio e investimento dos órgãos de fomento à pesquisa CAPES e FAPERJ, que promoveram suporte financeiro fundamental para o desenvolvimento deste trabalho de pesquisa.

Resumo

O termo *e*-Ciência – *e-Science* – refere-se à área científica que requer o uso de sistemas computacionais de alto desempenho devido a sua grande demanda de processamento de dados. Nos últimos anos, instituições, governos e empresas vêm investindo fortemente nas *e*-infraestruturas necessárias para atender estas demandas computacionais. Com esta tendência, as características destes ambientes estão mudando constantemente e tornando-se cada vez mais largas em escala. Graça às melhorias nas redes de conectividade, as aplicações de um modo geral podem tirar proveito da agregação de recursos geograficamente distribuídos, se forem capazes de lidar com a heterogeneidade e atrasos de comunicação. Além disso, com a necessidade de manter a eficiência em sistemas maiores, os recursos computacionais são tipicamente compartilhados. Consequentemente, o gerenciamento das aplicações nas *e*-infraestruturas encontra-se cada vez mais complexo. Com o objetivo de solucionar este problema, abordagens tradicionais sugerem o uso de uma gerência centrada nos recursos. Porém, deste modo, elas não consideraram características dinâmicas de execução das aplicações, tornando a sustentabilidade destas abordagens questionável diante do cenário atual de sistemas e número de aplicações cada vez maiores.

Esse problema de gerenciamento nos motiva a contemplar uma abordagem alternativa onde as aplicações não sejam consideradas entidades estáticas mas que sejam capazes de tomar decisões de gestão de acordo com suas necessidades específicas. O objetivo desta tese é estudar a viabilidade do projeto e execução de aplicações autônomas nas *e*-infraestruturas com características supracitadas. A principal diretriz da abordagem indica o uso de uma camada de gerenciamento autônomo integrada à aplicação, que a permita configurar-se, otimizar-se e recuperar-se de falhas. Ainda, aplicações podem ser agrupadas em classes e ter uma versão de gerenciamento comum, aliviando o processo de desenvolvimento. Seguindo uma metodologia proposta, versões autônomas de cinco aplicações científicas foram desenvolvidas e avaliadas. Experimentos foram conduzidos com tamanho de aplicações por volta de 500.000 processos e executando em ambientes dedicados e compartilhados. Com até 70% de melhoria de desempenho, os resultados demonstram claramente a eficácia de uma abordagem autônoma em relação à tradicional. Concluindo, o projeto de aplicações da *e*-ciência autônomas não é apenas viável, mas também pode trazer significantes benefícios no desempenho utilizando-se *e*-infraestruturas recentes e futuras.

Palavras-chave: Aplicações da *e*-Ciência, Computação Autônoma, Aplicações Autônomas, Computação Distribuída de Larga Escala, Computação de Alto Desempenho.

Abstract

The term *e-Science* refers to a class of science applications that requires the use of high performance distributed systems due to its high demand for data processing. In recent years, institutions, governments and companies have invested heavily in *e*-infrastructures to meet their computational demands. With this trend, the characteristics of these environments are constantly changing and becoming increasingly large in scale. Due to improvements in network connectivity, applications in general can benefit from the aggregation of geographically distributed resources, if they are able to deal with system heterogeneity and communication delays. On the other hand, with the need to improve efficiency in larger systems, computing resources are typically shared. Consequently, managing the execution of applications in these *e*-infrastructures becomes more complex. Aiming to solve this problem, traditional approaches suggest the use of a resource-centered management. However, they do not consider the dynamic characteristics of the application during execution, making the sustainability of these approaches questionable given the current scenario of ever larger systems and numbers of applications.

This management problem motivates us to contemplate an alternative approach where applications are not considered static entities but are delegated the responsibility to make management decisions according to their own specific needs. The goal of this thesis is to study the viability of the design and execution of autonomic applications in *e*-infrastructures with the aforementioned characteristics. This work proposes the use of an autonomic management layer integrated into the application, which allows it to self-configure, self-optimize and selfheal. Furthermore, applications can be grouped into classes that have a common style of management, simplifying the development process. Following a proposed methodology, autonomic versions of five scientific applications have been developed and evaluated. Experiments were conducted with applications sizes reaching around 500,000 processes and executing in dedicated and shared environments. With up to a 70% improvement in performance, the results clearly highlight the effectiveness of an autonomic approach in relation to the traditional one. In conclusion, the design of autonomic *e*-science applications is not only feasible but could provide significant performance benefits when executed in the current and expected future *e*-infrastructures.

Keywords: e-Science Applications, Autonomic Computing, Autonomic Applications, Large-Scale Distributed Computing, High Performance Computing.

Lista de Abreviaturas e Siglas

AMD	*Advanced Micro Devices*
AMS	*Application Management System*
ANTS	*Autonomous Nano Technology Swarm*
API	*Application Programming Interface*
BOINC	*Berkeley Open Infrastucture for Network Computing*
CERN	*Conseil Européen pour la Recherche Nucléaire*
COMPS	*COMP Superscalar*
COP	*Configurable Object Program*
CUDA	*Compute Unified Device Architecture*
DAGMan	*Directed Acyclic Graph Manager*
DARPA	*Defense Advanced Research Projects Agency*
EGEE	*Enabling Grids for E-science*
EGI	*European Grid Infrastructure*
ENM	*Ecological Niche Modelling*
FLOPS	*FLoating-point Operations Per Second*
GAD	Grafo Acíclico e Direcionado
GBIF	*Global Biodiversity Information Facility*
GCM	*Grid Component Model*
GIS	*Geographic Information System*
GPGPU	*General-Purpose Computing on Graphics Processing Units*
GrADS	*Grid Analysis and Display System*
HEFT	*Heterogeneous Earliest Finish Time*
IBM	*International Business Machines*
IP	*Internet Protocol*
JavaGAT	*Java Grid Application Toolkit*
LHC	*Large Hadron Colider*
MIC	*Many Integrated Core*
MPI	*Message Passing Interface*
NASA	*National Aeronautics and Space Administration*

OM	Open Modeller
OpenCL	*Open Computing Language*
OpenGL	*Open Graphics Library*
PCAM	Particionamento, Comunicação, Aglomeração e Mapeamento
PCI	*Peripheral Component Interconnect*
PDGDM	Problema Discreto da Geometria da Distância Molecular
PGDM	Problema da Geometria da Distância Molecular
QoS	*Quality of Service*
RMS	*Resource Management System*
RNM	Ressonância Nuclear Magnética
SAS	*Situational Awareness System*
SIG	Sistemas de Informação Geográfico
SIMD	*Single-Instruction Multiple-Data*
SMP	*Symmetric Multiprocessing*
SPS	*Self-Regenerative Systems*
SPU	*Synergistic Processor Unit*
TI	Tecnologia da Informação
UMS	*User Management System*
VO	*Virtual Organization*
VRE	*Virtual Research Environment*
WLCG	*Worldwide LHC Computing Grid*

Sumário

Lista de Figuras **xiv**

Lista de Tabelas **xvi**

Lista de Algoritmos **xvii**

1 Introdução **1**

1.1 Objetivo 5

1.2 Contribuição 6

1.3 Organização do texto 9

2 Revisão Bibliográfica **11**

2.1 Evolução das Infraestruturas para Computação em Alto Desempenho . . . 11

2.1.1 A Era *PetaScale* 14

2.1.1.1 Supercomputadores Atuais 14

2.1.1.2 Grades de Computadores 15

2.1.1.3 Nuvens de Computadores 16

2.1.2 O Futuro das e-Infraestruturas 17

2.2 Conceito de Computação Autônoma 18

2.2.1 Auto-tuning 22

2.3 Sistemas Gerenciadores 22

2.3.1 Nimrod/G 24

2.3.2 Gridbus 25

2.3.3 HTCondor 26
2.3.4 GLite 28
2.3.5 GrADS 29
2.3.6 COMPSs 29
2.3.7 EasyGrid AMS 31
2.3.8 Quadro Comparativo 32
2.4 Resumo 33

3 Projeto de Aplicações da e-Ciência Autônomas: uma Proposta 34

3.1 Motivação 35
3.1.1 AMS como Gerenciador Autônomo 37
3.1.2 EasyGrid AMS como middleware 40
3.1.2.1 Modelo de Particionamento 40
3.1.2.2 Escalonamento dinâmico 41
3.1.2.3 Tolerância a falhas 43
3.2 Projeto de Aplicações Paralelas 43
3.2.1 Classificação de Aplicações Científicas Paralelas 46
3.3 Metodologia de Projeto de Aplicações Autônomas 47
3.3.1 Autoconfiguração das Aplicações 50
3.3.2 Parâmetros de Configuração de Aplicações 51
3.4 Casos de Estudo na e-Ciência 52
3.5 Critério Geral de Avaliação 53
3.6 Resumo 54

4 Simulação Monte Carlo – Thermions 55

4.1 Método Monte Carlo 55
4.1.1 Caso de Estudo: Thermions 56

4.2 Simulações Monte Carlo em Paralelo 57
4.3 Simulações Monte Carlo Autônomas 59
4.4 Avaliação Experimental 61
4.4.1 Descrição dos Cenários Paralelos 61
4.4.2 Resultados 62
4.5 Resumo 66

5 Busca em Árvore Branch-and-prune – PDGDM 68

5.1 Algoritmos *Branch-and-Prune* 69
5.1.1 Exemplo Sequencial 69
5.1.2 Soluções Paralelas 70
5.2 *Branch-and-prune* Autônomo 72
5.3 Caso de Estudo: o Problema Discreto de Geometria de Moléculas 75
5.4 Avaliação Experimental 76
5.4.1 Descrição dos Cenários Paralelos 77
5.4.2 Resultados no Cenário 1 78
5.4.3 Resultados nos Cenários 2 e 3 81
5.5 Resumo 84

6 Decomposição de Domínio Fortemente Acoplado – N-corpos 85

6.1 Caso de Estudo: o Problema N-corpos 85
6.1.1 Algoritmo *Ring* 86
6.2 Algoritmo *Ring* Autônomo 88
6.2.1 Escalonamento Dinâmico 92
6.3 Avaliação Experimental 92
6.3.1 Teste de Sobrecarga 93
6.3.2 Teste em Ambiente Compartilhado 94

6.3.3 Teste do Escalonamento Dinâmico por Colunas 97

6.4 Resumo 99

7 Workflow Científico – Modelagem de Nicho Ecológico 100

7.1 Caso de Estudo: Modelagem de Nicho Ecológico 101

7.1.1 OpenModeller 102

7.1.2 Ferramenta de Projeção Paralela 104

7.2 Sistemas de *Workflow* Científico 105

7.3 ENM e Sistemas de *Workflow* Científico 106

7.3.1 Script Meta-escalonador 108

7.3.2 HTCondor DAGMan 108

7.3.3 EasyGrid AMS 109

7.4 Experimentos e Avaliação 110

7.4.1 Cenário do Experimento 110

7.4.2 Resultados 112

7.5 Resumo 116

8 Projeção em Domínio Geoespacial – Projeção do Modelo de Nicho Ecológico 118

8.1 Caso de Estudo: Projeção do Modelo de Nicho Ecológico 119

8.2 Versão MPI Original 121

8.3 Versão Autônoma com EasyGrid AMS 123

8.4 Avaliação Experimental 126

8.4.1 Experimento 1 127

8.4.2 Experimento 2 129

8.5 Resumo 131

9 Conclusão 133

9.1 Trabalhos Futuros 136

Referências **138**

Lista de Figuras

1.1 Camadas convencionais de um sistema gerenciador. 5

2.1 Elemento autônomo. 21

2.2 Níveis de camada de sistemas gerenciadores ou *middlewares*. 23

3.1 Comparação dos sistemas gerenciadores RMS, UMS e AMS. 38

3.2 Modelos de particionamento do EasyGrid AMS. 41

3.3 Hierarquia do escalonamento dinâmico do EasyGrid AMS. 42

3.4 PCAM: uma metodologia de projeto de algoritmos paralelos [44]. 44

3.5 Classificação das aplicações paralelas de acordo com as etapas PCAM. . . . 47

3.6 Arquitetura de camadas do EasyGrid AMS. 49

4.1 Resultados no cenário 1 para a aplicação *Thermions*. 63

4.2 Resultados no cenário 2 para a aplicação *Thermions*. 64

4.3 Resultados no cenário 3 para a aplicação *Thermions*. 66

5.1 Exemplo de particionamento tradicional de uma árvore de busca. 71

5.2 Exemplo de particionamento de uma árvore de busca usando o modelo 1Ptask. 72

5.3 Exemplo de sequência de átomos consecutivos no *backbone* de uma proteína. 76

5.4 Comparação da eficiência entre as versões EasyGrid AMS e tradicional no cenário 1 para a aplicação PDGDM. 80

5.5 Quantidade de tarefas da aplicação PDGDM com o EasyGrid AMS. 82

5.6 Comparação da eficiência normalizada entre as versões EasyGrid AMS e tradicional nos cenários 1, 2 e 3 para a aplicação PDGDM. 83

6.1 Estrutura dos processos para o algoritmo *Ring* usado com a aplicação N-corpos em 1 *time step*. 87

6.2 Estrutura das tarefas para o algoritmo *Ring* AMS usado com a aplicação N-corpos em 1 *time step*. 89

6.3 Exemplo da estrutura das tarefas para o algoritmo *Ring* AMS em 2 *time steps* com $W = 2$ no *time step* $I = 1$ e $W = 4$ no *time step* $I = 2$. 91

6.4 Sobrecarga do algoritmo *Ring* AMS em relação ao algoritmo *Ring*. 94

6.5 Número de tarefas por iteração para a aplicação N-corpos. 95

6.6 Comparação das propostas *Ring* em um ambiente compartilhado e dinâmico. 96

6.7 Comparação dos tempos de execução de cada *time step* para EDI e EDC. . 98

7.1 Demonstração da modelagem de nicho ecológico [93]. 101

7.2 *Workflow* OpenModeller. 104

7.3 Tempo de execução (em segundos) do algoritmo paralelo original *om_project*.105

7.4 Requisições OMWS de cada sistema de gerenciamento de *workflow*. 107

7.5 *Workflow* do *openModeller*. 111

7.6 *Workflow* openModeller com projeção em paralelo. 111

7.7 *Script* meta-escalonador *versus* HTCondor DAGMan *versus* EasyGrid AMS.112

7.8 Escalonamento obtido pelo *script* meta-escalonador usando o *workflow* com processos sequenciais. 113

7.9 Escalonamento obtido pelo *script* meta-escalonador com o *workflow* paralelo B. 114

7.10 Escalonamento obtido pelo HTCondor DAGMan com o *workflow* paralelo B.115

7.11 Escalonamento obtido pelo EasyGrid AMS com o *workflow* paralelo B. . . 116

8.1 Exemplo de modelagem e projeção de um nicho ecológico [93]. 119

8.2 Tempo de execução de cada bloco fixo do domínio da projeção ENM. . . . 124

8.3 Comparação percentual do ganho da versão EasyGrid AMS em relação a versão original MPI. 130

8.4 Eficiência da projeção para os algoritmos ENVDIST e SVM usando 24, 48, 96 e 128 CPUs. 131

Lista de Tabelas

2.1 Resumo das características autônomas dos sistemas gerenciadores. 32

3.1 Resumo das características dos grupos de aplicações tratados nesta tese. . 53

4.1 Principais resultados de desempenho das abordagens tradicional e EasyGrid AMS para a aplicação *Thermions* no cenário 1. 62

4.2 Principais resultados de desempenho das abordagens tradicional e EasyGrid AMS para a aplicação *Thermions* no cenário 2. 64

4.3 Principais resultados de desempenho das abordagens tradicional e EasyGrid AMS para a aplicação *Thermions* no cenário 3. 65

5.1 Informação sobre as instâncias PDGDM usadas nos experimentos. 77

5.2 Principais resultados de desempenho no cenário 1 para o PDGDM. 79

5.3 Média do número de tarefas da aplicação PDGDM com o EasyGrid AMS. 81

6.1 Médias dos tempos de execuções de um *time step* para os algoritmos *Ring* e *Ring* AMS no teste de sobrecarga. 93

6.2 Configuração do ambiente compartilhado para experimentos com o N-corpos. 95

6.3 Tempo de execução em segundos de cada iteração para cada combinação de uso de escalonamento dinâmico e maleabilidade. 96

6.4 Configuração do ambiente para o teste de comparação entre EDI e EDC. . 97

7.1 *Speed-ups* da versão paralela do *om_project*. 104

7.2 Tempo sequencial em segundos de cada fase e cada instância do *workflow*. 112

8.1 Resumo dos resultados comparativos entre as versões original com MPI e autônoma com o EasyGrid AMS para a projeção ENM (parte 1/2). 128

8.2 Continuação da Tabela 8.1 (parte 2/2). 129

8.3 Tempos em segundos (*map* e *reduce*) obtidos para ENVDIST e SVM. . . . 130

Lista de Algoritmos

1 Algoritmo básico de uma simulação Monte Carlo. 56

2 Algoritmo do processo mestre para versão tradicional de uma simulação Monte Carlo. 58

3 Algoritmo do processo trabalhador para versão tradicional e EasyGrid AMS de uma simulação Monte Carlo. 59

4 Algoritmo da tarefa mestre para versão EasyGrid AMS de uma simulação Monte Carlo. 60

5 Algoritmo sequencial *branch-and-prune*. 70

6 Algoritmo *branch-and-prune* paralelo e autônomo. 74

7 Algoritmo *Ring* para o problema N-corpos. 88

8 Algoritmo *Ring* AMS para o problema N-corpos. 90

9 Algoritmo sequencial de projeção do modelo de um nicho ecológico. 120

10 Algoritmo usado pelo processo *MestreD* da versão original MPI da aplicação de projeção ENM. 121

11 Algoritmo usado pelo processo *MestreR* da versão original MPI da aplicação de projeção ENM. 122

12 Algoritmo usado pelos processos trabalhadores da versão MPI original para a aplicação de projeção ENM. 123

13 Algoritmo usado pelas tarefas *map* da aplicação de projeção ENM com o EasyGrid AMS. 126

Capítulo 1

Introdução

A computação é atualmente considerada um dos três pilares da pesquisa científica juntamente à teoria e à experimentação. Ela tem se tornado uma ferramenta importante para a pesquisa da atualidade, contribuindo, por exemplo, para a compreensão do comportamento dos fenômenos fundamentais da natureza e para a exploração de sistemas complexos com bilhões de componentes, como o ser humano. Muitos avanços científicos foram conquistados graças ao uso da computação, seja para auxiliar ou para solucionar um problema científico. Previsão do tempo mais precisa, desenvolvimento de novos produtos farmacêuticos, otimização de rotas de fuga em prédios de grandes dimensões, melhores projetos automobilísticos, ampliação do conhecimento sobre o Universo e previsões de catástrofes mais confiáveis, são alguns exemplos de problemas científicos que obtiveram avanço nos últimos anos.

Uma aplicação científica é representada pela solução computacional atribuída a um problema científico. Duas etapas de desenvolvimento podem ser destacadas: a científica e a de controle. No *desenvolvimento científico*, o objetivo é elaborar um algoritmo que satisfaça os requisitos do problema científico. No *desenvolvimento de controle*, o objetivo é mapear a execução do algoritmo para um sistema computacional.

O termo *aplicações da e-Ciência* (*e-Science*) refere-se a aplicações científicas que requerem o uso de computação em alto desempenho devido à grande demanda de computação e processamento de dados. Muitas vezes, elas necessitam da análise de uma grande quantidade de dados ou de simulações numéricas de grandes dimensões, e apresentam o perfil de consumir uma alta quantidade de processamento, memória e/ou entrada e saída. A seguir, alguns exemplos de aplicações da e-Ciência são apresentados.

- A confirmação da existência do bóson de Higgs [1] (partícula elementar que ajuda

a explicar a formação do Universo) foi realizada através do LHC, um acelerador de partículas pertencente a CERN [125]. Ele é capaz de colidir partículas em aproximadamente 600 milhões de vezes por segundo, o que o faz gerar cerca de 30 *Petabytes* de dados anualmente para serem armazenados e analisados pelo WLCG – *Worldwide LHC Computing Grid*.

- A modelagem geográfica de espécies [26], usada para entender a sobrevivência das espécies em seus nichos ecológicos, realiza processamento intensivo sobre dados geográficos, podendo demorar meses processando.

- Simulações da formação de galáxias podem gastar mais de 2000 anos para explicar a evolução do Universo [120]. Tais simulações costumam envolver regiões espaciais tridimensionais de tamanho na ordem de 10^{10} anos-luz.

- O problema do dobramento de proteínas (*Folding Protein Problem*) [43] permite compreender a função das proteínas que podem ser usadas para desenvolver medicamentos especiais para o tratamento ou cura de doenças. Simulações computacionais do dobramento de proteína são feitas e costumam demorar cerca de 1 dia para processar o dobramento natural de uma proteína que dura 50 nanosegundos. No entanto, o dobramento de muitas proteínas é realizado em 1 milissegundo, o que faria o processamento computacional ser de aproximadamente 60 anos.

- Simulações de terremoto são realizadas computacionalmente para ajudar a prevê-los e possibilitar, por exemplo, que engenheiros construam estruturas prediais seguras. Simulações computacionais de terremotos em uma região como a do sudeste da Califórnia/EUA (com um volume de $810 \times 405 \times 85$ km), poderiam durar meses executando o processamento [27].

Cada vez mais, cientistas de diversas áreas como física, biologia, medicina, engenharia e economia, por exemplo, estão recorrendo à computação em alto desempenho para adquirir uma melhor eficiência no tempo de execução e qualidade da solução de suas aplicações científicas. Isto ocorre não só porque a escala de poder computacional vem aumentando em uma taxa crescente [115] mas também porque as infraestruturas para alto desempenho estão tornando-se mais acessíveis (mais baratos) aos cientistas em geral. As *e-infraestruturas* ou *cyber-infraestruturas* computacionais, como grades e nuvens de computadores (agregados virtuais de unidades de computação), são responsáveis pela redução de custo e aumento da disseminação entre as comunidades científicas. Enquanto relativamente baratas, elas trazem características, como heterogeneidade e compartilhamento,

que dificultam o desenvolvimento de aplicações. Além disso, é comum que, durante a execução das aplicações, a quantidade de recursos varie, caracterizando uma configuração dinâmica do sistema.

Nos dias de hoje, de acordo com a lista TOP500 de junho de 2014 [115], existem supercomputadores capazes de atingir alta velocidade de processamento (cerca de 33,86 PetaFLOPS pelo supercomputador chinês Tianhe-2 e 17,59 PetaFLOPS pelo supercomputador americano Titan [89]). Supercomputadores atuais são tipicamente formados por *clusters* de vários computadores conectados por rede local de alta velocidade. Multiprocessadores, processadores multinúcleos, aceleradores (gráficos para propósito geral – GPGPU – e aqueles que complementam funcionalidades do processador principal, como o Xeon Phi [108]) também podem fazer parte destes ambientes, tornando-os mais heterogêneos [94]. As grades computacionais também possuem potencial para fornecer uma grande escala de poder computacional, onde *clusters* de computadores, geralmente concentrados em instituições e empresas, podem estar conectados pela rede mundial. Outra classe de ambientes para computação em larga escala é representada pela computação em nuvem ou *cloud computing* que apresentam uma infraestrutura física de um *cluster* ou grade acoplado a um *software* que provê serviços. Tais sistemas oferecem serviços como acesso de aplicativos, serviços de armazenamento, etc., geralmente fazendo uso da virtualização para promover isolamento enquanto compartilham os recursos físicos para melhorar a eficiência e tornar a solução economicamente viável [9]. Um tipo de nuvem bastante acessível e utilizada no meio científico é chamada de *pay-per-use*. Este tipo de nuvem usa um modelo de preço conforme a expressão "pague conforme o consumo" (*pay-as-you-go*) e, assim, disponibiliza recursos computacionais ao público em geral. Um exemplo é a Amazon EC2 [5].

Cada um desses ambientes computacionais, como os já citados supercomputadores, grades computacionais e nuvem de computadores, apresentam separadamente seus problemas de gerenciamento, por terem recursos computacionais sob complexa administração, heterogêneos, suscetíveis a falhas, dinâmicos e compartilhados entre diversos usuários e aplicações. Uma possível junção destas classes de ambientes traz um crescente aumento da complexidade no gerenciamento de recursos, aplicações e usuários. As caraterísticas destas novas infraestruturas computacionais tornam o desenvolvimento de controle de aplicações da e-ciência muito mais difíceis. Na atual era *petascale* (e em um futuro próximo, a era *exascale*[1]), a quantidade de recursos necessária será dificilmente aproveitada

[1] *Petascale* e *exascale* são escalas baseadas principalmente no nível de processamento em FLOPS (operações de ponto flutuante por segundo), onde *peta* significa 10^{15} FLOPS e *exa* significa 10^{18} FLOPS.

eficientemente por aplicações científicas se suas implementações não forem repensadas.

Nesses tipos de sistemas, em termos de eficiência, todas as aplicações deveriam continuamente ter conhecimento do paralelismo e mapeamento ideal dos recursos. A computação autônoma [61, 72, 63] surge como uma alternativa promissora para prover uma boa utilização, por parte da aplicação, dos ambientes compostos pela *e*-infraestrutura. Através dela, a aplicação/sistema seria capaz de autogerenciar-se, tomando decisões baseadas em seu estado e no conhecimento do ambiente. Deste modo, a execução da aplicação/sistema poderia tornar-se mais eficiente. As principais propriedades autônomas são:

- auto-otimização – capacidade da aplicação/sistema melhorar seu desempenho durante a execução;
- autoconfiguração – a aplicação/sistema altera seus parâmetros de configuração conforme sua necessidade;
- autorrecuperação – a aplicação/sistema recupera seu estado de execução caso alguma falha ocorra;
- autoproteção – capacidade da aplicação/sistema proteger seus dados e execução, tornando-se confiável.

Desde a última década, algumas das características autônomas vêm sendo introduzidas principalmente em sistemas gerenciadores, que possuem a função de coordenar a execução de aplicações em uma determinada *e*-infraestrutura. Como uma alternativa para facilitar o desenvolvimento do controle de aplicações em tais ambientes, sistemas gerenciadores apresentam uma camada de *software* intermediária, muitas vezes chamados de *middleware*, entre os recursos da *e*-infraestrutura e a aplicação (ver Figura 1.1). É neste *middleware* que a autonomia é inserida para gerenciar a demanda dos usuários ou recursos ou aplicações para atingir algum objetivo. De fato, o ônus da etapa de desenvolvimento do controle é atribuído a um desenvolvedor especialista do sistema gerenciador, ao invés do cientista, que está preocupado apenas com o desenvolvimento científico.

É comum que a introdução de componentes autônomos no *middleware* adicione complexidade e sobrecarga ao sistema. Geralmente, os sistemas gerenciadores autônomos monitoram constantemente elementos como o uso de CPU, memória, entrada/saída e estado das máquinas. Através destas checagens contínuas, eventos fundamentais às decisões autônomas podem ser reconhecidos de modo que alguma ação cabível seja tomada au-

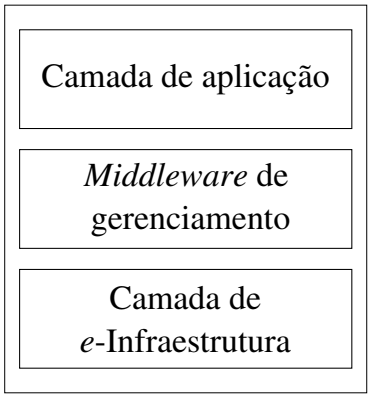


Figura 1.1: Camadas convencionais de um sistema gerenciador.

tomaticamente. Portanto, existe uma sobrecarga natural de gerenciamento que pode ser acrescida conforme o aumento da complexidade do sistema.

Na maioria dos sistemas gerenciadores presentes na literatura atual, o *middleware* é projetado de forma genérica em relação às aplicações. O objetivo destes gerenciadores é obter uma boa solução, na média, para qualquer tipo de aplicação possível. Em outras palavras, isto significa que uma grande quantidade de funcionalidades redundantes costumam ser implementadas nestes *middlewares*, causando sobrecarga desnecessária. Além disso, características próprias de uma aplicação, como, por exemplo, o particionamento dos dados, tendem a ser ignoradas pelo gerenciamento, já que são específicas do problema a ser tratado. Nesta tese, pretende-se adotar uma solução otimizada para uma determinada aplicação através de uma abordagem de gerenciamento autônomo. Assim, reduz-se a sobrecarga (a aplicação "possui somente aquilo que é necessário para ela") e as características próprias da aplicação são consideradas nas decisões do sistema, obtendo-se uma melhor eficácia de execução nas *e*-infraestruturas.

1.1 Objetivo

No cenário atual de pesquisa em computação de alto desempenho, aplicações científicas reais geralmente necessitam de um ambiente computacional de execução com a capacidade de tratar um grande volume de dados e de processamento em larga escala. No entanto, como já mencionado, tais ambientes computacionais são complexos e, portanto, difíceis de serem tratados eficientemente pelos cientistas desenvolvedores. Para permitir o uso destes sistemas, é comum introduzir uma camada de gerenciamento entre as aplicações e a *e*-infraestrutura. As abordagens atuais de gerenciamento, utilizam uma camada genérica, que implementa funcionalidades redundantes. Desta forma, o objetivo deste trabalho é

estudar o projeto de versões autônomas de aplicações científicas e comprovar experimentalmente se ele é capaz de tornar suas execuções mais eficientes em ambientes distribuídos de larga escala.

O conceito principal da tese é realizar o gerenciamento de aplicações da *e*-ciência de uma forma diferente do que vem sendo feito na literatura, que visa tomar decisões de forma generalizada por tipo de aplicações. Trabalhos de pesquisa estudados mostram que os componentes autônomos para *e*-infraestruturas são projetados através de uma camada externa à aplicação, o que não a permite adaptar-se ao ambiente. Na presente tese, as propriedades autônomas são introduzidas de acordo com as características da aplicação, através de um sistema gerenciador de aplicações, de forma a aproveitar melhor os recursos do ambiente. Portanto, as aplicações poderiam autoconfigurar-se de acordo com as mudanças nos ambientes distribuídos e usufruir da auto-otimização e autorrecuperação providas pelo *middleware*. A eficiência pode ser garantida através da autonomia concentrada na aplicação ao invés da tradicional concentração no sistema como um todo.

Mais detalhadamente, o objetivo desta tese é propor o uso de uma metodologia de desenvolvimento de modo que o processo de tornar as aplicações autônomas em sistemas distribuídos seja mais simples. Esta metodologia inclui uma etapa de particionamento mais efetivo da aplicação, uma etapa de identificação de parâmetros da aplicação, uma etapa de autoconfiguração específica (considerando os parâmetros identificados na etapa anterior) e o uso de um sistema gerenciador focado na aplicação.

1.2 Contribuição

Inicialmente, as contribuições desta tese são destacadas resumidamente a seguir e detalhadas posteriormente ainda nesta seção.

- ✓ Através de experimentação e avaliação, pode-se concluir que aplicações autônomas são viáveis e trazem benefícios executando em ambientes distribuídos.
- ✓ A implementação de 5 versões autônomas de aplicações da *e*-ciência mostram-se melhores que o estado da arte. O estado da arte considerado nesta tese engloba versões de aplicações tradicionalmente utilizadas pela respectiva comunidade científica.
- ✓ As aplicações avaliadas são representantes de diferentes classes importantes da *e*-ciência, o que permite inferir a validade da proposta a uma gama maior de aplicações. As classes são:

- Simulações Monte Carlo,
- algoritmos de busca em árvore,
- simulações com domínio fortemente acoplado,
- *workflows* científicos, e
- projeção de dados em domínio geoespacial.

✓ Uma metodologia de projeto de aplicações autônomas foi proposta e permite tornar a implementação de classes de aplicações mais simples.

✓ Uma arquitetura de aplicações autônomas foi apresentada utilizando a ferramenta EasyGrid AMS descrita no Capítulo 3.

✓ Foi incluída a autoconfiguração no EasyGrid AMS integrada ao processo de auto-otimização e autorrecuperação.

✓ Foram realizadas melhorias, atualizações e refinamentos no EasyGrid AMS.

A principal contribuição desta tese está na conclusão afirmativa da viabilidade do desenvolvimento de aplicações da *e*-ciência autônomas, comprovando que a autonomia traz grandes benefícios nas execuções de aplicações em ambientes distribuídos. Através da implementação de 5 aplicações, que são representantes de classes de aplicações da *e*-ciência, pôde-se inferir uma melhoria significante (em relação à eficiência e escalabilidade) em comparação com o estado da arte. Como já mencionado, o estado da arte considerado nesta tese engloba versões de aplicações tradicionalmente utilizadas pela respectiva comunidade científica e isto não implica que elas sejam as melhores (porém boas) soluções existentes na literatura.

Uma outra contribuição fundamental desta tese encontra-se na metodologia de projeto desenvolvida para tornar aplicações da *e*-ciência autônomas. Tal metodologia tem o objetivo de otimizar a execução de uma determinada aplicação científica sobre as *e*-infraestruturas, considerando suas características. O *middleware*, que faz o intermédio entre a aplicação e a *e*-infraestrutura, é responsável por gerenciar a aplicação e fornecer um comportamento autônomo. Assim, as propriedades de autoconfiguração, auto-otimização e autorrecuperação podem ser incorporadas a ela[2]. A metodologia indica também um particionamento efetivo da aplicação em um grande número de partições de granularidade fina, o que possibilita uma melhor integração com o *middleware*. Além disso, parâmetros

[2] A autoproteção não foi considerada nesta tese, mas ela poderá ser adicionada ao sistema futuramente.

de cada tipo de aplicação, como a granularidade e número de partições, podem ser considerados nas decisões autônomas do sistema. Isto possibilita um bom ajuste dinâmico da aplicação ao ambiente.

Seguindo a metodologia do projeto, implementações autônomas de cinco aplicações, que representam classes da *e*-ciência, destacam-se como outras contribuições. A ideia das classes é permitir que uma implementação para uma determinada aplicação pode ser utilizada para implementar outras aplicações de mesma classe. O *middleware* utilizado foi o *EasyGrid AMS*, que é responsável tornar aplicações autônomas e já oferece as propriedades de autoconhecimento (monitoramento), auto-otimização (escalonamento dinâmico) e autorrecuperação (métodos de tolerância a falhas). No entanto, só a auto-otimização e autorrecuperação não são suficientes para tratar aplicações em um ambiente compartilhado e dinâmico, visto que o grau de paralelismo é estático. É preciso que elas se moldem e se adaptem ao sistema mudando suas próprias configurações de execução. Logo, a propriedade de autoconfiguração é introduzida neste trabalho de forma integrada às outras. Assim, as propriedades autônomas são incluídas na aplicação para possibilitar execuções que a façam se autoconfigurar ao ambiente distribuído de forma otimizada e recuperável diante de cenários de falhas.

O método Monte Carlo é muito usado para simular problemas que podem ser representados por processos estocásticos em diversas áreas como engenharia, física e biologia. Tais simulações são aplicações massivamente paralelas cujas partições são independentes e executam o mesmo algoritmo. Para transformar este tipo de aplicação em autônoma basta particionar o domínio em diversas tarefas iguais de granularidade fina e utilizar o EasyGrid AMS para realizar o gerenciamento autônomo. Os bons resultados obtidos, comparados aos resultados de uma versão não autônoma comumente usada, mostram que a proposta é consistente e eficiente, mesmo em ambientes heterogêneos e compartilhados.

Algoritmos de busca em árvore do tipo *branch-and-prune* também ganharam sua versão autônoma. Tais algoritmos são muito usados em áreas como biologia, física e logística, para resolver problemas de otimização de forma exata. As aplicações que usam este algoritmo são transformadas em autônomas não só pelo emprego do *middleware* gerenciador mas também pelo particionamento dinâmico realizado de acordo com a configuração do ambiente distribuído. Os resultados mostram que a solução autônoma é mais eficiente em relação às soluções convencionais.

Simulações em geral costumam possuir um domínio de dados fortemente acoplado. Isto implica em um grande número de troca de mensagens nas versões paralelas e dis-

tribuídas. O problema N-corpos é um exemplo de simulação astrofísica que simula a evolução de um sistema físico com milhares de corpos interagindo entre si. A solução autônoma proposta para esta aplicação específica sugere a inclusão da autoconfiguração das partições de acordo com as mudanças no ambiente durante a execução. Os resultados revelam uma melhor utilização dos recursos pela aplicação quando as propriedades autônomas são empregadas.

Workflows científicos, que são experimentos executados centena de vezes com parâmetros diferentes, usam fluxos de execução de trabalho representados por um grafo direcionado acíclico. Eles são constantemente usados para resolver problemas científicos como de modelagem biológica, espacial e climática. Devido às dependências entre os trabalhos, o *middleware* necessita considerar todo o grafo para tomar decisões. Caso as decisões sejam autônomas de acordo com o sistema, um melhor desempenho é obtido comparado às soluções convencionais que não utilizam este artifício de conhecer melhor a aplicação e o sistema.

Aplicações científicas para resolver problemas de projeção de dados em domínio geoespacial são comuns nas áreas como geofísica e biodiversidade. O domínio pode ser particionado em blocos independentes cuja a quantidade de trabalho varia de acordo com o algoritmo que irá processar cada bloco. A solução autônoma proposta para este problema engloba o uso da autoconfiguração no particionamento dinâmico dos dados (já que ele varia) e a integração com um *middleware* próprio para este tipo de aplicação. Para a aplicação usada como estudo de caso, os resultados foram satisfatórios, uma vez que obteve-se um algoritmo mais eficiente para o problema comparado à versão paralela original existente.

1.3 Organização do texto

O texto está organizado de maneira que se possa ter uma descrição de trabalhos existentes na literatura sobre o tema, a explicação detalhada da proposta desta tese e um capítulo dedicado para cada tipo de aplicação da *e*-ciência considerada. A descrição do tipo bem como da aplicação usada como caso de estudo é relatada em cada capítulo respectivo. Ainda, em cada um destes capítulos, o desenvolvimento da aplicação paralela usando a proposta desta tese é apresentado junto com os experimentos realizados em um ambiente computacional simulado com características próprias das *e*-infraestruturas.

O Capítulo 2 apresenta uma visão geral das *e*-infraestruturas e sua evolução, assim

como do conceito de computação autônoma e dos sistemas gerenciadores que usam esses conceitos para gerenciar as *e*-infraestruturas. No Capítulo 3, a proposta desta tese é descrita junto com a metodologia usada para tornar aplicações da *e*-ciência autônomas. Como já mencionado, os próximos capítulos são dedicados para tratar cada tipo de aplicação científica, sua implementação empregando a proposta e a avaliação experimental de desempenho. O Capítulo 4 apresenta as simulações Monte Carlo como um tipo de aplicação da *e*-ciência massivamente paralelo e a aplicação chamada de *Thermions* como caso de estudo. No Capítulo 5, o destaque são para aplicações que usam métodos de busca exaustiva e a aplicação para o problema de encontrar a estrutura tridimensional de moléculas chamado de *PDGDM*. Aplicações que utilizam a decomposição de domínio fortemente acoplado, como N-corpos, são descritas no Capítulo 6. *Workflows* científicos são tratados no Capítulo 7 e a aplicação usado como estudo de caso é a *modelagem de nicho ecológico*. O Capítulo 8 descreve aplicações que realizam projeção de informações em um domínio geoespacial e foca na aplicação de *projeção de modelos de nicho ecológico* geograficamente. Por fim, o Capítulo 9 apresenta as conclusões finais obtidas através das avaliações experimentais em cada aplicação e os trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Este capítulo compreende os variados pontos relacionados ao tema deste trabalho. O tema trata de como resolver o problema de modelar a execução de aplicações da *e*-ciência para tirarem proveito do potencial de ambientes computacionais distribuídos de larga escala. Primeiramente, o capítulo descreve brevemente a evolução arquitetural dos sistemas computacionais distribuídos desde a época do aparecimento de supercomputadores e *clusters* até os dias de hoje, citando seus componentes e suas futuras integrações. Posteriormente, diversos pontos de vista sobre o conceito de computação autônoma são apresentados como uma área a ser explorada como possível solução para o problema. Por último, são descritos alguns sistemas gerenciadores que já incluem alguns conceitos de autonomia em suas soluções para gerenciar tais ambientes distribuídos e prover meios mais eficientes de executar as aplicações científicas.

2.1 Evolução das Infraestruturas para Computação em Alto Desempenho

Nas últimas décadas, e continuamente, as infraestruturas computacionais vêm desenvolvendo-se de um modo acelerado para a computação de alto desempenho. A partir da década de 60, supercomputadores eram construídos com múltiplos processadores para a realização de cálculos intensivos com o uso de memória compartilhada, mas eram considerados caros e de difícil acesso à comunidade científica. Como alternativa mais barata e mais acessível, aproximadamente na década de 90, *clusters* de computadores eram montados utilizando-se computadores pessoais interconectados por rede local. Inicialmente eram formados por computadores homogêneos e dedicados mas, logo, diferentes computadores eram usados e muitas vezes havia compartilhamento de recursos. Já no fim da década de

90 e início da década de 2000, o conceito de grades computacionais era estabelecido como forma de aumentar o poder computacional e prover diversos serviços a aplicações de alto desempenho, assim como para outras áreas afins.

Ao mesmo tempo que as grades computacionais eram definidas, via-se que a frequência de processamento não poderia ser aumentada já que ficava limitada fisicamente pelo superaquecimento dos transistores cada vez mais densos. No entanto, a produção de processadores multinúcleos ainda permitia a introdução de mais transistores por *chip*, mantendo a lei de Moore[1] válida [73, 53].

Em meados da década de 2000, iniciou-se, então, a produção de multiprocessadores e processadores multinúcleo. Multiprocessadores envolvem dois ou mais processadores idênticos que são conectados a uma única memória principal e são gerenciados por um único sistema operacional. A maioria deles usa arquitetura SMP (*symmetric multiprocessing*). Processadores multinúcleos ou *multicores* são aqueles que possuem no mesmo *chip* do processador mais de um núcleo (*core*) de processamento, e, portanto, também são gerenciados por um único sistema operacional. A arquitetura SMP é aplicada a eles, sendo que cada núcleo é tratado como um processador. Além disso, níveis de hierarquia maiores de memória *cache* estão sendo gerados para melhorar o desempenho de acesso a memória; por exemplo, utilizar uma memória *cache* L1 (nível 1) para cada núcleo de processador e uma L2 (nível 2) para acesso de todos os núcleos no mesmo *chip*. Em relação ao sistema operacional, que gerencia a execução de processos e *threads* nos processadores e núcleos, existe uma constante evolução no desenvolvimento em seu escalonador. Isto mostra que não há apenas heterogeneidade a nível de *hardware* mas também a nível de *software*.

A partir da década de 2000, aceleradores como as placas gráficas para propósito geral começaram a ser usados no processamento em alto desempenho. Com o advento da tecnologia GPGPU - *General-Purpose Computing on Graphics Processing Units* [94]. GPUs As placa gráficas não eram usadas apenas para processar gráficos mas também serviam para realizar processamento de uso geral. Linguagens próprias para programação em GPU se popularizaram como OpenCL e CUDA. Em 2006, a GPU GeForce 8800 da empresa NVidia obteve a capacidade de ser programável em linguagem C sob o modelo paralelo CUDA. O desenvolvimento de aplicações de propósito geral que antes era feito expressando a computação não gráfica através de API gráfica como OpenGL, neste momento passou a ter uma maneira mais simples de programar, embora obrigue o programador a

[1]Lei de Moore reflete uma profecia do presidente da Intel em 1965, Gordon E. Moore. Ela dizia que o número de transistores de *chips* de processamento teria um aumento de 100%, pelo mesmo custo, a cada período de 18 meses [105].

seguir o modelo CUDA [88]. Por volta de 2008, outro modelo chamado OpenCL permitiu ainda mais a popularização das GPUs para programação paralela, já que é um padrão aberto.

A partir de 2005, uma diferente microarquitetura de microprocessadores multinúcleo, chamada *Cell*, foi lançada com o objetivo principal de acelerar aplicações multimídia, de imagens digitais e de processamento de vetor [60]. *Cell* é um *chip* multiprocessado heterogêneo que consiste de um núcleo Power IBM de 64 bits, aumentado com 8 coprocessadores especializados baseados na arquitetura SIMD (*Single-Instruction Multiple-Data*). Tais coprocessadores são chamados de SPU (Synergistic Processor Unit) e são conectados ao núcleo principal através de um barramento de dados de banda larga e circular. Em 2007, o *console* para jogos *Playstation 3* foi lançado também com esta arquitetura do tipo *cell*. Ele é provavelmente um dos sistemas baseados em *cells* mais barato no mercado, possui um número reduzido de coprocessadores (de 8 para 6), 256 MB de memória, uma placa de vídeo NVidia e uma placa de rede *Gigabit Ethernet* [75].

Em 2008, um nova arquitetura de aceleradores foi anunciado pela Intel com o nome de Intel MIC (Intel Many Integrated Core). Ao invés de processadores serem multinúcleos (geralmente com até 8 núcleos por *chip*), os processadores passam a ter muitos núcleos ou *many-cores* (mais de 50 núcleos em um *chip* é considerado *many-core*). Esta arquitetura permite concentrar mais de 40 núcleos de processamento em um *chip* [106]. Larrabee foi o primeiro projeto a ser lançado pela Intel usando a arquitetura MIC. Ele mescla características de CPU e GPU e foi projetado para processamento de gráficos e de alto desempenho. Em relação à CPU, Larrabee usa arquitetura compatível com x86 e hierarquia de *cache*, enquanto em relação à GPU, há a herança do uso de unidades de vetorização SIMD e *hardware* para amostragem de textura. Seguindo esta tecnologia, o acelerador Xeon Phi foi lançado comercialmente em 2012 pela Intel [67]. Não mais interessada em processamento gráfico, como no caso da arquitetura Larrabee, a Intel fabricou um acelerador para apenas processamento de alto desempenho. Um acelerador Xeon Phi possui conexão com um processador Intel Xeon através de uma porta PCI *express* e, como é possível rodar um sistema operacional Linux, é visto como uma máquina com seu próprio IP virtualizado. Tal acelerador possui cerca de 61 núcleos e cada um deles possui um *cache* L2 e uma unidade de processamento de vetor SIMD associados. Eles são conectados entre si através de um barramento de banda larga bidirecional em forma de anel.

2.1.1 A Era *PetaScale*

Nos dias de hoje, existem computadores com apenas um processador, com processadores multiprocessados, podendo muitas vezes usar aceleradores como GPUs e Xeon Phi. Tais computadores podem ser interconectados por diversos tipos de rede como *Ethernet* (10 *Mbit/s*, *Fast*, *Gigabit* e *10-Gigabit*), *Infiniband*, por exemplo, e até a Internet. Neste contexto, tais sistemas formam e-infraestruturas ou *cyber*-infraestruturas [8] como os atuais supercomputadores, as grades de computadores e as mais atuais nuvens de computadores. Estas infraestruturas designam a atual era *petascale*, que compreende a faixa de uso de PetaFLOPS.

2.1.1.1 Supercomputadores Atuais

Supercomputadores atuais são aglomerações de computadores com ou sem aceleradores para processamento massivo geralmente interconectados por uma rede de alta capacidade e com topologia especial. Na maioria dos casos, os computadores são projetados para chassis de servidor e instalados em *racks* ou gabinetes. Vários *racks* podem ser usados para aumentar a capacidade do supercomputador. O Top500 [115] oferece uma lista dos 500 supercomputadores mais rápidos. Desde 2008, a escala de petaFLOPS é alcançada. Os primeiros lugares nos últimos anos exemplificam bem a variedade de tipos de aglomerações de computadores, incluindo as tecnologias de processadores multiprocessados, *cells*, GPUs e Intel MIC.

O supercomputador Roadrunner da IBM foi o primeiro a alcançar a escala de processamento na ordem de petaFLOPS em 2008 (desempenho de 1,1 PetaFLOPS) e representa um híbrido de processadores *cell* e processadores multinúcleos [10]. Ele é composto por um híbrido de 12960 processadores *cell* PowerXCell 8i e 6480 processadores AMD Opteron de dois núcleos.

Tianhe-1A (localizado no Centro Nacional de Supercomputação em Tianjin, China), um híbrido de processadores multinúcleos e GPU, alcançou o primeiro lugar da lista em 2010 e obteve o desempenho de 2,6 PetaFLOPS [130]. O sistema é composto de 112 gabinetes de computadores, 12 gabinetes de dispositivos para armazenamento, 6 gabinetes de dispositivos para comunicação e 8 gabinetes de dispositivos de entrada e saída. Cada gabinete de computadores é composto por 4 quadros, com cada um contendo 8 *blades* (mais um quadro de *switch* de 16 portas). Cada *blade* é composto de 2 nós de computadores que contém, cada, 2 processadores de 6 núcleos e uma GPU Nvidia modelo M2050. O

sistema tem no total 7168 GPUs e 86016 núcleos de CPU.

Titan (do laboratório *Oak Ridge National Laboratory* nos Estados Unidos), outro híbrido de processadores multinúcleos e GPU, alcançou o primeiro lugar do Top500 em 2012 e obteve o desempenho de 17,6 petaFLOPS. Ele é composto por 18688 nós, sendo 4 nós por *blade* e 24 *blades* por gabinete, e cada nó apresenta um processador de 16 núcleos e um GPU Nvidia Tesla K20X.

O supercomputador que encontra-se em primeiro lugar atualmente é o Tianhe-2, desenvolvido pela Universidade Nacional de Tecnologia de Defesa da China. O sistema é formado por 16000 nós, cada um com dois processadores *Ivy Bridge Xeon* (6 núcleos cada) e 3 chips Xeon Phi (61 núcleos cada), totalizando 3,12 milhões de cores. Em novembro de 2013 ele alcançou cerca de 33,9 PetaFLOPS na avaliação do Top500.

2.1.1.2 Grades de Computadores

Grade de computadores ou um ambiente de *grid computing* é uma infraestrutura tipicamente formada por recursos de computadores de diferentes tipos, localidades e administrados de forma distribuída [46]. Uma grade é composta por vários computadores heterogêneos interconectados por vários tipos de rede, incluindo a *Internet*. Grupos de computadores podem estar sob diferentes domínios administrativos, fazendo necessário o uso de autenticação e identificação dos usuários. Falhas de sistema são mais frequentes e os computadores apresentam diferentes configurações de *software* e *hardware*, como sistemas operacionais, memória e processadores. Além disso, os computadores são compartilhados e a garantia de dedicação exclusiva é praticamente improvável.

O LHC Grid [125] é um exemplo de grades computacionais de maior escala mundial atualmente. Ele engloba mais de 170 centros computacionais em 36 países para processar, analisar e armazenar cerca de 30 *Petabytes* de dados gerados anualmente pelo acelerador de partículas LHC, um projeto do CERN. O sistema distribuído apresenta 4 níveis de serviço: o nível 0 é formado pelo centro CERN e ele é responsável por distribuir os dados e juntar as respostas dos centros do nível 1; o nível 1 é formado atualmente por 11 grandes centros de computação da Europa, Ásia e América do Norte e é responsável por também armazenar dados, processar e agrupar as saídas do nível 2; o nível 2 costuma ser formado por universidades e outros institutos científicos (cerca de 140) que são capazes de armazenar dados e prover poder computacional para específicas tarefas de análise; e o nível 3 é formado por cientistas individuais que colaboram com o sistema. A conexão entre os centros do nível 1 é realizada por uma rede óptica privada e a comunicação entre

os centros de nível 1, 2 e 3 é feita por redes de propósito geral, como a Internet.

Um outro tipo de grades computacionais é a voluntária. Neste tipo de grade, os recursos são associados ao sistema espontaneamente por usuários e tendem a ser relativamente menos estruturados. Os computadores associados podem ser tanto computadores pessoais quanto *clusters* e supercomputadores de alguma instituição. É difícil saber quantos computadores e supercomputadores estão associados ao sistema devido ao seu alto grau de dinamismo, mas eles são capazes de alcançar picos de processamento assim como supercomputadores. Por exemplo, o projeto Folding@Home [43], à frente dos sistemas da BOINC [17], usa esta abordagem e chega a alcançar cerca de 34,3 PetaFLOPS de processamento. Os cálculos processados pelo projeto servem para realizar simulações de enrolamentos de proteínas, que contribuem para a comunidade científica entender melhor o desenvolvimento de variadas doenças, como Alzheimer e fibrose cística.

2.1.1.3 Nuvens de Computadores

Uma nuvem de computadores ou um ambiente de *cloud computing* [123, 47] utiliza uma infraestrutura semelhante a de uma grade computacional que permite aos usuários acessar serviços, na maioria das vezes, remotamente através da Internet e usando tecnologias de virtualização. Alguns serviços geralmente fornecidos são infraestrutura, plataforma, *software* e banco de dados. Quando uma nuvem é projetada para o acesso de usuários em geral, que frequentemente pagam pelo serviço conforme o consumo, ela é chamada de pública. Quando os serviços são apenas estabelecidos para uma determinada organização, está nuvem é chamada de privada [9].

As nuvens computacionais públicas estão sendo usadas recentemente para a computação de alto desempenho. Por exemplo, a *Amazon Web Service* [5] oferece a infraestrutura *Amazon Elastic Compute Cloud* como ambiente de execução de aplicações nas nuvens [40, 68]. Recursos de *clusters* de CPU e GPU podem ser usadas pelo usuário como serviço para a alocação de um ambiente de alto desempenho. Para exemplificar o poder da infraestrutura da *Amazon Elastic Compute Cloud*, um de seus *clusters* obteve o 64º lugar na lista Top500 em novembro de 2013, com desempenho de cerca de 484,2 TeraFLOPS. Atualmente, ocupa a 76ª posição na lista. Como outros nós da infraestrutura podem ser adicionados ao sistema, a escala de petaFLOPS pode ser alcançada.

2.1.2 O Futuro das e-Infraestruturas

Para atingir o próximo planalto de desempenho, que seria a era *ExaScale*, as *e*-infraestruturas do futuro poderão englobar características destas três classes de ambientes: supercomputadores, grades ou nuvens de computadores [33, 34]. Todas essas tecnologias podem compor uma *e*-infraestrutura utilizada para processamento de alto desempenho como resolver diversos problemas científicos de várias áreas. Deste modo, existem muitos desafios a serem tratados pelos cientistas usuários de tal *e*-infraestrutura:

Heterogeneidade: a infraestrutura é composta por computadores e dispositivos diferentes e são interconectados por diversos tipos de rede com distintas bandas. Eles podem possuir processadores, quantidade de núcleos, memória, níveis de memória, armazenamento de dados, tudo diferente. Além disso, os sistemas operacionais e programas instalados não são necessariamente os mesmos e, na maioria das vezes, não são.

Compartilhamento: em sistemas computacionais onde existem diversos domínios administrativos, como grades computacionais, o compartilhamento de recursos é inevitável. Dificilmente todo o poder computacional existente no sistema estará completamente disponível para a execução de uma aplicação.

Dinamismo: o ambiente computacional muda continuamente durante a execução da aplicação. Computadores podem tornar-se indisponíveis em alguns momentos e novos computadores podem ser incluídos no ambiente de execução da aplicação. Além disso, novas aplicações podem surgir no sistema e possivelmente todas as aplicações deverão interagir entre si em modo de compartilhamento.

Suscetibilidade a falhas: por ser um ambiente de grande escala computacional, falhas de *software* e *hardware* passam a ser mais frequentes. Computadores podem ficar inacessíveis, assim como as limitações (como o uso de memória, tempo de processamento, quota de armazenamento, e outros) estabelecidos pelo sistema operacional podem interromper abruptamente processos de execução das aplicações.

Incerteza: há dificuldade em se prever futuros acontecimentos no sistema, principalmente se considerar o sistema completamente. Não há um conhecimento global do sistema, dado que ele é descentralizado e assíncrono.

Segurança: por conta da infraestrutura fazer parte de um grupo de diferentes organizações e pelo fato do dono da aplicação desejar proteger seus dados e execução, a

segurança para prover autenticação, autorização, controle de acesso e controle dos dados é um desafio relevante.

2.2 Conceito de Computação Autônoma

Computação autônoma é apresentada neste trabalho como uma solução para que aplicações científicas consigam tirar proveito da complexa *e*-infraestrutura eminente e brevemente futura. Nos próximos parágrafos desta seção, serão descritos diversos pontos de vista do conceito de computação autônoma.

Um *survey* [63] publicado em 2008 apresenta um breve histórico da computação autônoma. O histórico é traçado a partir de 1997, por um projeto militar do DARPA - chamado SAS (*Situational Awareness System*) - que visa criar um autogerenciamento de uma rede sem fio com nós móveis [99]. A ideia é adaptar as rotas, frequência e banda de acordo com as mudanças de topologia da rede. Em 2000, outro projeto do DARPA chamado DASADA [51] faz medições e sondas para monitorar o sistema e, através dos dados coletados, planeja e aciona ordens para otimizar e tratar falhas de algum componente. Em 2001 foi introduzido o manifesto da IBM, momento em que o termo "*autonomic computing*" foi realmente usado como comparação ao sistema nervoso autônomo. Ainda neste histórico outro projeto do DARPA surge em 2003 com o nome de SPS - *Self-Regenerative Systems* [103]. Seu objetivo é tratar falhas do sistema e ataques maliciosos. Por último, o ANTS - *Autonomous NanoTechnology Swarm* - da NASA [117] é mencionado como sendo um sistema de sensores autônomo no espaço. O objetivo é fazer com que pequenos veículos espaciais, ao entrar no cinturão de um asteróide, se comuniquem entre si sem precisar de intervenções humanas (o que seria praticamente inviável, já que o atraso para uma mensagem chegar do espaço a Terra é muito grande) e tomem decisões sobre qual asteróide é de interesse.

O *survey* também descreve os graus de autonomia propostos pela IBM [64], indo do nível 1 ao 5. Os níveis são:

1. Básico: o sistema é configurado, monitorado e ações são tomadas manualmente pelo administrador.

2. Gerenciado: o sistema é monitorado e apresenta os dados de forma consolidada para o administrador, mas as ações tomadas ainda são manuais. Isto reduz o tempo que leva para o administrador coletar e gerar informação.

3. Preditivo: o sistema é capaz de reconhecer padrões e através deles, prever uma boa configuração e sugerir ações convenientes ao administrador. O administrador se torna mais confiante a aceitar as ações pois elas provem de diversos estudos da área de Inteligência Artificial.

4. Adaptativo: o sistema já é capaz de aplicar as ações automaticamente de acordo com as medições realizadas.

5. Autônomo: um sistema é completamente autônomo quando ele é autogerenciado através de políticas e objetivos. Os usuários poderão interagir com o sistema para alterá-los.

O artigo da IBM publicado em 2001 [61], por muitos chamado de manifesto, indica que a computação autônoma deve vir como uma forma de aliviar a crise da complexidade das infraestruturas. Enquanto os sistemas vão se tornando cada vez mais complexos (devido aos diferentes tipos de computadores com diversos tipos de memória, processadores, sistemas operacionais), o gerenciamento por intervenção humana fica cada vez mais inapropriado. Para resolver o problema da complexidade de manter e gerenciar a infraestrutura é preciso criar sistemas mais complexos.

Há a necessidade dos sistemas de computação serem autônomos assim como o sistema nervoso humano, que é capaz de se manter e governar automaticamente. Por exemplo, quando um ser humano se machuca, seu organismo tem a capacidade de reconhecer o ferimento e curar aquela região. Ou quando o corpo humano sente frio, começa a tremer para gastar mais energia e melhorar seu aquecimento. O sistema nervoso humano consegue tomar decisões sem a intervenção consciente do cérebro.

Horn [61] propõe que o sistema autônomo possua ao menos os seguintes oito elementos chaves ou características:

- autoconhecimento - o sistema deve conhecer seus componentes, suas características e todas as conexões com outros sistemas.

- auto-otimização - para atingir certas metas do sistema ou simplesmente manter eficiente sua execução, o sistema deve obter informações através de monitoramento, avaliar o estado e tomar decisões de modo a otimizar o sistema, isto é, torná-lo mais robusto diante do dinamismo. É importante saber com que frequência fazer as medições e agir no sistema, além de como as ações irão impactar em todo o sistema.

- autoconfiguração - devido às mudanças no ambiente, o sistema precisa reconfigurar automaticamente seus estados e parâmetros para que possa se manter atualizado.
- autoproteção - o sistema deve detectar, identificar e proteger o sistema de ameaças externas que podem comprometer sua segurança e integridade.
- autorrecuperação - o sistema deve ser capaz de descobrir falhas ou possíveis falhas no sistema e recuperar um estado anterior, seguro e correto. Ele possui a capacidade de descobrir, diagnosticar e reagir a distúrbios ou falhas. Ele pode também prever problemas potenciais e tomar decisões para prevenir uma falha.
- conhecimento do ambiente - ter entendimento do ambiente, saber como ele está integrado e automaticamente descobrir outros dispositivos e recursos é importante para a autonomia.
- desenvolvimento colaborativo - os sistemas autônomos devem manter sua implementação em padrão de código aberto, para que as eventuais modificações sejam incluídas o mais rápido possível.
- prever a otimização dos recursos - um sistema autônomo deve prever e proativamente agir na otimização da utilização dos recursos. O sistema deve estar preparado para possíveis mudanças.

Kephart e Chess [72] sustentam a analogia da computação autônoma com o sistema nervoso autônomo humano e a hierarquia social humana. Começando desde as máquinas comparadas as moléculas até a sociedade comparada ao gerenciamento dos sistemas autônomos entre si. Também pesquisadores da IBM, os autores descrevem quatro aspectos ou fundamentos do autogerenciamento: **autoconfiguração (*self-configuring*), autootimização (*self-optimization*), autorrecuperação (*self-healing*), autoproteção (*self-protection*)**. Esses quatro *self-** são bem conhecidos nos dias de hoje e é sugerido analisá-los e estudá-los separadamente, apesar de poder ser necessária a integração entre eles.

Kephart e Chess descrevem os sistemas autônomos como sendo elementos autônomos: sistemas individuais que contêm recursos e entrega de serviços para pessoas usuárias ou outros elementos autônomos. Eles são capazes de se autogerenciar internamente e gerenciar a interface com os usuários ou outros elementos autônomos de acordo com políticas estabelecidas. Segundo os autores, a estrutura de um elemento autônomo é mostrada na Figura 2.1.

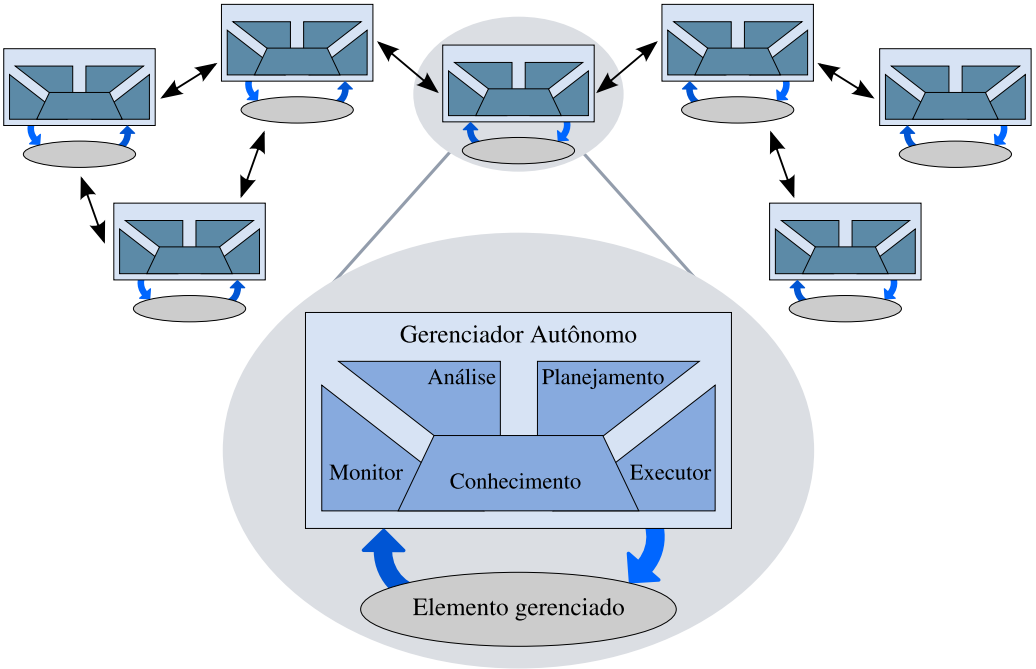


Figura 2.1: Elemento autônomo.

O elemento gerenciado é equivalente a recursos de *hardware* de sistemas comuns como CPU, dispositivos de armazenamento ou impressora, e a recursos de *software* como base de dados, serviço de diretório ou sistemas de legado. O gerenciador autônomo é o responsável por monitorar, analisar, planejar ações e executá-las através dos conhecimentos obtidos. Isto é feito automaticamente em ciclos, deixando o usuário livre deste trabalho. Em mais alto nível, os elementos autônomos lidam entre si de acordo com políticas estabelecidas, havendo um autogerenciamento do sistema completo.

Ganek e Corbi [50] tem um posicionamento semelhante aos outros artigos de autores da IBM sobre computação autônoma. O artigo apresenta uma visão centrada em TI e levanta aspectos de autogerenciamento em indústias e empresas.

Kephart, no artigo em [71] publicado em 2005, disserta sobre o desafio de criar sistemas autônomos e que se intercomuniquem cooperativamente. Separa o espaço de pesquisa desta área em três partes:

- Elementos autônomos: estuda a base do sistema autônomo, como já explicado.
- Sistemas autônomos: estuda a interação entre os elementos autônomos para alcançar algum objetivo a nível de sistema e como quantificar o grau de autogerenciamento nos sistemas.
- Interações computador-humano: estabelecimento de políticas de autogerenciamento

e interface dos sistemas com humanos.

Mais recentemente, Dobson *et al.* [32] também traçam um pequeno histórico da computação autônoma e indaga existir uma deficiência de entendimento de aspectos mais amplos de engenharia de *softwares* no desenvolvimento dos sistemas por parte dos pesquisadores. A solução seria explorar os requisitos, especificações e verificações de tais sistemas.

2.2.1 Auto-tuning

O termo auto-*tuning* (*self-tuning*) é usado para classificar *softwares* ou programas de computadores que são capazes de empregar técnicas empíricas para avaliar um conjunto de alternativas de mapeamento de dados para uma determinada arquitetura e selecionar aquele que obtém o melhor desempenho dentre o conjunto de mapeamentos. O ajuste ou *tuning* pode ser feito de três maneiras: através de técnicas automáticas de otimização durante a compilação [126, 70]; através do uso de pacotes de *software* de auto-*tuning* que servem para testar diferentes implementações de uma determinada rotina crítica e selecionar aquela que obteve melhor desempenho. Exemplos destes pacotes são o ATLAS [124] e PhiPAC [12] para álgebra linear e FTTW [49] e SPIRAL [129] para processamento de sinal; e através de buscas empíricas automáticas sobre um conjunto de parâmetros definidos pelo programador da aplicação [87].

Desde a década de 1970, esforços vem sendo introduzidos para adaptar um código de programa a uma determinada arquitetura, de modo a otimizar sua execução [128, 126, 70]. O objetivo era moldar empiricamente o programa para uma arquitetura específica, já que cada infraestrutura possui características diferentes como memória (por exemplo, dimensões e hierarquias distintas) e processador (por exemplo, conjunto de instruções, tamanho de registradores e velocidade de processamento distintos). Historicamente, a técnica de auto-*tuning* representa um passo importante na evolução da computação autônoma focada na propriedade de autoconfiguração.

2.3 Sistemas Gerenciadores

Sistema gerenciadores representam uma camada de gerenciamento entre a aplicação e a infraestrutura, e esta camada intermediária também é conhecida como *middleware*. O *middleware* é responsável pelo gerenciamento do sistema composto pelas aplicações e in-

fraestruturas, de modo a tirar a complexidade do escopo do desenvolvedor. A Figura 2.2 representa os níveis de camada de sistemas gerenciadores em geral. No nível mais baixo, representado pela camada de *e*-infraestrutura, encontram-se todos os elementos computacionais descritos na Seção 2.1 deste capítulo, além de dispositivos de armazenamento, redes de diferentes tipos e estruturas e instrumentos científicos que fornecem dados (*e.g.*, satélite, telescópios e *lasers*). A parte de *software* dos sistemas operacionais também está incluída nesta camada. O nível intermediário representa a camada do sistema gerenciador ou *middleware*. Esta camada é dividida em duas partes: *middleware* básico e *middleware* de serviços. O *middleware* básico é capaz de oferecer serviços fundamentais como alocação de recursos, serviços de diretórios, transferência de arquivos, bibliotecas de funções e segurança básica. O *middleware* de serviços provê um alto grau de abstração, incluindo serviços como escalonamento dinâmico ou em tempo de submissão de *jobs* ou tarefas, balanceamento de carga, tolerância a falhas, monitoramento, configuração de sistema manual ou automática e proteção de dados. O nível mais acima é representado pela camada das aplicações que irão explorar os recursos da *e*-infraestrutura disponíveis através dos sistemas gerenciadores.

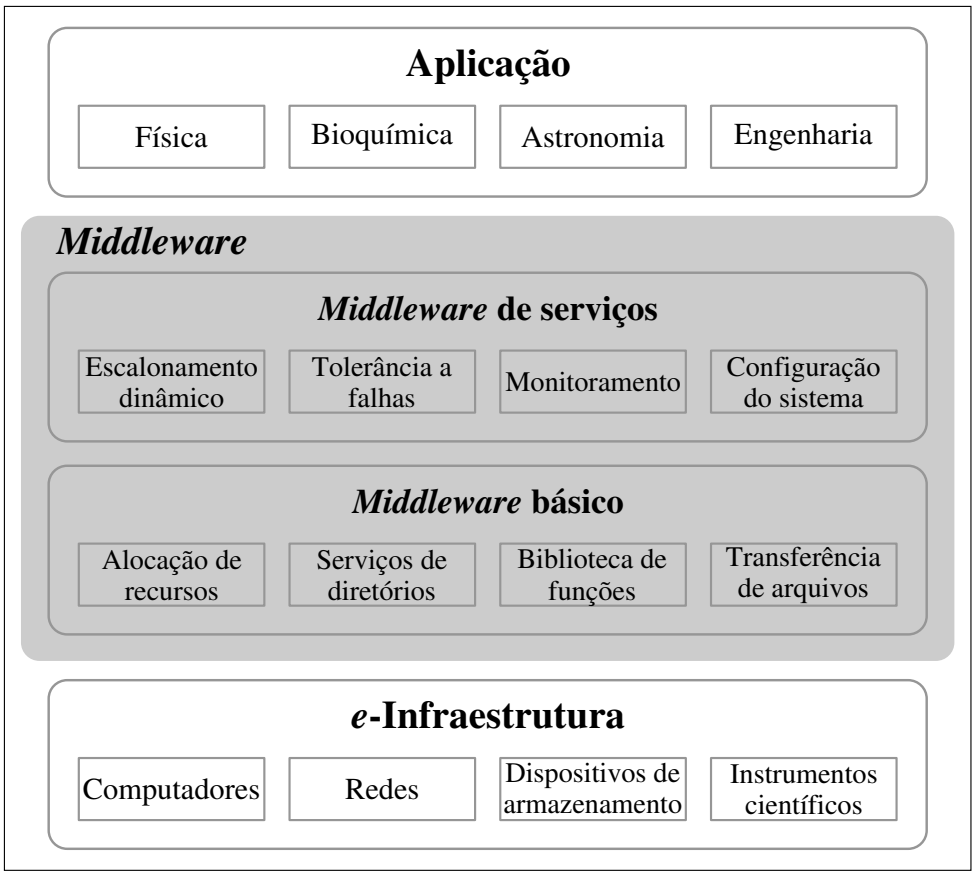


Figura 2.2: Níveis de camada de sistemas gerenciadores ou *middlewares*.

Atualmente, considera-se que existam três filosofias para os sistemas de gerenciamento de acordo com seus objetivos: os sistemas de gerenciamento de recursos (*Resource Management Systems*, ou RMS), os sistemas de gerenciamento de usuário (*User Management Systems*, ou UMS) e os sistemas de gerenciamento de aplicação (*Application Management Systems*, ou AMS).

Um RMS [74] é tipicamente um sistema centralizado projetado para associar *jobs* às máquinas considerando os requisitos de recursos. O DAGMan HTCondor (*Directed Acyclic Graph Manager* do HTCondor) [112] e o Taverna [90], citados na taxonomia de Yu e Buyya [132], são considerados RMSs, assim com o Glite [76]. Um UMS [25] tem o foco em permitir o gerenciamento das aplicações de cada usuário separadamente. O gerenciamento dos recursos (descoberta e alocação) é feito por um RMS associado. A visão é centrada no usuário e existe um gerenciador para cada um deles. O GridBus [131] e Nimrod/G [20] são exemplos de UMS. Um AMS [16] tem uma visão centrada na aplicação, na qual as suas características internas e requisitos são considerados. EasyGrid AMS [86], GrADS [11] e COMPSs [114] são exemplos de AMS.

Em relação à autonomia, muitos sistemas designados para gerenciar a execução de aplicações em ambientes computacionais distribuídos já vêm incorporando mecanismos de autogerência. No entanto, a maioria deles é limitado ou incompleto, isto é, fica restrito a poucos cenários de execução (geralmente apenas um) ou não possui todas as características autônomas.

Nesta seção são apresentados alguns sistemas de gerenciamento que já possuem algum tipo de autonomia ou são projetados para servir como base em sistemas a se tornarem autônomos. O foco está em sistemas projetados para executarem, em ambientes distribuídos, aplicações científicas, isto é, aquelas que resolvem problemas de áreas como física, engenharia, biologia, astronomia e outras. O foco são aplicações científicas porque são o conjunto de aplicações tratado neste trabalho.

2.3.1 Nimrod/G

Nimrod [4, 3] foi inicialmente desenvolvido para gerenciar sistemas de computação distribuídos em que diversas aplicações executam. Tais aplicações são do tipo *parameter sweep*, isto é, que apresentam o mesmo tipo de execução independente apenas trocando-se os parâmetros de entrada. Ele provê uma linguagem de modelagem declarativa e parametrizada para expressar aplicações paramétricas. Os desenvolvedores podem criar um plano de parâmetros para tais aplicações e usar o sistema de gerenciamento Nimrod de

forma centralizada através de um *broker* para submeter, executar e coletar os resultados de múltiplos computadores. Nimrod tem sido usado para executar aplicações que vão de bio-informática a pesquisa operacional para simular processos de negócio.

Nimrod/G [20] é a versão que usa o Globus [45], um *middleware* de serviços para descoberta de recursos dinâmica e gerência das tarefas da aplicação em uma grade computacional, que necessita de suporte para acessar diferentes domínios administrativos.

Pode-se dizer que este sistema é limitado em relação aos tipos de aplicações a serem executadas sobre ele e é pouco autônomo, já que o usuário é quem dita as políticas de gerenciamento. Ele apenas apresenta capacidade de autorrecuperação e auto-otimização. Diante de falhas, ele as reconhece e recupera através de técnicas de *checkpoint* como a gravação de estado da execução pelo *broker* em dispositivo de armazenamento. O escalonador de tarefas Nimrod/G busca otimizar o custo da computação.

2.3.2 Gridbus

Gridbus [21] é um gerenciador de usuários direcionado para execução de aplicações que possuam computação intensiva de dados. O acesso remoto e a otimização de transferência de dados são características importantes do gerenciador [119]. Além deste UMS administrar as aplicações do usuário de acordo com propriedades como orçamento e *deadline*, a proximidade dos dados também é considerada nas decisões automáticas.

Cada usuário possui um *broker* associado pelo qual ele pode submeter suas aplicações. O usuário deve informar manualmente requisitos como *deadline* e orçamento; Seu escalonador recebe um conjunto de nós e processos das aplicações do usuário e suas configurações e combina os requisitos dos processos com os serviços disponíveis pelos recursos. Se existe a necessidade da obtenção de dados remotos, o escalonador interage com o serviço de monitoramento do sistema para obter informações sobre a capacidade dos canais de comunicação entre os repositórios de dados e tenta estabelecer um escalonamento que otimize a transferência. No caso de falhas no sistema, existe um mecanismo de detecção e recuperação via reexecução das tarefas. Há também uma extensão capaz de gerar *checkpoints* para recuperar parte das tarefas e reiniciar o cálculo a partir de um ponto antes da falha [107].

O gerenciador Gridbus é parcialmente autônomo em relação à autorrecuperação por parte do mecanismo de tolerância a falhas e à auto-otimização por conta do escalonamento em tempo de submissão de tarefas que considera a localização dos dados. A parte

de autoconfiguração é precária, já que o usuário deve indicar manualmente a maioria das configurações e quase nenhum parâmetro do sistema é alterado dinamicamente e automaticamente. Apesar de ter mecanismos de autenticação, este sistema não sugere possuir a característica de autoproteção.

2.3.3 HTCondor

HTCondor[2] [112, 48] é um sistema gerenciador de recursos para tarefas de computação intensiva. Neste caso, uma tarefa é uma subdivisão da aplicação. Assim como a maioria dos outros gerenciadores, provê um mecanismo de enfileiramento de tarefas, políticas de escalonamento, esquemas de prioridades, monitoramento de recursos e gerenciamento de recursos. Os usuários submetem suas tarefas ao Condor. Elas são colocadas em uma fila e escolhidas pelo seu meta-escalonador DAGMan (*Directed Acyclic Graph Manager*) para executar em uma determinada máquina de acordo com políticas previamente estabelecidas pelo usuário. Durante a execução, elas são monitoradas e, no fim, o HTCondor informa ao usuário a terminação e os resultados.

Algumas características do HTCondor são:

Submissão distribuída - não existe uma única máquina centralizada para submissão de tarefas. Elas podem ser submetidas de diferentes máquinas e cada máquina contém sua própria fila de tarefas.

Prioridade de tarefas - o usuário pode priorizar tarefas para controlar a execução delas.

Prioridade de usuários - o administrador pode associar prioridades a usuários usando diferentes políticas.

Aplicações GAD (Grafo Acíclico e Direcionado) - suporta aplicações em que as tarefas possuem dependência de outras para serem executadas. DAGMan é responsável pelo escalonamento.

Suporte para múltiplos modelos de tarefas - permite o uso de tarefas sequenciais e paralelas, como MPI.

ClassAds - é um mecanismo que permite associar tarefas às máquinas de acordo com os atributos delas. Uma máquina possui atributos ClassAd como memória disponível,

[2] O gerenciador HTCondor era conhecido como 'Condor' deste 1988 até 2012, quando o nome foi alterado devido a uma ação judicial.

tipo de CPU e velocidade, tamanho da memória virtual, média de carga corrente, e outros. Além disso, também é configurado em que condições a máquina está disposta a executar uma tarefa do Condor e que tipo de tarefa ele prefere. Por sua vez, as tarefas possuem seus atributos ClassAd como os requisitos da arquitetura da máquina, uso de disco, sistema operacional, entre outros. O *matchmaking* será responsável por casar os atributos entre tarefas (ainda em fila) e máquinas e associá-las entre si.

Checkpoint e migração de tarefas - Condor possui a execução de *checkpoints* periódicos que permitem a migração de tarefas de uma máquina para outra, recuperando o estado referente ao último *checkpoint* gravado.

Checkpoint e recuperação de tarefas - os *checkpoints* periódicos também permitem que as tarefas sejam recuperadas em caso de falhas no sistema como o desligamento repentino de uma máquina.

Suspensão e retomada de tarefas - baseado em regras de política, Condor pode pedir ao sistema operacional para suspender e retomar uma tarefa quando necessário.

Autenticação e autorização - administradores podem requerer dos usuários autenticação para autorizar o uso do sistema.

Plataformas heterogêneas - tarefas HTCondor podem ser executadas em plataformas Unix e Windows concomitantemente.

Computação em grades - através do Globus, Condor pode interagir com os recursos em uma grade.

HTCondor apresenta alguns aspectos autônomos como o mecanismo de ClassAd, que permite selecionar máquinas às tarefas, e o mecanismo de tolerância a falhas através de *checkpoint*. No entanto, deixa a desejar em relação à autoconfiguração. A maioria das configurações é feita manualmente [23]. Além disso, uma vez que a tarefa esteja executando em uma máquina previamente escolhida, ela permanecerá até o fim da execução nesta máquina independente do estado em que se encontra. Apenas em caso de falha ocorreria uma migração. Neste caso, a auto-otimização do sistema é limitada apenas em relação a manipulação da fila de execução do HTCondor.

2.3.4 GLite

GLite [76, 81] é uma pilha de *middlewares* que combina componentes desenvolvidos em vários projetos relacionados, como HTCondor e Globus. Projetado pela EGEE[3], ele disponibiliza ao usuário serviços de alto nível para escalonar e executar tarefas, acessando e movendo dados, e obtendo informações sobre o estado dos recursos e tarefas. É considerado também um RMS. Os seus serviços de alto nível podem ser agrupados em cinco grupos:

- Serviços de acesso - fornece *interfaces* para se obter acesso aos recursos da grade computacional através de um portal.
- Serviço de segurança - engloba os serviços de autenticação, autorização e auditoria, que possibilitam identificar entidades (usuários, sistemas e serviços), permitir ou negar acesso a serviços e recursos, e prover informação de segurança.
- Serviço de informação e monitoramento - provê mecanismos de publicar informação e usá-la para propósitos de monitoramento. O serviço de monitoramento de tarefas permite obter informações sobre a utilização dos recursos pelas tarefas da aplicação.
- Serviço de Gerenciamento de tarefa - agrupa 4 serviços: elemento de computação, sistema gerenciador de carga, gerenciador de pacotes e proveniência de tarefas. O serviço de elemento de computação provê informação sobre os recursos e oferece uma *interface* comum para submeter e gerenciar tarefa nos recursos. O sistema gerenciador de carga fornece um escalonador de tarefas nos elementos computacionais disponíveis de acordo com preferências do usuário e várias políticas. Em caso de falha de um recurso, a tarefa que o utilizava é reexecutada. O gerenciador de pacotes colabora no desenvolvimento do *software* da aplicação. O serviço de proveniência de tarefas oferece informações persistentes sobre as tarefas executadas nos recursos para posteriores inspeções, mineração de dados e possíveis re-execuções.
- Serviço de dado - agrega 4 serviços: elemento de armazenamento, catálogo de metadado, catálogo de arquivo e réplica e gerenciamento de dados. O serviço de elemento de armazenamento provê a virtualização de recursos de armazenamento. O catálogo de metadado mantém um histórico de informações de relevante metadados

[3]EGEE - *Enabling Grids for E-sciencE* - foi o projeto desenvolvedor da infraestrutura para o CERN. O projeto terminou em 2010 e, atualmente, chama-se EGI - *European Grid Infrastructure*.

como *checksum* e tamanhos de arquivos. O catálogo de arquivo mantém informações sobre a localização dos dados. O serviço de gerenciamento de dados controla a transferência de dados entre os elementos de armazenamento.

O gLite introduz aspectos autônomos de autoconhecimento através do constante monitoramento da execução da aplicação no sistema. A autoproteção é atendida pelo seu robusto serviço de segurança. Da mesma forma que no Condor, o escalonamento em tempo de submissão das tarefas da aplicação é feito apenas enquanto ela está na fila de execução, o que torna a auto-otimização limitada especialmente no caso de tarefas de granularidade grossa. A autorrecuperação é feita através detecção da falha e da reexecução da tarefa. Basicamente não há autoconfiguração, já que o usuário deve fornecer todas as informações da aplicação e requisitos do sistema.

2.3.5 GrADS

GrADS [11] é um exemplo de sistema gerenciador de aplicações. Diferentemente dos já citados gerenciadores de recursos, seu objetivo é maximizar a eficiência de execução das aplicações e não dos recursos. Este *middleware* fornece ferramentas e bibliotecas que permitem ao usuário criar aplicações que possam ser encapsuladas como programas objetos configuráveis (COPs). Um COP é constituído do código da aplicação, de um mapeador de tarefas e de um modelo que estima o desempenho da aplicação nos recursos selecionados. Assim, seu objetivo é minimizar o tempo de execução da aplicação, o *makespan*. Seu escalonamento é centralizado, já que ele considera todos os recursos e tarefas, fazendo um mapeamento da estimativa de desempenho delas nos recursos e aplicando heurísticas para minimizar o tempo de execução da aplicação.

O *middleware* apresenta mecanismos de tolerância a falhas (por *checkpoints*) e monitoramento dos recursos. GrADS pode ser considerado um sistema gerenciador parcialmente autônomo através de sua autoconfiguração e auto-otimização. Além disso, apresenta projetos específicos para aplicações, como é o caso do GrADSAT [24] para resolver instâncias o problema de satisfabilidade.

2.3.6 COMPSs

COMP Superscalar [114] (COMPSs) é um *framework* de programação cujo principal objetivo é facilitar o desenvolvimento de aplicações para ambientes distribuídos. Ele é

baseado no GCM (*Grid Component Model*) [96] , um modelo de componentes distribuídos que usa a especificação Fractal [19][4] como referência.

COMPSs apresenta uma hierarquia de componentes de execução formada por:

- Analisador de Tarefas: recebe tarefas da aplicação e detecta suas precedências, gerando um grafo de dependência. Uma tarefa COMPSs é um método descrito no código da aplicação. Quando uma tarefa for selecionada para executar, este componente analisará o grafo de dependências e, se todas as dependências estiverem resolvidas, ele a enviará para o *Escalonador de Tarefas*.

- Escalonador de Tarefas: este componente decidirá onde a tarefa irá executar no ambiente. A decisão é feita de acordo com certos algoritmos de escalonamento que verificam quais recursos estão disponíveis e suas capacidades, um conjunto de restrições da tarefa definidas pelo usuário e a localização do dado necessário para a tarefa executar.

- Gerenciador de *Jobs*: é responsável pela submissão de *jobs* e monitoramento. Este componente recebe uma tarefa já escalonada pelo *Escalonador de Tarefas* e encarrega as necessárias transferências de arquivo para o *Gerenciador de Arquivos*. Quando a transferência torna-se completa para a tarefa, ele a transforma em um *job* para que seja submetido a execução no ambiente, e, então, controla uma finalização adequada do *job*.

- Gerenciador de Arquivos: este componente cuida de todas as operações onde os arquivos estão envolvidas. Ele é composto por dois componentes: o *Provedor de Informação de Arquivos* e o *Gerenciador de Transferência de Arquivos*. O primeiro reúne todas as informações relacionadas aos arquivos como que tipo de acessos a arquivos tem sido feito, quais as versões de cada arquivo existem e onde eles estão localizados. Este último é o componente que realmente transfere os arquivos de uma máquina para outra e também informa ao *Provedor de Informações de Arquivo* sobre a nova localização dos arquivos.

COMPSs suporta a linguagem Java, necessita de uma biblioteca de *middleware* para grades em Java chamada ProActive [69] e faz uso do JavaGAT - *Java Grid Application Toolkit* [118], um conjunto de ferramentas para grades computacionais que fornece acesso a

[4]Fractal é um modelo de componente que se destina a implementar, implantar e gerenciar (isto é, monitorar, controlar e dinamicamente configurar) sistemas de *software* complexos.

serviços, portal e gerenciamento de dados. O modelo de programação sugere não modificar o código sequencial, apesar de ter disponível APIs que iniciam e terminam a execução de processos e acessam arquivos. O usuário deve apenas implementar uma *interface* de Java para indicar ao COMPSs quais métodos irão executar no ambiente de grades. Para identificar as máquinas a serem usadas e suas especificações, um arquivo em XML deve ser configurado.

A autonomia existe por conta dos componentes de gerenciamento descritos. No entanto, o nível de autonomia é baixo, pois é possível destacar apenas a auto-otimização, através dos componentes *Escalonador de Tarefas* que decide que tarefa do grafo de dependências executar e qual o melhor local para iniciá-la de acordo com a configuração atual do sistema. Porém, uma vez que uma tarefa é escalonada, a decisão não pode ser alterada. A autoconfiguração da aplicação é debilitada já que o usuário deve especificar um arquivo de configuração como entrada para a execução.

2.3.7 EasyGrid AMS

O EasyGrid AMS [16] é um sistema gerenciador de aplicações (AMS) que funciona como um meio entre a aplicação e o sistema computacional, de forma a facilitar o uso dos recursos. Foi desenvolvido para executar aplicações MPI [83] em grades computacionais, mas em seu projeto não há impedimento para a utilização de outros ambientes. Ele pode fazer o uso do Globus [56] para utilizar algumas ferramentas como autenticação.

O *middleware* EasyGrid AMS possui um gerenciamento hierárquico, tendo um gerenciador global, gerenciadores de *site* e gerenciadores de máquina. O gerenciador global é o nível mais alto e tem o objetivo de gerenciar os *sites* no ambiente. *Sites* são conjuntos de máquinas que geralmente separam o ambiente de acordo com a localidade geográfica. Gerenciadores de *sites* gerenciam as máquinas dentro de seu conjunto. Gerenciadores de máquinas gerenciam os processos das aplicações nas máquinas.

A filosofia EasyGrid AMS [16, 92] indica que as aplicações devem ser gerenciadas por um sistema simples enquanto a aplicação é gerenciada pelo seu próprio *middleware* autônomo. Como características autônomas, o EasyGrid AMS possui um escalonador de tarefas estático e dinâmico [15, 85] que fornecem auto-otimização às tarefas da aplicação; um mecanismo tolerante a falhas por reexecução de tarefas e *checkpoint* no nível do gerenciador global que é capaz de detectar e recuperar as falhas da aplicação (autorrecuperação) [109]; e autoproteção através da ferramenta de autenticação e troca segura de dados pelo Globus. Trabalhos da literatura que utilizam o EasyGrid AMS apresentam

bons resultados mas não tratam aplicações reais [85, 86, 109, 91].

Nesta tese, propõe-se o uso de um modelo de particionamento que é apoiado pelo *middleware* EasyGrid AMS. O seu escalonamento dinâmico e método de tolerância a falhas são igualmente aproveitados pelas aplicações desde que elas estejam sob o modelo de particionamento sugerido que será descrito no Capítulo 3. Ainda, a autoconfiguração é a característica autônoma do *middleware* mais trabalhada já que deve ser implementada de acordo com uma aplicação específica.

2.3.8 Quadro Comparativo

A Tabela 2.1 apresenta um quadro comparativo que resume as características autônomas e a orientação dos gerenciadores mencionados nesta seção.

Tabela 2.1: Resumo das características autônomas dos sistemas gerenciadores.

Gerenciador	Orientação	Auto-configuração	Auto-otimização	Autor-recuperação	Auto-proteção
Nimrod/G	UMS	Não	Sim (tempo de submissão)	Sim (*checkpoint*)	Sim (Globus)
Gridbus	UMS	Não	Sim (tempo de submissão)	Sim (reexecução e *checkpoint*)	Não
HTCondor	RMS	Não	Sim (tempo de submissão)	Sim (*checkpoint*)	Sim (Globus)
Glite	RMS	Não	Sim (tempo de submissão)	Sim (reexecução)	Sim
GrADS	AMS	Não	Sim (tempo de submissão)	Sim (*checkpoint*)	Não
COMPSs	AMS	Não	Sim (tempo de submissão)	Não	Não
EasyGrid AMS	AMS	Sim	Sim (dinâmico)	Sim (reexecução e *checkpoint*)	Sim (Globus)

2.4 Resumo

Este capítulo foi responsável por descrever propriedades associadas ao tema de aplicações autônomas nas *e*-infraestruturas. A primeira seção apresentou o cenário de infraestrutura atual, reportando as complexidades dos componentes que compõem e dos próprios ambientes distribuídos recentes e, em perspectiva, futuros. A *e*-infraestrutura é dinâmica, heterogênea, compartilhada, suscetível a falhas.

A Seção 2.2 descreveu os conceitos e evolução da computação autônoma, assim como o processo de autoconfiguração pelo auto-*tuning*. A computação autônoma aparece como potencial solução para o problema de gerenciamento da execução de aplicações em sistemas distribuídos. A introdução de componentes autônomos não só ajudam a manter o sistema, mas também podem otimizar a execução. Já a última seção relacionou um conjunto de sistemas gerenciadores que existem na literatura e já fazem uso de algumas propriedades autônomas.

Capítulo 3

Projeto de Aplicações da e-Ciência Autônomas: uma Proposta

Este capítulo tem o objetivo de apresentar a proposta desta tese. Inicialmente, a motivação é relatada através da descrição do problema de gerenciar aplicações científicas em ambientes distribuídos, das vantagens de utilizar um *Sistema Gerenciador de Aplicações* (AMS) e da escolha pelo EasyGrid AMS como ferramenta base. Uma vez que o EasyGrid AMS é um *middleware* capaz de embutir na aplicação as propriedades autônomas de auto-otimização e autorrecuperação, viu-se a necessidade de atribuir também à aplicação a propriedade de autoconfiguração integrada. Esta necessidade vem do fato dos ambientes computacionais distribuídos atuais estarem mudando constantemente e cada vez mais largos em escala, heterogêneos, compartilhados e dinâmicos. Por esta razão, a aplicação precisa ter a capacidade de se ajustar ao ambiente. A metodologia proposta na Seção 3.3 é baseada em uma já existente e bem estabelecida, descrita na Seção 3.2, que guia o desenvolvedor na paralelização e mapeamento de sua aplicação no ambiente de execução escolhido. Aproveitando este conhecimento, a nova proposta pretende adotar esta metodologia dinamicamente durante a execução da aplicação para que, desta forma, ela possa se moldar ao ambiente. A implementação da metodologia proposta tende a ser distinta por cada aplicação, já que são consideradas suas próprias características e as aplicações, de modo geral, são diferentes. Consequentemente, uma classificação de aplicações é realizada de modo que a implementação da metodologia de uma classe possa ser empregada às outras aplicações de mesma classe (Seção 3.2). Assim, as aplicações da *e*-ciência estudadas nesta tese são classificadas (Seção 3.4) e avaliadas de acordo com um critério geral (Seção 3.5).

3.1 Motivação

A pesquisa científica sempre envolveu um grande número de pessoas de diferentes tipos e níveis de conhecimento, trabalhando em diversas funções colaborativas. Com os avanços tecnológicos das últimas décadas, tarefas computacionais e o uso de instrumentos científicos (como satélites, telescópios e *lasers*) têm contribuído com uma imensa quantidade de dados para o processamento de informação. De modo a auxiliar o progresso de seus processos científicos, equipes de diversas organizações ou instituições científicas dependem tipicamente de executar atividades computacionais de larga escala, analisar uma grande massa de dados distribuída e compartilhar resultados obtidos entre si.

A *e-ciência* ou *e-science* [59] é um termo usado para designar a integração da tecnologia computacional ao processo científico. A preparação, a experimentação, a obtenção e armazenamento de dados e a propagação dos resultados compõem eventos do processo científico que utilizam intensivamente a computação de dados em ambientes distribuídos de larga escala. A *e*-ciência é a base em torno da qual os cientistas estão trabalhando para alcançar novas descobertas e avanços em áreas como física, biologia, medicina, física, engenharia e economia, por exemplo. A medida que os cientistas usam a tecnologia para realizar as suas investigações, as infraestruturas computacionais vêm tornando-se cada vez mais poderosas tanto no processamento computacional quanto na capacidade de armazenar os dados correspondentes.

No contexto desse trabalho, *aplicações da e-ciência* representam soluções computacionais para problemas científicos que utilizam a tecnologia disponível. Geralmente elas estão relacionadas ao mundo real, resolvendo ou ajudando a resolver problemas que se aplicam ao dia a dia. Simulações matemáticas de algum fenômeno físico, previsão do tempo, simulações de enrolamentos de proteínas, modelagem da distribuição de espécies de fauna e flora, são alguns exemplos. São problemas complexos que necessitam de recursos computacionais como memória, processamento e armazenamento em larga escala (na escala de tera, peta e exa *bytes* ou FLOPs).

As *e-infraestruturas* ou *cyber-infraestruturas* denominam as tecnologias presentes nas *organizações virtuais* ou VOs (*Virtual Organizations*) científicas. Os grupos de pesquisa compartilham recursos computacionais virtualmente através de grades e nuvens de computadores, que são responsáveis por oferecer os requisitos das aplicações da e-ciência como memória, processamento e armazenamento em larga escala. Através destas infraestruturas, tais aplicações podem ser executadas de modo a atender as necessidades dos cientistas

e disponibilizar resultados às equipes envolvidas na VO de forma satisfatória.

No entanto, tanto grades quanto nuvens de computadores, enquanto são *e*-infraestruturas potencialmente fornecedoras de grande poder computacional, apresentam diversos fatores que dificultam o desenvolvimento de aplicações. Como forma de agregar recursos, tais *e*-infraestruturas possuem diferentes tipos de computadores, com capacidades de memória, processamento e armazenamento distintas. Um componente de uma *e*-infraestrutura pode ser um computador com apenas uma unidade de processamento; pode ser um computador multiprocessado; alguns dos computadores podem conter aceleradores ou coprocessadores; e, de uma forma geral, pode ser outra arquitetura existente nos dias de hoje e futuramente. Além disso, os computadores representam nós em rede conectados com diferentes velocidades de transferência de dados e essa rede pode perder ou ganhar nós ao longo do tempo. Falhas de *software* podem ocorrer devido a alguma situação não prevista pelo programador, já que o sistema é complexo. Falhas de *hardware* implicam na remoção temporária ou permanente de um nó do sistema. Quantidade de memória, processamento e armazenamento disponível a uma aplicação, que além de serem diferentes, variam de acordo com a utilização das máquinas compartilhadas por diversos usuários nos sistemas computacionais. Além disso, a execução das aplicações está sujeita a diferentes políticas de uso de acordo com o domínio administrativo em que um componente do ambiente computacional se encontra.

O problema abordado nesta tese resume-se em responder a questão de como projetar uma aplicação da *e*-ciência (usando uma *e*-infraestrutura) de forma que os requisitos do cientista sejam atendidos. Na grande maioria das vezes, o requisito do cientista é obter uma boa eficiência da execução de sua aplicação, onde a métrica utilizada é o tempo de execução ou qualidade da solução. Em outras palavras, o compromisso entre os resultados obtidos e os recursos empregados deve ser satisfeito. Por um lado, a aplicação da *e*-ciência necessita de uma larga escala de recursos computacionais para obter resultados em um tempo de execução viável ou de boa qualidade. Por outro lado, existe a *e*-infraestrutura complexa disponível que potencialmente contribuirá para fornecer alto poder de processamento e armazenamento. Entretanto, o cientista, geralmente, não está interessado em entender o ambiente de execução por completo onde seu programa será introduzido, já que são muitas variáveis envolvidas. Para reduzir o esforço de desenvolvimento, é importante existir um meio de gerenciamento que faça o intermédio entre estes dois lados: a aplicação científica e a *e*-infraestrutura. Este meio de gerenciamento é também responsável pela integração da autonomia.

Neste capítulo, o projeto de aplicações científicas paralelas e autônomas é tratado considerando um modelo de projeto de aplicações paralelas existente. Neste projeto, um sistema gerenciador de aplicações (AMS) é utilizado para permitir uma composição de um *middleware* autônomo com a aplicação, de forma a fazer com que ela seja autônoma também. O projeto também aborda o particionamento do problema científico tendo em vista que a aplicação possa aproveitar o que um AMS específico tem a oferecer em relação ao autogerenciamento. Usando dinamicamente a metodologia do projeto de aplicações paralelas tradicional e identificando características dos problemas científicos de uma determinada aplicação, é possível implementar o autogerenciamento eficientemente. Ainda, de acordo com a metodologia do projeto de aplicações paralelas tradicional, algumas classes foram selecionadas para aplicar o modelo de projeto autônomo. A ideia é reutilizar o projeto entre aplicações de mesma classe.

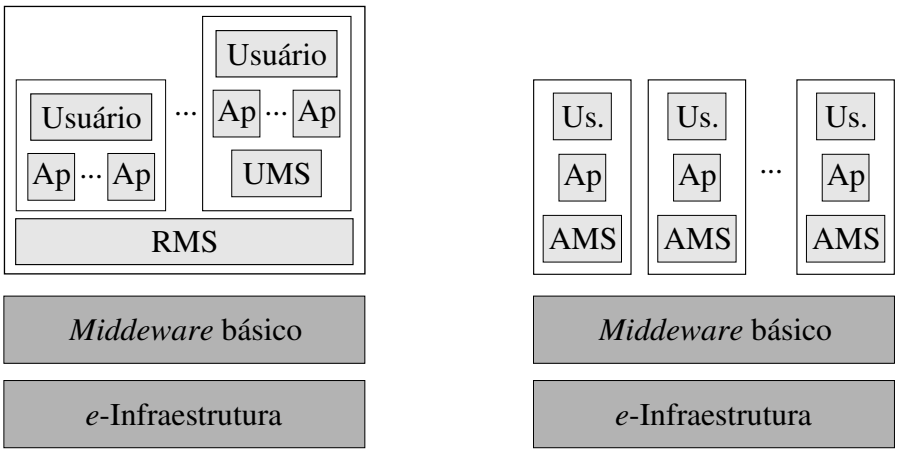
3.1.1 AMS como Gerenciador Autônomo

Um *sistema gerenciador* representa uma camada de gerenciamento entre a camada de aplicação e a camada de infraestrutura. Através de um sistema gerenciador, as aplicações em geral de usuários em geral podem ser administradas e coordenadas de modo a tirar a complexidade de gerenciamento da *e*-infraestrutura do escopo do desenvolvedor. Conforme mostrado na Seção 2.3 do Capítulo 2, existem 3 tipos de sistemas gerenciadores, de acordo com seus objetivos: o *Sistema Gerenciador de Usuários* (UMS), o *Sistema Gerenciador de Recursos* (RMS) e o *Sistema Gerenciador de Aplicações* (AMS). O AMS é aquele que melhor se encaixa no projeto de aplicações autônomas.

Os UMSs têm uma visão direcionada ao usuário ou a um grupo deles (no caso de grupos colaborativos científicos, como as VRE – *Virtual Research Environment* [102]), o que significa que as aplicações de cada usuário são gerenciadas independentemente e cada usuário possui o seu próprio UMS. Tais gerenciadores precisam geralmente que o usuário indique as configurações do sistema que melhor satisfazem seus objetivos. Por esta razão, UMSs não são considerados autoconfiguráveis embora possam ter as características de auto-otimização, autoproteção e autorrecuperação. Estas três últimas propriedades autônomas tendem a utilizar automaticamente apenas informações do sistema (monitoramento constante) enquanto as outras informações são configuradas manualmente pelo usuário. Os UMSs necessitam também de um RMS para fazer a descoberta e alocação de recursos, já que eles visam apenas gerenciar as aplicações de um usuário de acordo com suas especificações em relação aos recursos.

Um RMS é centrado no sistema e tipicamente associam *jobs* às máquinas considerando apenas os requisitos de recursos atribuídos pelas aplicações. Os RMSs constituem a maioria dos sistemas de gerenciamento desenvolvidos para computação em grades, no qual o objetivo é maximizar a utilização do sistema, independentemente dos requisitos e características internas das aplicações. Os RMSs têm uma visão global dos recursos, já que o objetivo é otimizar o seu uso. Do ponto de vista da aplicação, os serviços necessários para o gerenciamento, como o escalonamento, o monitoramento e o tratamento de falhas, são implementados de modo a suprir necessidades básicas de qualquer tipo de aplicação, sem especialização.

Um AMS tem uma visão centrada na aplicação, na qual as suas características internas e requisitos podem ser considerados. Neste caso, recursos do sistema são avaliados para determinar uma aproximação para a melhor execução dos *jobs* da aplicação de acordo com alguma métrica, como o tempo total de execução ou um *deadline*.



(a) RMS/UMS (b) AMS

Figura 3.1: Comparação dos sistemas gerenciadores RMS, UMS e AMS.

A Figura 3.1 compara as camadas de gerenciamento de cada um dos sistemas em relação a organização estrutural. Em todos eles existem a camada de *e*-infraestrutura e a camada de *middleware* básico, conforme a Figura 2.2 da Seção 2.3 do Capítulo 2. A camada de *e*-infraestrutura é formada por todos os recursos computacionais que as aplicações podem usar no sistema. Como descrito na Seção 2.1 do Capítulo 2, as três classes de ambientes: supercomputadores, grades e nuvens de computadores podem compor uma *e*-infraestrutura. Tais classes são potencialmente formadas por computadores multiprocessados, com aceleradores como GPGPU e coprocessadores, com diferentes configurações de memória e estão conectados por diferentes tipos de redes. Ainda podem conter serviços de armazenamento de dados de diferentes tipos como discos e fitas magnéticas, além de

instrumentos científicos responsáveis por fornecer os dados computacionais. A camada de *middleware* básico, comum a todos os gerenciadores, representa funcionalidades básicas como segurança, transferência de arquivos, alocação de recursos e serviços de diretório.

A pilha de camadas de um sistema gerenciador de recursos apresenta um RMS global que administra as aplicações de todos os usuários e é representada na Figura 3.1(a). Ele é responsável por coordenar também as aplicações gerenciadas por um UMS. Tais sistemas gerenciadores convencionais basicamente inspecionam o ambiente, escalonam as tarefas da aplicação, disponibilizam serviços para controle de acesso e tratamento de falhas e outros tipos de serviços como acesso remoto a base de dados. Isto faz com que sejam adicionadas camadas extras capazes de lidar com todos os serviços que podem ser oferecidos pela *e*-infraestrutura. O problema deste tipo de *middleware* é que, inserindo estas camadas de serviço, várias tarefas de gerenciamento genérico são adicionadas. Do ponto de vista do desempenho da aplicação, quanto mais camadas existirem e quanto mais o gerenciamento for genérico pior tende a ser sua eficiência [16]. Isto ocorre porque a quantidade de camadas aumenta a sobrecarga do sistema sobre sua execução, enquanto o gerenciamento genérico faz com que seja introduzido serviços desnecessários à aplicação (aumentando a sobrecarga também).

A Figura 3.1(a) apresenta, ainda, a pilha de camadas de um UMS. Um sistema de gerenciamento de usuários necessita de um gerenciador de recursos RMS e, por esta razão, UMS e RMS estão representados na mesma figura. Existe um bloco do par usuário/UMS englobando suas aplicações para cada usuário. A *e*-infraestrutura é gerenciada por um RMS que fica abaixo da camada UMS. Como o UMS é centrado no usuário, conceitualmente, as configurações das aplicações em geral são especificadas por usuário para que a otimização do sistema atenda suas necessidades. Neste caso, o uso da característica fundamental autônoma de autoconfiguração é prejudicado.

Na Figura 3.1(b), é mostrada uma alternativa de um sistema gerenciador focado nos requisitos da aplicação. Neste caso, existe um AMS específico para cada aplicação. Apenas as camadas de serviço necessárias à aplicação são implementadas e a sobrecarga do sistema tende a ser relativamente menor. As características da aplicação, como uso de processamento, comunicação e acesso a dados em dispositivos de armazenamento, podem ser tratas de modo a otimizar seu desempenho. A própria aplicação teria a capacidade de adquirir autogerenciamento de forma específica para ela.

Na proposta desta tese, o uso de um sistema gerenciador de aplicações (AMS) é considerado para torná-las autônomas executando em ambientes distribuídos de larga escala,

as *e*-infraestruturas. Este tipo de gerenciador é capaz de se manter mais próximo a aplicação e, portanto, conhecer melhor suas características. Desta forma, as propriedades de autoconhecimento, autoconfiguração, auto-otimização, autorrecuperação e autoproteção podem ser implementadas de modo que as características internas da aplicação sejam apreciadas para tirar proveito dos sistemas distribuídos.

3.1.2 EasyGrid AMS como middleware

O AMS escolhido nesta tese foi a ferramenta EasyGrid AMS já descrita no Capítulo 2. Este *middleware* é intrinsecamente um sistema gerenciador de aplicações que possibilita fornecer propriedades autônomas para que a aplicação torne-se autogerenciável. Ele possui escalonadores dinâmicos eficientes capazes de permitir a auto-otimização da aplicação no sistema. Possui ainda um mecanismo de tolerância a falhas simples e funcional e disponibiliza meios de efetuar a autoconfiguração da aplicação.

Para que a aplicação possua as capacidades autônomas que o *middleware* oferece, é sugerido o uso do modelo de particionamento 1Ptask, embora não esteja limitado a isto. Através deste modelo, a aplicação pode ser particionada em uma grande quantidade de tarefas de granularidade (quantidade de trabalho) fina e esta quantidade inclusive tende a ser maior que a quantidade de processadores existentes no ambiente. O objetivo é fazer com que a aplicação libere uma quantidade grande de tarefas que possa ser administrada pelo EasyGrid AMS. Deste modo, o escalonador dinâmico poderá estabelecer um escalonamento proativo e eficiente. No caso de falhas, simplesmente reexecutando as tarefas, é possível ter baixo custo no tratamento de falhas já que elas possuem granularidade fina.

No entanto, para se adaptar melhor ao ambiente, a aplicação necessita de etapas de autoconfiguração. Embora o EasyGrid AMS já apresente as propriedades de auto-otimização e autorrecuperação, a autoconfiguração da aplicação também deve ser integrada ao AMS. Deste modo, mudanças comuns nos ambientes distribuídos atuais poderiam ser tratadas pelas próprias aplicações de acordo com seus interesses.

3.1.2.1 Modelo de Particionamento

O modelo de particionamento de dados do EasyGrid AMS [86] promove o desenvolvimento da paralelização de aplicações em ambientes heterogêneos, compartilhados e dinâmicos como grades computacionais. Ele sugere o uso de um processo por tarefa, chamado modelo 1Ptask, que difere do modelo tradicional, chamado de 1Pproc.

O modelo 1Pproc indica o uso de um processo por processador, fazendo com que cada processo fique alocado a uma CPU (considerando que as máquinas podem ser multiprocessadas). Para realizar alguma migração de processo de uma CPU para outro, geralmente é necessário o uso de mecanismos de *checkpoints*. O modelo 1Ptask propõe um processo por tarefa, onde uma tarefa representa uma unidade de carga definida pelo desenvolvedor. Sendo assim, poderão existir (e geralmente é isto que acontece) mais processos do que máquinas. Através disso, o *middleware* garante mais flexibilidade com as tarefas da aplicação para aplicar um escalonamento apropriado para cada tarefa. Pelo uso deste modelo, também é possível usufruir do mecanismo de tolerância a falhas de baixo custo do EasyGrid AMS, que apenas realiza a reexecução da tarefa com falha, caso houver.

A Figura 3.2(a) representa um exemplo de uso do modelo 1Pproc. Cada processo fica associado a um processador e as setas entre processos indica que há comunicação ou alguma dependência de dados entre eles. Em comparação com a Figura 3.2(a), a Figura 3.2(b) contém tarefas ao invés de processos e elas compõem os processos da Figura 3.2(a) repeitando as dependências.

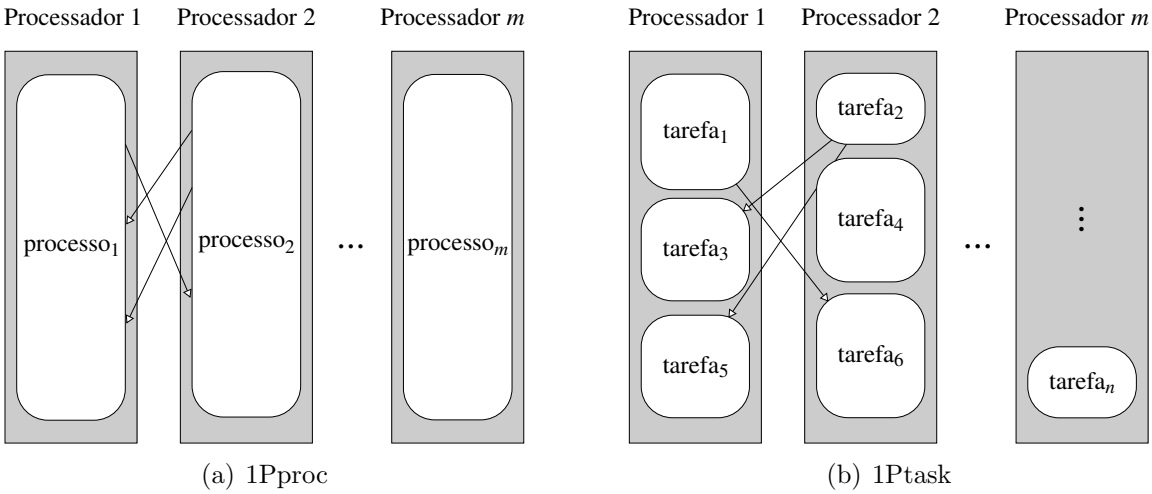


Figura 3.2: Modelos de particionamento do EasyGrid AMS.

3.1.2.2 Escalonamento dinâmico

O *middleware* EasyGrid AMS apresenta uma estrutura de gerenciamento distribuído de 3 níveis. No último nível, encontram-se os *host managers* (HM) ou gerenciadores de máquinas que são responsáveis por gerenciar os processos das aplicações nas próprias máquinas. O segundo nível é composto pelos *site managers* (SM) ou gerenciadores de *sites*. Eles coordenam grupos de HM cooperando com outros SM. *Global manager* (GM) ou gerenciador global está no topo da hierarquia e corresponde ao processo inicial responsável

por criar e coordenar os gerenciadores de níveis imediatamente mais abaixo, os SM. Estes três tipos de gerenciadores apresentam um método de monitoramento considerando a hierarquia do *middleware* e o HM realiza o monitoramento diretamente das máquinas e das tarefas da aplicação.

O escalonador dinâmico está diretamente associado aos processos de gerenciamento da hierarquia do EasyGrid AMS, como mostra a Figura 3.3. No topo da hierarquia, o *global dynamic scheduler* (GDS), ou escalonador dinâmico global é executado no GM; cada SM contém um *site dynamic scheduler* (SDS) ou escalonador dinâmico de *sites*; e cada HM tem um *host dynamic scheduler* (HDS) ou escalonador dinâmico de máquinas.

A alocação (ou mapeamento) dos recursos para as tarefas da aplicação (prontas para executar) é feito pelo HDS e ela não é realiza no momento em que a tarefa se torna pronta para executar, isto é, possui todas as dependências resolvidas. Ao invés, o HDS atribui um grupo[1] de tarefas à máquina associada e deixa o restante em um fila de execução. O desempenho estimado do restante das tarefas é periodicamente calculado de modo a prever possibilidades de reescalonamento das tarefas nas filas de cada HDS. Os SDS são responsáveis por checar o desbalanceamento de carga entre as máquinas e realizar a migração de tarefas. Por sua vez, o GDS checa por desbalanceamento entre os *sites* e ativa eventos de reescalonamento se for necessário. O desempenho estimado usado como métrica por cada escalonador é calculado de acordo com o objetivo da aplicação em relação aos recursos utilizados, como uso intensivo de CPU, memória ou I/O (entrada ou saída).

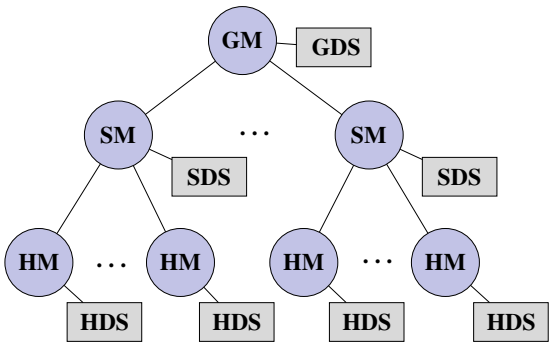


Figura 3.3: Hierarquia do escalonamento dinâmico do EasyGrid AMS.

[1]O HDS atribui um grupo de tarefas/processos à máquina geralmente de acordo com o número de núcleos. Por exemplo, se a máquina possui 4 núcleos no total, 4 processos são executados concorrentemente. O sistema operacional é responsável por escalonar localmente estes processos na máquina.

3.1.2.3 Tolerância a falhas

O mecanismo de tolerância a falhas do EasyGrid AMS serve para detectar e recuperar falhas no sistema tanto de *hardware* (*e.g.*, desligamento de máquinas, perda de conexão) quanto de *software* (*e.g.*, sistema operacional decide eliminar algum processo). No caso de falhas de *software*, geralmente, os processos podem ser recuperados e retornar a execução na mesma máquina. No caso de falha em *hardware*, geralmente, os processos necessitam ser migrados e alocados a outra máquina, uma vez que a máquina falha encontra-se indefinidamente inutilizada.

A detecção de falhas é integrada ao monitoramento e aos escalonadores dinâmicos de cada nível hierárquico do EasyGrid AMS. A etapa de recuperação apresenta estratégias diferenciadas de acordo com o tipo de processo: de aplicação ou gerenciadores. Se for um processo da aplicação, a recuperação é realizada pela recriação dos processos, considerando o modelo de particionamento 1Ptask que apresenta tarefas de granularidade fina e, portanto, recriar o processo tende a ser melhor que implementar os comuns e custosos mecanismos de *checkpoint* [109]. Se for um processo gerenciador, ele poderá ser recriado, caso ainda tenha conexão com a máquina onde executou, ou poderá ser eliminado do sistema. Neste último caso, a configuração da execução das tarefas da aplicação foi previamente guardada pelo gerenciador de camada acima e, portanto, as tarefas pertencentes ao gerenciador falho poderão ser reescalonadas para outras máquinas. As tarefas que estariam executando no momento da falha de seu gerenciador devem ser recriadas. Apenas o gerenciador global faz *checkpoint* em disco.

3.2 Projeto de Aplicações Paralelas

Paralelizar as aplicações da *e*-ciência tende a não ser uma tarefa simples. Os problemas que envolvem as aplicações são complexos e geralmente relacionam um grande volume de dados e, tanto o algoritmo quanto os dados devem ser levados em consideração ao desenvolver a solução paralela. Além disso, o uso de *e*-infraestruturas requer que esta paralelização seja suficiente para explorar o poder computacional disponível. O projeto de paralelização dos algoritmos e dados destas aplicações é imprescindível para que haja um aproveitamento dos recursos computacionais. Este seria um primeiro passo do desenvolvimento de aplicações da *e*-ciência autônomas para sistemas computacionais paralelos e distribuídos.

Ian Foster [44] define quatro fases da metodologia de projeto de algoritmos paralelos

chamado PCAM: 1) Particionamento, 2) Comunicação, 3) Aglomeração e 4) Mapeamento. A Figura 3.4 traz uma visualização do processo de paralelização de aplicações. Inicialmente existe uma descrição do problema relacionando seu objetivo, dados e algoritmos. O problema é, então, particionado em tarefas. Geralmente os dados ou o algoritmo (funções) são particionados em tarefas. A comunicação necessária entre as tarefas é estabelecida de acordo com a dependência de dados. A aglomeração da estrutura de tarefa junto com a comunicação é realizada para que, depois, sejam mapeadas a processos. Neste caso, uma tarefa representa um subproblema da aplicação e um processo representa o elemento computacional da tarefa (mapeamento da tarefa nos recursos).

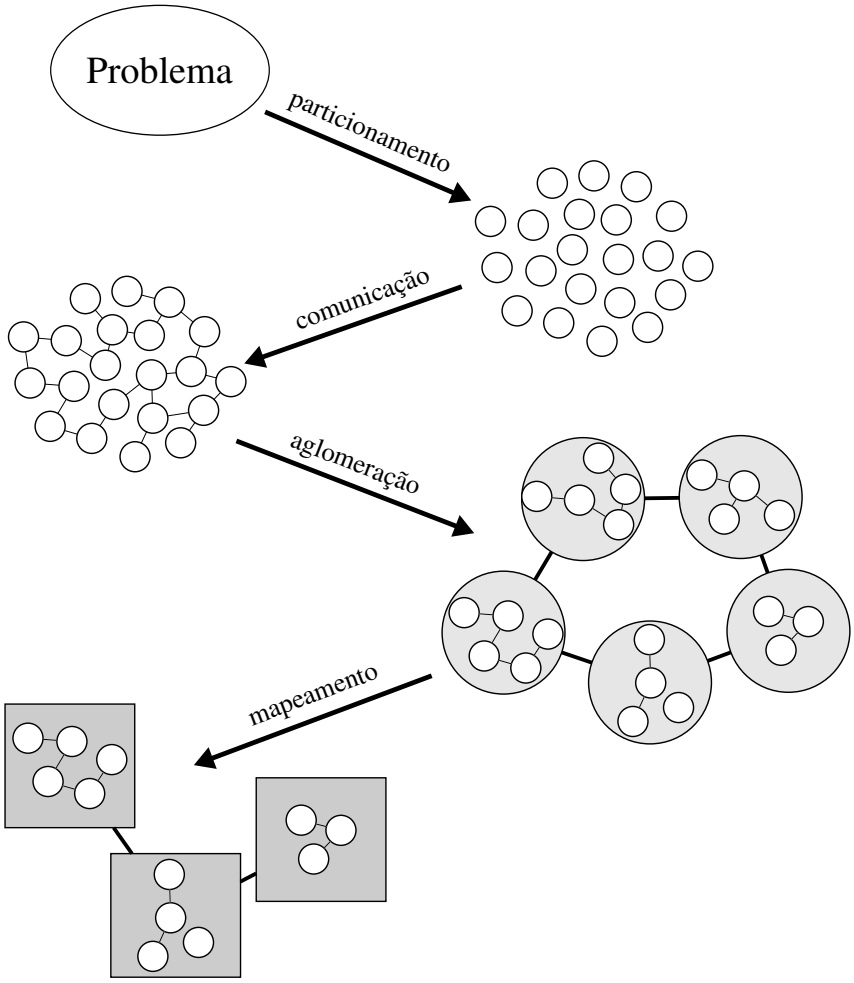


Figura 3.4: PCAM: uma metodologia de projeto de algoritmos paralelos [44].

A etapa de particionamento compreende duas técnicas complementares segundo Ian Foster [44]: decomposição de domínio e decomposição funcional. A decomposição de domínio divide os dados associados ao problema em pedaços. A cada partição, é associada uma operação (que pode ser a mesma entre elas). É comum uma operação requisitar dados de outras partições. O domínio costuma ser de 1, 2 ou 3 dimensões, dependendo do

problema. A decomposição funcional consiste em dividir a computação do problema em funções que possam ser executadas em paralelo. Os dados não são considerados para o particionamento, neste caso. É uma técnica menos comum comparada ao particionamento por decomposição de domínio mas é bastante usada para dividir problemas complexos que devem ser estruturados, por exemplo, como coleções de módulos conectados via interfaces.

Segundo Grama Ananth *et al.* [6], podem ser citados quatro tipos de técnicas de particionamento: decomposição recursiva, decomposição de dados, decomposição exploratória e decomposição especulativa. A recursiva indica o uso da estratégia de divisão e conquista. Nesta técnica, o problema é resolvido pela sua divisão recursiva entre subproblemas mais simples e independentes. A decomposição de dados sugere que o domínio seja particionado assim como na decomposição de domínio apresentada no livro de Ian Foster [44]. A decomposição exploratória é usada para decompor problemas de busca em um espaço de soluções, onde este espaço é particionado em partes menores. Na decomposição especulativa, N técnicas da solução do problema são tentadas simultaneamente e $N - 1$ delas são ignoradas assim que a primeira retornar uma resposta aceitável.

A comunicação entre cada partição é outra fase importante no projeto de algoritmos paralelos. No caso de não haver comunicação, as partições são totalmente independentes. No livro de Ian Foster [44], o autor categoriza livremente os padrões de comunicação ao longo de quatro eixos ortogonais: local ou global, estruturada ou não-estruturada, estático ou dinâmico, e síncrona ou assíncrona. A categoria local seria uma comunicação entre tarefas agrupadas enquanto global envolve todas. A comunicação estruturada sugere uma estrutura regular, como árvore ou grade, e a não-estrutura pode ser representada por um grafo arbitrário. Em uma comunicação estática, a identidade das tarefas parceiras nas comunicações não mudam conforme o tempo enquanto no dinâmico, sim. A comunicação síncrona tem uma coordenação no tempo entre os parceiros na comunicação e a assíncrona não contém essa cooperação. Outra forma de classificar a comunicação é através de sua estrutura. São consideradas a comunicação independente, onde não há interseção de dados, dependente, onde um DAG é usado para representar as dependências, e fortemente acoplado, onde as partições são altamente dependentes entre si.

A aglomeração das partições tende a aumentar a granularidade das tarefas e evitar altos custos de comunicação, considerando que tarefas que interagem bastante possam ser agrupadas. A definição de granularidade, nesta tese, envolve apenas quantidade de trabalho de uma tarefa. Ainda, a questão do tamanho da granularidade está relacionada com a quantidade de tarefas e o grau de paralelismo (largura), isto é, o quanto uma

aplicação pode ser particionada de modo que as partições possam executar simultaneamente. A granularidade geralmente é medida ou em unidades de trabalho ou em tempo de execução.

O mapeamento das tarefas em processos podem associar estaticamente tarefas a cada processo computacional ou permitir balanceamento de carga heurístico gerenciando um conjunto (*pool*) de tarefas entre os recursos disponíveis. O problema de encontrar um mapeamento ótimo para um conjunto de tarefas é NP-completo [42] e, por este motivo, heurísticas de escalonamento são tipicamente usadas.

3.2.1 Classificação de Aplicações Científicas Paralelas

De acordo com a metodologia PCAM [44], uma classificação de aplicações científicas paralelas pode ser feita considerando as três primeiras fases, de acordo com a estrutura de particionamento e comunicação. Tal classificação tem o objetivo de associar um perfil às aplicações em comum a fim de que uma mesma metodologia possa ser atribuída a aplicações de mesma classe, com nenhuma ou pouca alteração do projeto. Deste modo, cientistas podem desenvolver suas aplicações paralelas com menos esforço.

A Figura 3.5 representa a classificação utilizada nesta tese visando as aplicações científicas. Na etapa de particionamento, as decomposições de dados e algoritmos mais comuns são por domínio (por exemplo, aplicações de simulação de astrofísica cujos domínios costumam ser corpos celestes ou uma amostra tridimensional espacial), funcional (por exemplo, aplicações de modelagem climática que gera, em cada função, modelos da atmosfera, oceano, hidrologia, gelo entre outros), exploratória (por exemplo, aplicações que requerem análise de árvore de busca) e especulativa (por exemplo, aplicações meta-heurísticas).

A etapa de comunicação também caracteriza uma aplicação que pode ter suas tarefas independentes entre si, com dependência de dados entre si (formando um DAG) e fortemente acopladas. Uma comunicação fortemente acoplada possui um DAG estruturado e comunicação intensa.

Quando é feito o particionamento da aplicação em tarefas, muitas vezes, elas apresentam granularidade bastante fina e a etapa de aglomeração é necessária para tornar a granularidade mais grossa. Em casos como este, o valor ideal da granularidade é uma questão discutida na Seção 3.3.2. Ainda, por razões estruturais como diminuir o grau de comunicação entre as tarefas, a aglomeração de tarefas é requerida. Nesta etapa, a apli-

cação pode ser classificada por ter granularidade das tarefas fina, grossa ou desconhecida. As granularidades fina e grossa são representadas por valores relativos conhecidos previamente a execução. A granularidade desconhecida é aquela em que não se sabe seu valor *a priori* e deve ser conhecido apenas durante a execução. Na maioria das vezes, técnicas de particionamento dinâmico devem ser efetuadas para estabelecer um bom desempenho da aplicação paralela.

Decomposição	$\Rightarrow$	**Comunicação**	$\Rightarrow$	**Aglomeração**
por domínio funcional exploratória especulativa		independente dependente (DAG) fortemente acoplada		granularidade fina granularidade grossa granularidade desconhecida

Figura 3.5: Classificação das aplicações paralelas de acordo com as etapas PCAM.

3.3 Metodologia de Projeto de Aplicações Autônomas

Nesta tese, serão consideradas as quatro fases da metodologia PCAM para o projeto de algoritmos paralelos realizado pelos cientistas em geral para resolver seus problemas científicos. Em relação a primeira fase de particionamento, os tipos abordados serão decomposição de domínio, decomposição funcional e decomposição exploratória e decomposição especulativa. Em relação à fase de comunicação, os tipos podem ser independente, com dependência e com comunicação fortemente acoplada. A fase de aglomeração implica em associar uma granularidade às tarefas que serão tratadas como fina, grossa e desconhecida. A granularidade desconhecida representa o tamanho das tarefas onde não se sabe, *a priori*, quanto trabalho será executado. A granularidade varia de acordo com parâmetros do algoritmo da aplicação. A fase do mapeamento de tarefas aos processadores pode ser feita por um *middleware* dinamicamente ou estaticamente.

A autonomia é introduzida a medida que as etapas da metodologia PCAM evoluem. A utilização do *middleware* EasyGrid AMS é importante para tornar a aplicação científica autogerenciável. Assim, para que a aplicação aproveite eficientemente as funcionalidades autônomas que o EasyGrid AMS oferece, o modelo 1Ptask deve ser considerado ao invés do modelo tradicional 1Pproc propício para ambientes estáticos. Além disso, parâmetros da aplicação devem ser identificados para que sua autoconfiguração possa ser projetada. Tais parâmetros devem ser ajustados dinamicamente às mudanças do sistema. O escalonamento dinâmico das tarefas feito pelo EasyGrid AMS permite, ainda, a auto-otimização da aplicação e seu mecanismo de tolerância a falhas incorpora a autorrecuperação. Como

já mencionado, a característica autônoma de autoproteção não será tratada neste trabalho.

O desenvolvimento de uma aplicação paralela e autônoma é composto por duas fases. A primeira fase (etapas 1 e 2) é realizada estaticamente envolvendo a construção de um modelo de desempenho. A segunda fase (etapas 3 e 4) utiliza este modelo durante a execução da aplicação para sua adaptação ao ambiente.

1. Colocar, primeiramente, a aplicação no modelo 1Ptask.
2. Identificar parâmetros ou valores importantes da aplicação paralela para sua configuração, que podem ser, por exemplo:
   - granularidade;
   - número de tarefas;
   - quantidade de dados em memória; e
   - quantidade de dados de comunicação.
3. Adaptar tais parâmetros de acordo com informações do sistema (autoconfiguração), o que pode ser feito no algoritmo da aplicação ou no *middleware*. Esta etapa aplica os métodos PCAM continuamente ainda considerando o modelo 1Ptask:
   (a) particionar os dados e operações da aplicação em partições (etapa de particionamento);
   (b) identificar dependências de dados entre tarefas e gerar DAG (etapa de comunicação);
   (c) Aglomerar as partições em tarefas (etapa de aglomeração), mas manter a granularidade relativamente fina.
4. Usar o *middleware* EasyGrid AMS para o mapeamento, escalonamento dinâmico de tarefas e tolerância a falhas (auto-otimização e autorrecuperação) continuamente.

A etapa 1 sugere que a aplicação seja projetada sob o modelo 1Ptask, isto é, tenha várias tarefas com granularidade fina. A etapa 2 implica em identificar parâmetros para realizar a autoconfiguração. Esses parâmetros podem ser a granularidade (que pode ser ajustada dinamicamente), o número de tarefas (que pode ser aumentado ou diminuído), quantidade de dados em memória (auxilia na alocação de recursos) e quantidade de dados de comunicação (auxilia na alocação de recursos). Uma vez identificados os parâmetros,

a etapa 3 consiste da autoconfiguração da aplicação. Os parâmetros da aplicação e do sistema podem ser ajustados dinamicamente de modo a adaptar a execução dos processos da aplicação ao sistema. É nesta etapa que as fases do PCAM de particionamento, comunicação e aglomeração são aplicadas dinamicamente com o intuito de restruturar e reconfigurar a aplicação paralela. Por fim, uma vez a aplicação estando no modelo 1Ptask, a etapa 4 indica o uso do *middleware* EasyGrid AMS para o mapeamento de tarefas dinâmico e automático (escalonamento dinâmico) e para adicionar outras características autônomas como tolerância a falhas. A autoconfiguração da aplicação é, então, integrada aos outros componentes autônomos do *middleware*.

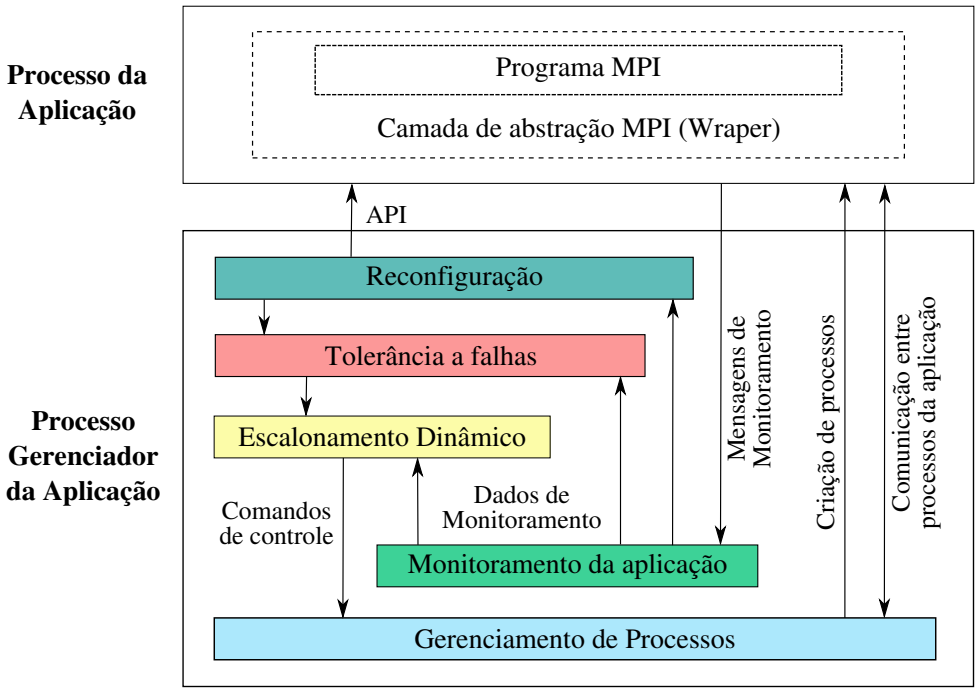


Figura 3.6: Arquitetura de camadas do EasyGrid AMS.

Pela arquitetura de camadas do EasyGrid AMS na Figura 3.6, pode-se identificar os aspectos de autonomia das aplicações presentes na metodologia. Como já descrito na Seção 3.1.2, a cada tarefa da aplicação existe um gerenciador do *middleware* associado. No processo da aplicação, existe o programa em MPI e a camada de abstração MPI pela qual o *middleware* intercede a aplicação. No processo gerenciador, existem 5 camadas integradas. A mais abaixo na figura é a camada do gerenciamento de processos, que é responsável por criar processos e redirecionar as mensagens da aplicação. A camada de monitoramento é responsável pelo **autoconhecimento** da aplicação, recebendo mensagens implícitas (pela camada de abstração) sobre a execução de seus processos. Além disso, alimenta as camadas de escalonamento dinâmico, tolerância a falhas e reconfiguração com informações atuais do sistema e da aplicação. A camada de escalonamento

dinâmico (ver Seção 3.1.2.2) representa a **auto-otimização** da aplicação. Ela envia comandos de controle para a camada de gerenciamento de processos e se comunica com as camadas de tolerância a falhas e reconfiguração (por transitividade). A camada de tolerância a falhas (ver Seção 3.1.2.3) é responsável pela **autorrecuperação** da aplicação. A **autoconfiguração** está localizada na camada de reconfiguração, que modifica o estado do processo da aplicação de acordo com as informações de monitoramento, passando informações para a camada de tolerância a falhas e escalonamento dinâmico.

Esta tese, portanto, propõe o uso da metodologia de projeto de aplicações autônomas que inclui basicamente o uso do modelo de particionamento 1Ptask, o uso de um AMS, como o EasyGrid AMS, e a implementação da autoconfiguração específica por aplicação. Em outras palavras, a tese recomenda o uso do modelo 1Ptask inicialmente como o processo de paralelização da aplicação, diferentemente do modelo tradicional. Uma vez estando no modelo 1Ptask, as tarefas paralelas da aplicação podem usufruir do gerenciamento e monitoramento de processos pertencentes a um AMS, assim como de seu escalonamento dinâmico e mecanismo de tolerância a falhas. A etapa de autoconfiguração é tratada especificamente por aplicação, considerando suas características próprias (como granularidade e quantidade de tarefas). Logo, a camada de reconfiguração apresentada na Figura 3.6 é uma funcionalidade desenvolvida na presente tese. Os Capítulos 4, 5, 6, 7 e 8 mostram como foram realizadas o particionamento no modelo 1Ptask e a autoconfiguração para um conjunto de aplicações diferentes (cada capítulo representa uma aplicação). Ainda, características autônomas do EasyGrid AMS podem ser alteradas pelo cientista para afinar a sua aplicação, como é feito no caso do escalonamento dinâmico no Capítulo 6.

3.3.1 Autoconfiguração das Aplicações

A autoconfiguração de uma aplicação é realizada de acordo com suas características específicas e pode ser implementada modificando-se o *middleware* AMS ou utilizando-se o algoritmo da própria aplicação. A implementação pelo *middleware* indica que ações de mudanças dinâmicas de parâmetros da aplicação são realizadas por intervenção do gerenciador. Ele pode interceptar mensagens da aplicação e as tratar conforme indicado pelo desenvolvedor. Em outras palavras, o desenvolvedor pode realizar modificações – geralmente por inclusão de código – nos próprios gerenciadores do *middleware* já que ele é um AMS e é específico por aplicação. Esta é uma maneira mais trabalhosa porém pode causar bom impacto no desempenho da aplicação (ver Capítulo 6).

Quando a implementação da autoconfiguração é feita na própria aplicação, o desenvolvedor geralmente necessita de informações da execução da aplicação para tomar uma decisão. Estas informações são obtidas explicitamente pela aplicação através de funções descritas em uma API, já que o EasyGrid AMS faz o seu monitoramento no sistema. Um exemplo seria uma aplicação que tivesse a granularidade desconhecida e recorresse a informações do sistema para realizar particionamento e aglomeração dinâmico de tarefas. As informações são transferidas à aplicação através de funções explicitamente chamadas por ela.

A API oferece funções como: captar o número de tarefas executando localmente e globalmente ou a carga geral das máquinas. Algumas foram implementadas para as aplicações avaliadas nesta tese, mas elas podem ser implementadas de acordo com características específicas das aplicações, como o uso de memória e comunicação também.

3.3.2 Parâmetros de Configuração de Aplicações

Nesta metodologia, definir a granularidade das tarefas é um passo importante para o desempenho da aplicação em uma *e*-infraestrutura. É comum que, durante o particionamento, uma aplicação científica tenha tarefas iguais ou com trabalho determinado. Além disso, a etapa de aglomeração permite agrupar tarefas ou para gerar tarefas com mais trabalho ou para diminuir custos de comunicação. Assim, as tarefas podem ter granularidade fina ou grossa. Existe ainda o caso de realizar o particionamento e não conhecer a quantidade de trabalho. Estruturas condicionais do algoritmo para o problema da aplicação podem gerar tarefas com granularidade desconhecida. Neste caso, pode-se particionar e aglomerar as tarefas dinamicamente para determinar suas granularidades.

Em relação a escolha da granularidade, uma fina demais pode acarretar um acúmulo de sobrecarga no gerenciamento das tarefas. Por outro lado, uma granularidade grossa não permite um gerenciamento flexível principalmente em ambientes distribuídos heterogêneos e compartilhados, como é o caso das *e*-infraestruturas. Portanto, no modelo 1Ptask, o valor da quantidade de trabalho deve ser suficientemente pequeno, de modo a não ressaltar sobrecarga do *middleware* de gerenciamento. Geralmente o valor da granularidade é definido através de alguma análise prévia do trabalho das tarefas.

O número de tarefas está diretamente relacionado à granularidade das tarefas. Geralmente, quanto mais fina é a granularidade, maior é o número de tarefas. O gerenciador das tarefas deve ser capaz de lidar com esse grande número de tarefas e realizar um escalonamento com baixa sobrecarga.

Ainda, conhecer outras características da aplicação podem ajudar a melhorar o desempenho da aplicação. Por exemplo, a aplicação pode conhecer os seus recursos massivamente usados (como memória ou comunicação) e realizar ajustes de parâmetros para otimizar a utilização deles.

3.4 Casos de Estudo na e-Ciência

Aplicações da *e*-ciência são soluções computacionais desenvolvidas por cientistas para resolver seus problemas em sistemas de larga escala computacional, chamados de *e*-infraestruturas. O termo refere-se àquelas aplicações que tratam diversos problemas das áreas de física, biologia, medicina, engenharia e economia, por exemplo.

Neste trabalho, cinco grupos de aplicações, distintos em sua estrutura de comunicação, particionamento de dados e tipo de granularidade, são considerados para a avaliação da proposta. Como existem um grande número de aplicações da *e*-ciência em cada grupo, apenas uma aplicação real foi implementada e avaliada. O objetivo é mostrar que aplicações da *e*-ciência já existentes podem tornar-se autônomas através do uso do *middleware* EasyGrid AMS. Os grupos investigados são: simulação Monte Carlo, decomposição de domínio fortemente acoplado, busca em árvore *branch-and-prune*, *workflow* científico e projeção em domínio geoespacial.

O grupo de **simulações Monte Carlo** engloba algoritmos massivamente paralelos, isto é, as partições do domínio não apresentam comunicação de dados entre si e a granularidade pode ser fina ou grossa (e, portanto, são conhecidas *a priori*). A **decomposição de domínio fortemente acoplados** indica um particionamento de domínio cujas partições necessitam de um alto grau de comunicação e a granularidade das tarefas tende a ser fina. Algoritmos de **busca em árvore *branch-and-prune*** usam o particionamento exploratório. Existe dependência de dados entre as tarefas e sua granularidade geralmente é desconhecida. Os ***workflows* científicos** são compostos por partições funcionais e apresentam tarefas de granularidade grossa que possuem dependência de dados de acordo com o fluxo de trabalho. Aplicações do grupo de **projeção em domínio geoespacial** têm um particionamento do domínio sem comunicação entre si (a comunicação é apenas usada para retornar os resultados) e manipulam uma larga escala de dados tanto de entrada quanto de saída. A granularidade tende a ser desconhecida também. A Tabela 3.1 apresenta um resumo das características dos grupos de aplicações tratados neste trabalho.

Tabela 3.1: Resumo das características dos grupos de aplicações tratados nesta tese.

	Decomposição	Comunicação	Granularidade
Simulação Monte Carlo	Domínio	Independente	Fina
Decomposição de domínio fortemente acoplado	Domínio	Fortemente acoplado	Fina
Busca em árvore *branch-and-prune*	Exploratória	Dependente	Desconhecida
Workflow científico	Funcional	Dependente	Grossa
Projeção em domínio geoespacial	Domínio	Independente	Desconhecida

3.5 Critério Geral de Avaliação

Esta seção atribui o critério de avaliação utilizado para analisar os resultados experimentais obtidos com as aplicações previamente mencionadas e classificadas. Apenas as características autônomas de autoconfiguração e auto-otimização serão efetivamente avaliadas. A característica de autorrecuperação já foi avaliada em trabalhos anteriores de pesquisa [28, 109, 86] e eles mostraram que o método de tolerância a falhas funciona eficientemente junto às aplicações, detectando e recuperando das falhas associadas a cenários comuns nas *e*-infraestruturas. O objetivo principal é verificar a eficiência da solução proposta considerando que o mecanismo de tolerância a falhas funciona eficientemente e adiciona baixa sobrecarga. No entanto, pretende-se futuramente avaliar estes mecanismos, já que podem estar defasados em relação às implementações atuais. em relação à autoproteção, propostas de técnicas são requeridas e, logo, não serão consideradas aqui.

A avaliação será conduzida em cada uma das aplicações caso de estudo descritas na Seção 3.4. Para cada aplicação, existe um capítulo que é formado por uma explicação do problema, por abordagens de desenvolvimento já existentes, pela abordagem proposta nesta tese e pela avaliação conforme descrita no início desta seção. As abordagens tradicionais consideradas já possuem características de auto-otimização através de implementações simples de balanceamento de carga mas não são implementações completamente autônomas e nem as mais eficientes. O objetivo não é comparar com a melhor solução, e sim, comparar com uma solução comumente usada (geralmente por ser simples de implementar) e mostrar que a versão implementada utilizando a metodologia proposta é competitiva e mais eficiente que as abordagens tradicionais. Sendo assim, a proposta pode ser considerada viável por obter bom desempenho com baixo esforço de desenvolvimento.

A eficiência de uma abordagem paralela é calculada quando a aplicação possui uma versão sequencial disponível. Neste caso, seu valor equivale a $\frac{speed-up}{P}$, onde *speed-up* é a razão entre o tempo sequencial e a média do tempo paralelo e P é o número de processadores ou unidades de processamento. Caso contrário, a abordagem proposta é comparada com alguma abordagem paralela tradicional representativa.

3.6 Resumo

O início deste capítulo foi responsável por apresentar o problema de projetar aplicações autônomas da *e*-ciência. O problema envolve a dificuldade de gerenciar uma *e*-infraestrutura dinâmica, heterogênea, compartilhada, ao passo que os cientistas necessitam deste ambiente de execução para resolver seu problema através de uma implementação computacional. As formas de gerenciamento consideradas podem ser orientadas ao usuário, recursos e aplicações, onde a orientação baseada em recursos é tida como a convencional. Diferentemente de um sistema gerenciador convencional, um sistema gerenciador de aplicações (AMS) é visto, neste trabalho, como uma solução que atende aos requisitos de QoS (*Quality of Service*) de uma aplicação da *e*-ciência. Com ele, é possível desenvolver aplicações autônomas, que consideram suas características internas para tomar decisões. O EasyGrid AMS foi apresentado como *middleware* escolhido assim como seu modelo de particionamento 1Ptask.

Em seguida, a metodologia de projeto de aplicações paralelas existente, o PCAM, foi descrito de forma que as etapas de particionamento, comunicação, aglomeração e mapeamento possam ser empregadas neste trabalho. Além disso, das três primeiras etapas obtém-se uma classificação de aplicações científicas que é usada para entender as características de cada aplicação estudo de caso. A metodologia de desenvolvimento de aplicações autônomas usando o EasyGrid AMS foi descrita posteriormente. Ela segue os passos PCAM e introduz as etapas de autoconfiguração usando modelo 1Ptask, auto-otimização e autorrecuperação.

Como as aplicações têm características diferentes e existe um AMS para cada, as aplicações casos de estudo apresentam uma implementação distinta. Por esta razão, 5 capítulos foram projetados para descrever cada um dos casos, a solução proposta usando a metodologia explicada neste capítulo e as avaliações experimentais realizadas para fortalecer a proposta.

Capítulo 4

Simulação Monte Carlo – Thermions

Nos dias de hoje, o método de Monte Carlo é uma ferramenta matemática comumente utilizada para simular problemas que podem ser representados por processos estocásticos em diversas áreas como engenharia [14], física [13], biologia [98], medicina [7, 22], finanças e negócios [66, 121]. Eles fazem parte do grupo de aplicações massivamente paralelas e, por isso, seu particionamento costuma ser simples. Para este tipo de aplicação, o domínio do problema é decomposto em subdomínios sem dependências entre si e a granularidade pode ser facilmente projetada como fina.

Este capítulo engloba a descrição do método Monte Carlo, de sua versão paralela tradicional e da versão proposta neste trabalho. A aplicação *Thermions* é usada como estudo de caso como também nos experimentos para avaliação. Os cenários experimentais consideram ambientes de execução dedicados, heterogêneos e compartilhados, podendo-se analisar o comportamento das abordagens paralelas diante de características encontradas nas *e*-infraestrutura.

4.1 Método Monte Carlo

O método de Monte Carlo (MMC) pode ser descrito como um método estatístico, no qual utiliza-se uma sequência de números aleatórios para a realização de uma simulação. Este método é muito usado para obter soluções numéricas para problemas que são complicados de resolver analiticamente. Ele necessita ser alimentado por um gerador de números aleatórios para que, dada uma função de distribuição de probabilidade que caracteriza os processos físicos do fenômeno em estudo, possa aplicar técnicas de amostragens que fazem a conexão entre as funções de probabilidade e os números aleatórios. Como resultado, obtêm-se valores médios que estão relacionados com grandezas físicas integrais de

interesse.

O algoritmo de simulações Monte Carlo variam de acordo com a aplicação, mas tendem a seguir os passos básicos do Algoritmo 1. Inicialmente, o domínio do problema deve ser definido. Em seguida, entradas são randomicamente geradas de uma distribuição de probabilidade sobre este domínio. Sobre as entradas geradas, uma computação determinística deve ser realizada, gerando resultados. Estes valores devem ser, então, agregados para obter-se soluções.

Algoritmo 1: Algoritmo básico de uma simulação Monte Carlo.

1 define um domínio
2 gera entradas randomicamente sobre o domínio
3 realiza uma computação determinística sobre as entradas
4 agrega os resultados

4.1.1 Caso de Estudo: Thermions

Thermions representa uma aplicação real da e-ciência e é descrita por um método numérico que usa o Monte Carlo para simular a dispersão térmica em meios porosos periódicos. Um *thermion* seria uma partícula hipotética que carrega um certa quantidade de energia [110, 29]. O estudo da dispersão térmica em meios porosos envolve áreas da indústria do petróleo em temas como dispersão de poluentes, secagem de madeira, escoamento de águas subterrâneas e conversores catalíticos, e da medicina em temas como absorção de medicamentos pela pele ou pelas vias aéreas. Como existe uma grande variedade de materiais e substâncias porosas, tal estudo não fica apenas limitado a estas áreas.

A dispersão das partículas de *thermions* no meio poroso é feita através de repetidas simulações. Basicamente, o método repete o processo iterativo de mover o *thermion*, tentar mudar de meio, adicionar um deslocamento difuso e convectivo, armazenar a posição de cada *thermion* e calcular os momentos parciais da distribuição. Ao final da simulação, são calculados os momentos centrados totais e, por regressão linear, as grandezas físicas associadas aos mesmos: a velocidade média do centroide da distribuição de *thermions* e o coeficiente do tensor efetivo de dispersão [29].

4.2 Simulações Monte Carlo em Paralelo

Os métodos Monte Carlo são massivamente paralelos [38, 111]. As simulações são independentes entre si e basta dividi-las entre os processadores para cada um processar, então, os cálculos específicos do método.

Por sua vez, o método *thermions* considera o deslocamento de todas as N partículas no meio poroso a cada incremento de tempo discretizado. É este deslocamento que sofre a ação aleatória inerente do método Monte Carlo. Ao final de cada deslocamento, os resultados de cada *thermion* são armazenados com vistas ao cálculo dos momentos finais. O método pode ser então executado em vários p processadores, cada qual contendo uma determinada fração do número total de *thermions* e, somente ao final, transferir os valores calculados por cada processador de modo a calcular os momentos totais da distribuição.

Uma solução MPI existente do tipo mestre-trabalhador para o método Monte Carlo e, consequentemente para, a aplicação *thermions*, é tida neste trabalho como a solução tradicional. Ela simplesmente cria um *pool* de partículas e usa um processo mestre para distribuir o cálculo sobre cada partícula para os processo trabalhadores conforme a demanda. O processo mestre inicialmente distribui p partículas para p processos trabalhadores. Conforme os processos trabalhadores forem terminando, eles requisitam ao mestre um novo dado. Esta interação sob demanda é realizada até que as partículas terminem sob o comando do processo mestre. Nota-se que esta solução já é uma tentativa de auto-otimizar a execução do método através de seu balanceamento de carga reativo (o balanceamento é ativado sob demanda, isto é, quando um processo fica sem trabalho e precisa requisitar mais).

O Algoritmo 2 representa os passos de execução do processo mestre da versão MPI tradicional para a simulação Monte Carlo. As entradas do algoritmo são o número de partições N do domínio D da simulação e o número total P de processos trabalhadores. O processo mestre, então, é responsável por fazer o particionamento do domínio D em N partições e colocá-las no conjunto T na linha 1. Para cada processo trabalhador i, o mestre remove uma tarefa t_i de T (linha 3) e envia para i(linha 4). Para continuar a sequência das tarefas em T, na linha 5, o índice i é atualizado para o valor P, a próxima tarefa. A partir da linha 6, há uma repetição para receber a solução de um processo j (o que indica o fim de uma tarefa e o pedido de uma nova tarefa) na linha 7, remover uma nova tarefa de T (linha 8), enviar t_i para a tarefa j que acabou de terminar uma tarefa (linha 9) e manipular a solução (linha 10). Esta repetição é feita até que o conjunto T

esteja vazio e o índice i deve ser incrementado a cada laço.

Algoritmo 2: Algoritmo do processo mestre para versão tradicional de uma simulação Monte Carlo.

Entrada: N - número de partições
D - domínio
P - número de processos trabalhadores
Saída: *solucao* - solução encontrada

1 $T \leftarrow particiona(D, N)$
2 **para** $i \leftarrow 0$ *a* $P - 1$ **faça**
3 remove t_i de T
4 envia t_i para i
5 $i \leftarrow P$
6 **enquanto** $T \neq \varnothing$ **faça**
7 recebe *solucao* de j
8 remove t_i de T
9 envia t_i para j
10 manipula *solucao*
11 $i \leftarrow i + 1$

O Algoritmo 3, por sua vez, representa o processo trabalhador. Ele irá receber uma tarefa t do mestre na linha 1, fará o cálculo da simulação (linha 2), isto é, o trabalho da tarefa, e enviará a solução parcial para o mestre na linha 3. Geralmente, as componentes aleatórias são geradas pelo mestre e recebidas no início do algoritmo incorporadas na tarefa t.

Na verdade, este algoritmo mestre-trabalhador serve para qualquer aplicação que possa ser colocada na forma *bag-of-tasks* - BoT, isto é, existe um "saco" de tarefas independentes geradas pelo processo mestre e cada uma é selecionada para ser executada sob demanda pelos processos trabalhadores. Este algoritmo apresenta o clássico problema de gargalo do mestre, principalmente ao coletar as soluções. Vários processos, tentam, simultaneamente, enviar suas soluções para o mestre pela rede e ele tenta suprir essas mensagens. No entanto, geralmente o mestre fica ocupado e não consegue lidar rapidamente com as mensagens, atrasando a execução. Dentre as soluções para melhorar este problema, encontram-se as hierarquias distribuídas que usam tarefas sub-mestres para lidar com seus grupos de processos. Esta solução, por exemplo, requer uma implementação mais complexa.

Algoritmo 3: Algoritmo do processo trabalhador para versão tradicional e EasyGrid AMS de uma simulação Monte Carlo.

1 recebe t de mestre
2 calcula simulação em t
3 envia solução para mestre

4.3 Simulações Monte Carlo Autônomas

A versão autônoma e paralela de uma simulação Monte Carlo, , desenvolvida nesta tese de acordo com a metodologia proposta, é proveniente da integração entre o EasyGrid AMS e a aplicação. Tal integração é dada pelo uso do modelo de particionamento 1PTask, que indica que a aplicação deve ser particionada em diversas tarefas de granularidade fina e cada tarefa é um processo. No caso da aplicação *Thermions*, cada tarefa representa o cálculo de uma partícula e a granularidade é conhecida, portanto.

A etapa de decomposição do problema define as tarefas da aplicação que apresentam a mesma quantidade de trabalho. Na etapa de comunicação, as tarefas são postas como independentes. Na etapa de aglomeração, geralmente, as tarefas são aglomeradas mantendo-se ainda a sua granularidade fina. A ideia é usar o modelo 1Ptask presente na metodologia. Seu valor pode ser determinado pela quantidade de trabalho necessária para ser executada em poucos segundos (por exemplo, de 2 a 10 segundos). Isto é feito para ajudar a eliminar sobrecarga de gerenciamento de processo, já que cada tarefa é um processo no modelo 1Ptask. A etapa de mapeamento é realizada por um *middleware* de gerenciamento. As diversas tarefas são, portanto, gerenciadas pelo EasyGrid AMS. Como as tarefas são simples e facilmente particionadas, não foi encontrada a necessidade de autoconfigurar parâmetros da aplicação. A autoconfiguração fica apenas por conta do *middleware* que selecionará heuristicamente o melhor mapeamento das tarefas entre os processadores de tempos em tempos e, portanto, não há autoconfiguração a nível de aplicação.

Da mesma maneira que a versão tradicional de simulação Monte Carlo, a versão EasyGrid AMS apresenta uma tarefa mestre responsável por realizar o particionamento e envio dos dados e captar as soluções das tarefas trabalhadoras. O Algoritmo 4 mostra os passos executados pela tarefa mestre. Diferentemente da versão tradicional que faz o envio de partições sob demanda, o particionamento é realizado totalmente no início (linha 1) e cada parte é enviada para todas as tarefas trabalhadoras associadas (linhas 2, 3 e 4). O domínio do problema é dividido em subdomínios com a mesma quantidade de trabalho e

Algoritmo 4: Algoritmo da tarefa mestre para versão EasyGrid AMS de uma simulação Monte Carlo.

Entrada: N - número de partições
Saída: *solucao* - solução encontrada

1 $T \leftarrow particiona(D, N)$
2 **para** $i \leftarrow 0$ *a* $N - 1$ **faça**
3 remove t_i de T
4 envia t_i para i
5 **para** $i \leftarrow 0$ *a* $N - 1$ **faça**
6 recebe *solucao* de j
7 manipula *solucao*

granularidade fina, resultando em muitas partições e, consequentemente, tarefas (modelo 1Ptask). Em seguida, nas linhas 5, 6 e 7, existe a espera pelas soluções de qualquer tarefa e cada solução é combinada de modo a gerar o resultado final. O algoritmo para as tarefas trabalhadoras é idêntico àquele da versão tradicional, representado pelo Algoritmo 3.

Portanto, o EasyGrid AMS recebe todas as tarefas da aplicação, distribui entre os gerenciadores de máquinas (ver Seção 2.3.7 do Capítulo 2) e executa uma por vez em cada processador ou núcleo conforme vão ficando prontas para a execução. Uma tarefa é considerada pronta quando o *middleware* já contém todas as mensagens necessárias para iniciar seu processamento intensivo. O EasyGrid AMS intercepta todas as mensagens da aplicação e, assim que as mensagens de dependência chegam (no caso, a dependência é entre mestre e trabalhador), as respectivas tarefas são inseridas nas suas filas de prontos em cada gerenciador de máquinas. Desta forma, as tarefas não ficam ociosas esperando receber as mensagens e permitem executar outras tarefas que já estão prontas. No caso de um desbalanceamento de carga, o EasyGrid AMS é capaz de balancear as suas filas de tarefas que não estão em execução e promover um eficiente escalonamento dinâmico. No caso de falha de tarefas, elas são reexecutadas em outra máquina (ou na mesma, se a falha for de *software*) já que possuem granularidade fina.

A principal diferença existente entre os algoritmo tradicional considerado e o proposto encontra-se na maneira como os dados resultantes são enviados de volta para o mestre. Na versão tradicional, os dados são enviados sem nenhum controle, o que pode ocasionar gargalo e, portanto, queda de desempenho. Na versão proposta, a API do EasyGrid AMS disponibiliza uma função de redução[1], semelhante ao *MPI_Reduce*. No entanto,

[1]A função *AMS_Reduce* possui os mesmos argumentos da *MPI_Reduce* do padrão MPI, excetuando o quinto argumento, que indica a operação sobre os dados. No caso do EasyGrid AMS, o valor do argumento pode ser *AMS_USER* e o desenvolvedor pode especificar seu próprio tratamento dos dados.

a operação de redução deve ser implementada pelo desenvolvedor. Com esta função, os dados são reduzidos pelos processos gerenciadores de máquinas do EasyGrid AMS correspondentes aos processos da aplicação, eliminando possíveis gargalos. O processo mestre da aplicação apenas reunirá os dados restantes. É importante ressaltar que esta implementação é específica para este tipo de aplicação, onde os dados resultantes podem ser independentemente reunidos.

4.4 Avaliação Experimental

Os experimentos foram executados em um *cluster* chamado Oscar na Universidade Federal Fluminense. Este *cluster* é composto por 42 máquinas, 2 *switches* gigabit e um armazenador de dados de 14,5 TB. Das 42 máquinas, 40 são usadas para a computação de *jobs*. Cada máquina tem 2 processadores de quatro núcleos Intel Xeon de 2,00 GHz (8 núcleos no total) e 16 GB de memória RAM. Todos os nós são interconectados por *switches* gigabit Ethernet. No entanto, por motivos de disponibilidade, apenas 16 das 40 máquinas foram utilizadas, de forma dedicada.

Uma implementação da aplicação *thermions* foi empregada nestes experimentos baseada na solução apresentada em [110], tanto para a versão MPI tradicional quanto para a versão EasyGrid AMS. A instância utilizada possui 10000 *thermions* e cada simulação usando um *thermion* gasta cerca de 4 segundos nas máquinas utilizadas. Portanto, nestes experimentos, este valor representa a granularidade de cada tarefa no modelo 1Ptask, já que cada tarefa calcula um *thermion*. No caso da versão com o EasyGrid AMS, o número de tarefas é sempre 10000 (e também o número de processos).

4.4.1 Descrição dos Cenários Paralelos

Os cenários para avaliar a proposta são listadas a seguir:

Cenário 1 Todas as H máquinas são dedicadas.

Cenário 2 Metade das H máquinas possuem cargas externas estáticas.

Cenário 3 Metade das H máquinas possuem cargas externas dinâmicas.

Em todos os cenários, a variação do número de máquinas H é de 2, 4, 6, 8, 10, 12, 14, e 16, lembrando que cada máquina possui 8 núcleos no total. No cenário 1, todas

as máquinas estão dedicadas para avaliar os *speed-ups* e a eficiência da proposta. O segundo cenário possui 8 cargas externas em cada $\frac{H}{2}$ máquinas. Uma carga externa é um processo que faz uso intensivo de CPU durante o tempo que executar. As cargas externas no cenário 2 são estaticamente alocadas, permanecendo nelas até o final da execução da aplicação. A mesma estratégia é conduzida no cenário 3, mas as cargas são dinâmicas. As 8 cargas rodam por 10 segundos em cada $\frac{H}{2}$ máquinas e alternam sua execução nas outras máquinas. Então, cada $\frac{H}{2}$ máquinas têm 8 cargas por 10 segundos, depois as outras $\frac{H}{2}$ máquinas têm 8 cargas por 10 segundos e assim sucessivamente.

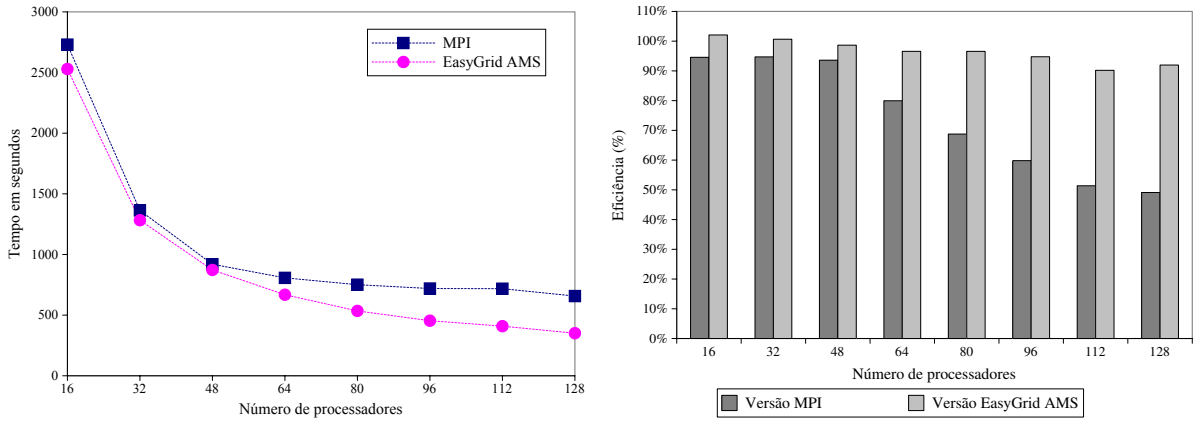
4.4.2 Resultados

O objetivo da avaliação do primeiro cenário é comparar as duas versões MPI e EasyGrid AMS em um ambiente dedicado. A versão tradicional apresenta um algoritmo do tipo mestre-trabalhador e é esperado um declínio da eficiência conforme o aumento do número de processadores devido a possíveis gargalos de comunicação entre mestre (centralizado) e trabalhadores. Por sua vez, a versão do EasyGrid AMS possui uma visão distribuída que permite a não ocorrência de tal gargalo. A avaliação do segundo cenário ajuda a simular um ambiente heterogêneo, pois as cargas externas estáticas (fixas nas máquinas) reduzem o poder computacional da máquina.

A Tabela 4.1 mostra os principais resultados de desempenho para a versão tradicional e EasyGrid AMS no cenário 1. Variando o número de processadores P (número de núcleos total), para cada versão, a média dos tempos em segundos (de 5 rodadas), o intervalo de confiança de 96%, o *speed-up* (razão entre o tempo sequencial e a média do tempo paralelo) e eficiência ($\frac{speed-up}{P}$). O tempo de execução sequencial obtido foi de 41311,23 (cerca de 11,48 horas) e este valor é usado para o cálculo dos *speed-ups*.

Tabela 4.1: Principais resultados de desempenho das abordagens tradicional e EasyGrid AMS para a aplicação *Thermions* no cenário 1.

	Versão tradicional				Versão EasyGrid AMS			
P	Média	I.C.	*Speed-up*	Eficiência	Média	I.C.	*Speed-up*	Eficiência
16	2730,22	0,52	15,13	95%	2529,26	0,07	16,33	102%
32	1363,02	0,08	30,31	95%	1282,62	0,21	32,21	101%
48	919,48	0,16	44,93	94%	872,44	0,30	47,35	99%
64	807,27	0,14	51,17	80%	668,38	0,23	61,81	97%
80	751,04	0,12	55,01	69%	534,74	0,30	77,26	97%
96	719,34	0,19	57,43	60%	454,15	0,36	90,96	95%
112	717,95	0,31	57,54	51%	408,96	0,43	101,02	90%
128	657,22	1,02	62,86	49%	350,93	0,38	117,72	92%



(a) Tempo de execução em segundos. (b) Eficiência.

Figura 4.1: Resultados no cenário 1 para a aplicação *Thermions*.

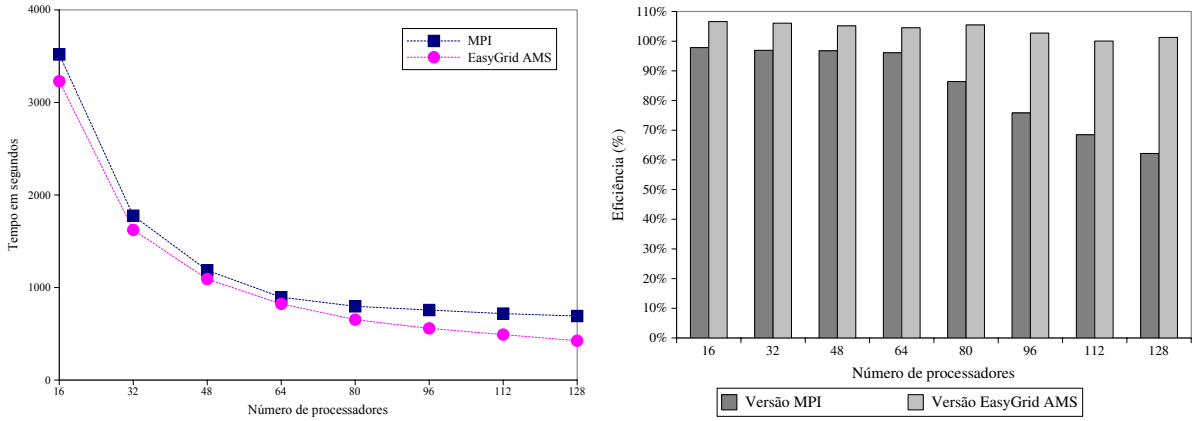
Através dos gráficos da Figura 4.1, é possível analisar melhor os resultados de desempenho das propostas em um cenário dedicado. A Figura 4.1(a) mostra as curvas do tempo de execução ao aumentar o número de processadores para a versão MPI tradicional e EasyGrid AMS. A curva cujos pontos são representados por um quadrado referencia a versão tradicional enquanto a curva com pontos em forma de círculo referencia a versão EasyGrid AMS. A Figura 4.1(b) mostra duas barras que representam a eficiência em relação ao tempo de execução sequencial obtido para cada número de processadores utilizado. A barra mais escura referencia a versão tradicional e a barra mais clara referencia a versão EasyGrid AMS.

Os resultados mostram (ver Figura 4.1(a)) que a versão do EasyGrid AMS apresenta sempre um tempo de execução no ambiente dedicado inferior ao da versão MPI tradicional e, a partir do uso de 64 processadores, a diferença entre o tempo de execução de cada tende a crescer. A Figura 4.1(b) indica uma maior eficiência (acima de 90%) do EasyGrid AMS conforme o aumento do número de processadores. Mostra ainda que a versão tradicional é pouco escalável, uma vez que sua eficiência declina diante do aumento da quantidade de processadores. Isto se deve ao fato do gerenciamento de tarefas ser feito por um mestre centralizado. Por outro lado, o EasyGrid AMS não possui este problema já que sua estrutura é hierarquizada e a aplicação é tratada por ela, embora a aplicação esteja implementada no modelo mestre-trabalhador. E mesmo quando o gargalo do mestre não afeta a execução (com 16, 32 e 48 CPUs), o EasyGrid AMS mostra-se mais eficiente que a versão tradicional, devido ao seu escalonamento dinâmico proativo (auto-otimização).

Os gráficos da Figura 4.2 apresentam resultados resumidos da Tabela 4.2 sobre o desempenho da versão tradicional e EasyGrid AMS no cenário 2, com cargas externas

Tabela 4.2: Principais resultados de desempenho das abordagens tradicional e EasyGrid AMS para a aplicação *Thermions* no cenário 2.

	Versão tradicional				Versão EasyGrid AMS			
P	Média	I.C.	*Speed-up*	Eficiência	Média	I.C.	*Speed-up*	Eficiência
16	3518,78	0,16	11,74	98%	3229,49	0,27	12,79	107%
32	1775,91	0,21	23,26	97%	1622,81	0,04	25,46	106%
48	1185,93	0,15	34,83	97%	1091,19	0,09	37,86	105%
64	895,60	0,18	46,13	96%	823,42	0,10	50,17	105%
80	797,05	0,13	51,83	86%	652,83	0,97	63,28	105%
96	756,82	0,14	54,59	76%	558,49	0,35	73,97	103%
112	718,25	0,13	57,52	68%	491,58	0,25	84,04	100%
128	692,16	0,46	59,68	62%	424,97	0,31	97,21	101%



(a) Tempo de execução em segundos.

(b) Eficiência.

Figura 4.2: Resultados no cenário 2 para a aplicação *Thermions*.

estáticas. No primeiro gráfico (Figura 4.2(a)), as médias de tempo de execução são plotadas semelhantemente ao gráfico na Figura 4.1(a). A curva com os pontos em forma de quadrado representam os resultados para a versão tradicional e com os pontos em forma de círculo representam os resultados para a versão EasyGrid AMS. O segundo gráfico (Figura 4.2(b)) mostra a eficiência obtida normalizada de acordo com o poder computacional disponível para a aplicação. Como as cargas extras detêm 50% do uso de CPU de metade dos processadores, isto indica que as tarefas da aplicação têm na média 75% da capacidade das CPUs (100% de $\frac{P}{2}$ processadores e 50% dos outros $\frac{P}{2}$ processadores). Logo, a eficiência é calculada da seguinte forma: $\frac{speed-up}{P*0,75}$.

Pela Figura 4.2(a), pode-se perceber que o comportamento das curvas são semelhantes aos resultados do cenário 1. A versão MPI tradicional possui um tempo de execução sempre acima do tempo de execução da versão EasyGrid AMS e a diferença entre os tempos vai aumentando conforme o número de processadores cresce. Em relação à eficiência, pelo gráfico da Figura 4.2(b), a versão paralela do algoritmo *thermions* com o EasyGrid AMS

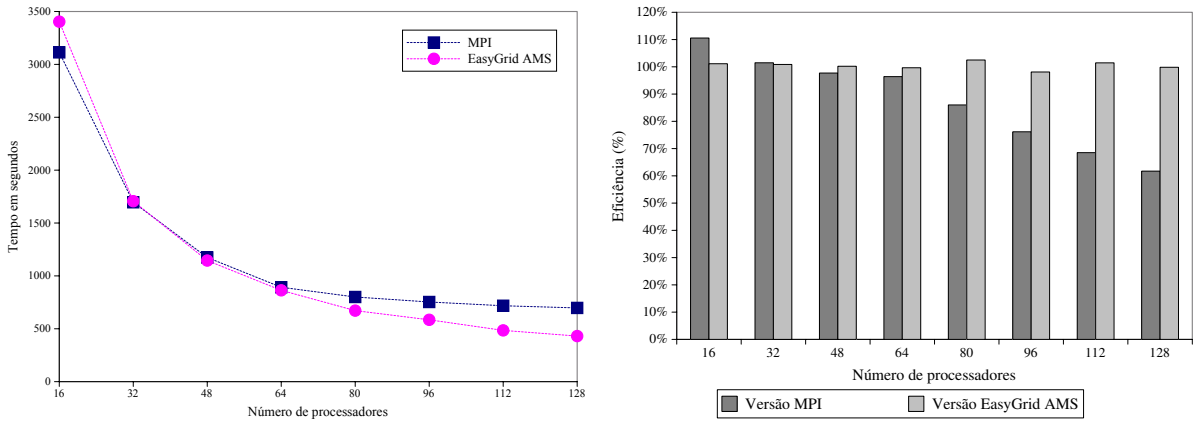
apresentou uma ótima eficiência próxima a 100% conforme o aumento de P e, portanto, mostrou-se bem escalável. Em contrapartida, a versão tradicional apresentou uma boa eficiência com $P \leq 64$ (o valor foi cerca de 97%) e, a partir do uso de 80 processadores, ocorre seu decaimento, chegando a 62% com 128 CPU. Como existe uma falta de precisão em relação ao consumo de CPU pelas cargas extras (elas migram de CPU a CPU em cada máquina multinúcleo), não necessariamente a aplicação possui exatamente 75% da capacidade das máquinas e isto pode explicar os valores maiores que 100%. No entanto, não ficou claro se o *speed-up* super linear aconteceu porque a proposta fez bom uso dos recursos (aproveitamento de *cache*, por exemplo) ou se foi por esta falta de precisão.

No cenário 3, a Tabela 4.3 e os gráficos da Figura 4.3 mostram os resultados de desempenho para a versão tradicional e EasyGrid AMS, da mesma maneira que no cenário 1 e 2. Na Figura 4.3(a), onde os tempos de execução são plotados para cada quantidade de processadores, pode-se notar que, para $P = 16$, a média do tempo de execução da versão tradicional foi melhor em relação à versão EasyGrid AMS e, para $P = 32$, são semelhantes. A partir de $P = 48$, os resultados se invertem e o EasyGrid AMS passa a ter o tempo de execução menor. Pelo gráfico da Figura 4.3(b), pode-se entender melhor o que acontece. A eficiência da versão EasyGrid AMS se mantém em cerca de 100% (normalizado da mesma forma do cenário 2) para as execuções com cada P processador, o que mostra a boa escalabilidade da proposta em um cenário com cargas dinâmicas. Novamente, os *speed-ups* super lineares costumam aparecer pois a normalização realizada sobre os valores da eficiência não é precisa (75% pode não ser a eficiência esperada), mas serve como valor comparativo.

Tabela 4.3: Principais resultados de desempenho das abordagens tradicional e EasyGrid AMS para a aplicação *Thermions* no cenário 3.

	Versão tradicional				Versão EasyGrid AMS			
P	Média	I.C.	*Speed-up*	Eficiência	Média	I.C.	*Speed-up*	Eficiência
16	3113,76	0,66	13,27	111%	3404,02	0,13	12,14	101%
32	1696,16	0,43	24,36	101%	1706,52	0,07	24,21	101%
48	1174,38	0,53	35,18	98%	1145,19	0,08	36,07	100%
64	892,74	0,07	46,27	96%	863,62	0,05	47,84	100%
80	800,49	0,39	51,61	86%	671,74	1,26	61,50	102%
96	753,28	0,13	54,84	76%	584,88	0,56	70,63	98%
112	717,81	0,14	57,55	69%	484,76	0,13	85,22	101%
128	697,10	0,37	59,26	62%	431,06	0,50	95,84	100%

A baixa eficiência e escalabilidade da versão tradicional é explicada pelo seu gerenciamento de tarefas centralizado e sob demanda. Especialmente a escalabilidade é prejudicada pelo gerenciamento centralizado, já que com o aumento de processadores, aumenta-se



(a) Tempo de execução em segundos. (b) Eficiência.

Figura 4.3: Resultados no cenário 3 para a aplicação *Thermions*.

o número de requisições ao processo mestre. A baixa eficiência, mesmo quando o número de processadores é menor, é atribuída ao fato do gerenciamento ser sob demanda. Tarefas são associadas a processadores apenas quando outras terminam e isto causa um atraso na execução delas. No caso da implementação EasyGrid AMS, a aplicação obtém melhor desempenho devido ao uso do modelo de particionamento 1Ptask e do *middleware* AMS, especialmente por seu gerenciamento hierárquico e escalonamento dinâmico proativo.

4.5 Resumo

Simulações Monte Carlo são bastante usadas em processos estocásticos úteis para simular problemas de diversas áreas como engenharia, biologia e finanças. Por serem massivamente paralelos e, portanto, não terem comunicação entre as partições, tais simulações são paralelizados de modo que cada partição seja independente uma da outra.

A versão paralela comumente empregada (chamada de versão MPI tradicional) para as simulações Monte Carlo indica o uso de um processo mestre e processos trabalhadores. O mestre divide o trabalho em partições iguais e as entrega, sob demanda, para cada trabalhador. É uma solução simples de ser implementada, mas apresenta suas deficiências como ter uma entidade centralizadora: o processo mestre. Utilizando a mesma solução mestre-trabalhadora mas modificando-a para usar o modelo 1Ptask com o EasyGrid AMS, é possível obter uma melhor eficiência para a aplicação.

Uma avaliação experimental foi conduzida utilizando uma aplicação real da *e*-ciência chamada *thermions*. *Thermions* implementa um método numérico que usa o Monte Carlo para simular a dispersão térmica em meios porosos periódicos. Três tipos de cenários

foram avaliadas: um dedicado, um heterogêneo (simulado através de introdução de cargas externas que usam intensivamente CPU em alguns processadores) e um heterogêneo e dinâmico (as cargas mudam constantemente entre os processadores).

Os resultados mostraram uma boa eficiência (acima de 90%) para as execuções com o EasyGrid AMS em um ambiente dedicado e uma boa escalabilidade. Neste mesmo cenário, a versão tradicional apresentou uma eficiência inferior em relação ao EasyGrid AMS e uma escalabilidade debilitada. Nos outros cenários, aqueles não dedicados, os resultados de desempenho foram similares. Isto é, a versão EasyGrid AMS obteve bom desempenho usando todos os processadores testados e uma boa escalabilidade enquanto houve uma diminuição da eficiência na versão tradicional conforme o aumento do número de processadores. O escalonamento proativo e hierárquico do EasyGrid AMS supera, em termos de desempenho, o gerenciamento simples da versão MPI comumente adotado.

Capítulo 5

Busca em Árvore Branch-and-prune – PDGDM

Algoritmos de busca em árvore do tipo *branch-and-prune* são muitas vezes usados para ajudar a resolver problemas de busca exaustiva de forma exata, como é o caso do problema de geometria das moléculas. Estes problemas estão concentrados em áreas científicas importantes como biologia, física, bioquímica, logística e geofísica. Geralmente, um algoritmo *branch-and-prune* consiste em uma busca em árvore com *backtracking* e informações sobre regiões de poda (satisfação de restrição). É muito semelhante aos algoritmos *branch-and-bound* cuja a etapa de poda é calculado por limites de uma função objetiva de otimização. A poda é uma fase significativa destes algoritmos porque proporciona a redução do espaço de busca árvore, eliminando alguns ramos e, consequentemente, a diminuição do tempo de busca. No entanto, isto provoca um desequilíbrio na árvore de busca em relação a uma árvore completa (sem poda).

Devido aos algoritmos de *branch-and-prune* terem uma complexidade de tempo exponencial, no pior caso, o processo de paralelização é requerido. Sistemas distribuídos como *clusters*, grades e nuvens de computadores, por exemplo, provêm uma infraestrutura capaz de executar instâncias do algoritmo em paralelo e de ter o benefício de tornar disponível uma grande quantidade de recursos computacionais. Porém, não é trivial lidar com o gerenciamento destes sistemas. Geralmente, recursos de computadores são heterogêneos, com diferentes quantidades e tipos de processadores e núcleos, diferentes memórias e hierarquia de memória e diferentes bandas de rede entre os nós. Eles também são compartilhados, isto é, vários usuários podem estar usando, e são suscetíveis a falhas.

Reconhecendo os benefícios da paralelização de algoritmos *branch-and-prune*, encontramos basicamente dois problemas. Primeiramente, recursos que compõem a infraestru-

tura devem ser gerenciados para que torne a execução da aplicação no ambiente possível e adequada. Alocação de recursos, comunicação entre processos, detecção e tratamento de falhas, balanceamento de carga, configuração do sistema, tudo deve ser gerenciado. Por segundo, a árvore de buscar desbalanceada não permite uma distribuição da árvore homogênea entre os processos. Isto causa processos com granularidade mais larga que outras e fazem o tempo total de execução ser limitado pela maior delas. Em outras palavras, não se conhece *a priori* a quantidade de trabalho de cada tarefa, dado que a poda pode acontecer sem previsão.

5.1 Algoritmos *Branch-and-Prune*

Várias áreas de estudo e pesquisa usam algoritmos exatos para resolver seus problemas. Algoritmos que exploram buscar em árvore podem exaustivamente determinar a solução ótima, no caso de problemas de otimização, ou um conjunto de soluções, no caso de problemas de satisfação de restrições (cujas soluções são definidas por restrições do problema. O termo *branch-and-prune* é usado aqui para referir a algoritmos de busca em árvore com ramificação e *backtracking* (retrocesso em árvore) causado por restrições não satisfeitas (isto é, a etapa de poda) [18]. Tais algoritmos apenas buscam por soluções sem considerar limites de otimização, como fazem os algoritmos de *branch-and-bound*[1]. Embora estes algoritmos gastem muito tempo para executar, eles são preferidos e vários esforços são realizados de modo a acelerar o processo de poda [122, 78] uma vez que quanto mais cedo a poda é feita, menos trabalho é computado.

5.1.1 Exemplo Sequencial

Um exemplo simples e com recursividade simulada através de pilha de um algoritmo sequencial *branch-and-prune* é apresentado no Algoritmo 5. Seja *Pilha* uma pilha de nós da árvore de busca. A Linha 1 e 2 mostram a inicialização da solução com 0 e a pilha *Pilha* com o nó raiz. Na linha 3 uma repetição é iniciada até que a pilha torne-se vazia. Dentro desta repetição, que simula a recursão, um nó é retirado da pilha na linha 4. Depois, se o nível do nó na árvore for o último, uma solução foi encontrada. Portanto, a variável *solucao* é incrementada na linha 6.

Se o nível do nó corrente não é o último da árvore, nós filhos são calculados e inseridos

[1] Algoritmos *branch-and-bound* são usados para encontrar soluções ótimas para vários problemas de otimização. Todos os candidatos a solução são enumerados através de uma estrutura de árvore e ramos infrutíferos são descartados de acordo com os limites superior e inferior da variável otimizada.

Algoritmo 5: Algoritmo sequencial *branch-and-prune*.

Entrada: N - número de níveis da árvore
Saída: *solucao* - contagem de soluções encontradas

1 $solucao \leftarrow 0$
2 $push(Pilha, no_{raiz})$
3 **enquanto** $Pilha \neq \varnothing$ **faça**
4 $\quad no \leftarrow pop(Pilha)$
5 $\quad$ **se** $no.nivel = N$ **então**
6 $\quad\quad solucao \leftarrow solucao + 1$
7 $\quad$ **senão**
8 $\quad\quad$ Calcular os filhos de *no* e inserir no conjunto C
9 $\quad\quad$ **para cada** $no_c \in C$ **faça**
10 $\quad\quad\quad$ **se** no_c ***não*** *é poda* **então**
11 $\quad\quad\quad\quad push(Pilha, no_c)$

no conjunto C (linha 8). Para cada nó filho em C, a fase de poda é calculada na linha 10 e a etapa de *backtracking* acontece aqui, caso a poda seja afirmativa. No caso da não existência da poda, o novo nó no próximo nível da árvore é inserido na pilha. A linha 11 representa a chamada recursiva de forma simulada.

5.1.2 Soluções Paralelas

Com o objetivo de paralelizar um algoritmo *branch-and-prune*, existem dois pontos que deve ser analisados: a partição do problema e o método de balanceamento de carga. A ideia por trás do particionamento da árvore de busca consiste em separar os ramos da árvore e distribuir estas sub-árvores entre os processadores. A Figura 5.1 representa um exemplo de um particionamento de uma árvore de buscar com 4 partições T_0, T_1, T_2 e T_3. As abordagens tradicionais fazem a distribuição das sub-árvores sob demanda. Isto é, conforme os processadores vão se tornando ociosos, novas sub-árvores são oferecidas a eles de modo a manter todos os processadores trabalhando. Cada processador usa um algoritmo sequencial similar ao Algoritmo 5 para computar as sub-árvores.

Uma vez feito o particionamento e a distribuição das sub-árvores, deve existir um mecanismo para remover ramos das sub-árvores e enviá-los para processadores ociosos. Este mecanismo é o balanceamento de carga. O método mais comum usado é o *work-stealing*. Este método indica que os processadores, quando se tornar ociosos, devem roubar tarefas de algum processo vítima randomicamente escolhido. Em outras palavras, ramos das sub-árvores, que representam novas sub-árvores que ainda não foram calculadas, são

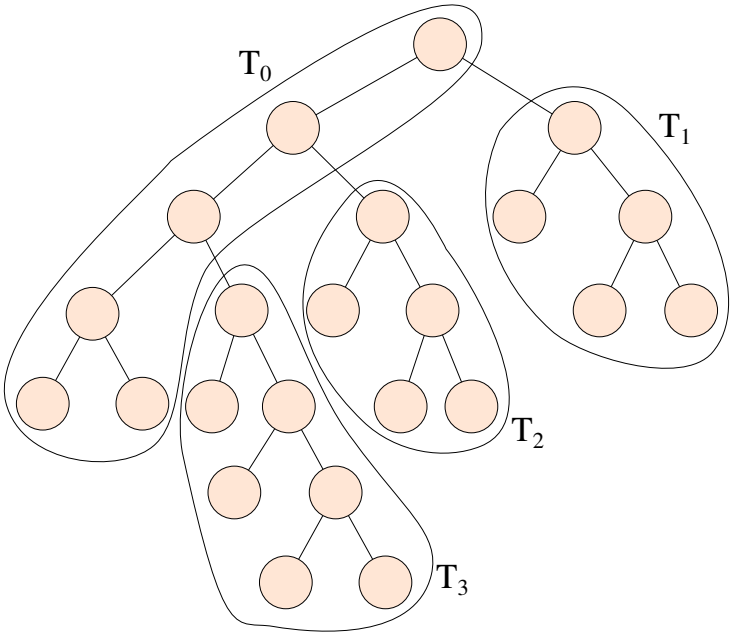


Figura 5.1: Exemplo de particionamento tradicional de uma árvore de busca.

removidos de um processador ocupado e alocados a um processador ocioso.

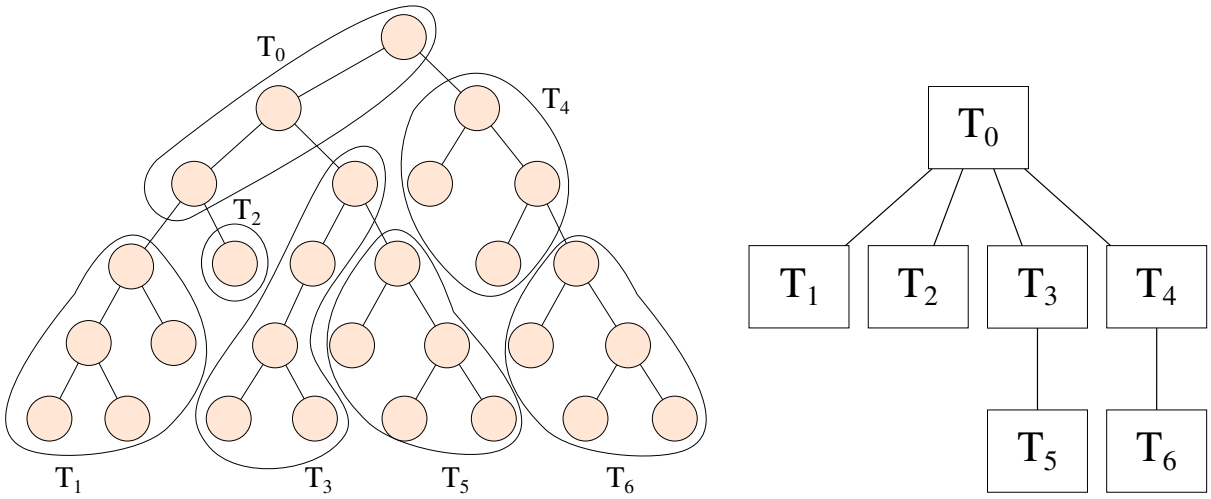
A abordagem tradicional particiona o problema sob demanda e usa um padrão mestre-trabalhador para gerenciar o balanceamento de carga. No entanto, este método não é aconselhável para sistemas distribuídos de larga escala já que o gargalo de mensagens do mestre causa baixa eficiência. Embora existam implementações distribuídas na literatura [35] para este tipo de problema mais sofisticadas, este padrão mestre-trabalhador é utilizado por ser mais simples de implementar.

Algoritmos paralelos *branch-and-bound* tem um comportamento diferente comparado ao *branch-and-prune*, apesar de serem semelhantes na estrutura. Enquanto algoritmos *branch-and-prune* geralmente buscam por todas as soluções do problema, algoritmos *branch-and-bound* procuram por uma solução ótima. Por esta razão, algoritmos *branch-and-bound* paralelos podem obter *speed-ups* super-lineares através da divisão de trabalho e compartilhamento da solução corrente [54]. Isto faz com que os processos antecipem uma melhor solução e realizem poda mais rapidamente. Por outro lado, os algoritmos *branch-and-prune* paralelos sempre processarão a mesma quantidade de trabalho comparado à versão sequencial e dificilmente obterá um *speed-up* acima do linear [18].

5.2 *Branch-and-prune* Autônomo

Como descrito no Capítulo 3, o algoritmo *branch-and-prune* apresenta decomposição exploratória, comunicação dependente e granularidade desconhecida. A decomposição é exploratória por se tratar de um algoritmo de busca em árvore e a comunicação é dependente gerando um DAG entre as tarefas. A granularidade desconhecida é devido ao método de poda que consiste na remoção de parte dos ramos da árvore devido a não satisfação das restrições do problema da aplicação.

O algoritmo *branch-and-prune* autônomo é composto por um método de autoconfiguração durante o particionamento da aplicação e pelo uso do *middleware* EasyGrid AMS. Neste caso, tal algoritmo paralelo da aplicação difere do algoritmo paralelo tradicional no modo como é feito o particionamento. O particionamento da árvore de busca pelo algoritmo tradicional é realizado sob demanda, sempre que um processador torna-se ocioso e o escalonador básico que faz o balanceamento de carga é acionado. O algoritmo *branch-and-prune* autônomo executa dinamicamente o ajuste da granularidade e número de tarefas de acordo com informações do sistema captadas pelo EasyGrid AMS (autoconfiguração) e mantém o algoritmo paralelo no modelo 1Ptask. Assim, o *middleware* gerencia as tarefas usando seus escalonadores dinâmicos e métodos de tratamento e recuperação de falhas.



(a) Árvore *branch-and-prune*. (b) Grafo de dependências de tarefas.

Figura 5.2: Exemplo de particionamento de uma árvore de busca usando o modelo 1Ptask.

O algoritmo paralelo autônomo da aplicação *branch-and-prune* faz o cálculo recursivo em sub-árvores de busca distribuídas inicialmente por um processo mestre. Este processo mestre também é responsável por receber as soluções encontradas pelas tarefas trabalhadoras. A busca procede em uma pilha que guarda os valores de nós da árvore já

calculados. Porém, é introduzido um tempo máximo de execução da busca determinado pelo valor MAX_TIME. Este valor delimita o tamanho da tarefa e pode ser definido por 5 a 10 segundos, representando a granularidade fina. Deste modo, toda vez que a busca estourar este tempo, a tarefa guarda a pilha de busca em memória e a execução do cálculo do problema termina. O término do trabalho da tarefa também pode ocorrer por pilha vazia, o que a faz ter um granularidade menor do que o esperado. Ao chegar ao fim de sua execução, a tarefa é responsável por criar novas tarefas particionando a pilha guardada entre elas. Esta geração de novas tarefas considera informações do ambiente de execução e da própria aplicação (autoconfiguração) e cada nova tarefa tem dependência de seu pai, gerando um GAD dinâmico. A Figura 5.2 apresenta um exemplo da árvore particionada e as dependências entre tarefas. A Figura 5.2(a) mostra a árvore particionada em $T_0, \ldots,$ T_5 e T_6. Estas tarefas apresentam dependências entre si. T_0 (pai) cria T_1, T_2, T_3 e T_4 (filhos); T_4 cria T_5 (T_4 é o pai de T_5); T_3 cria T_6 (T_3 é o pai de T_6).

O Algoritmo 6 descreve melhor o algoritmo *branch-and-prune* autônomo e ele é usado por todas as tarefas trabalhadoras. Uma tarefa mestre inicia a execução criando as tarefas in e envia um conjunto com o nó raiz. Esta tarefa mestre também é responsável por receber as soluções. Diferentemente do algoritmo paralelo tradicional, o mestre não faz intermédio das tarefas migrantes e não realiza nenhuma operação de balanceamento e gerenciamento. O responsável por lidar com o gerenciamento e balanceamento das tarefas é o EasyGrid AMS.

Assim, depois da inicialização de algumas variáveis na linha 1 e 2, cada tarefa espera receber o conjunto M de nós do pai e o insere na pilha (linhas 3, 4 e 5). No início, o processo mestre é o pai (linhas 3). Em seguida, a mesma repetição realizada pelo algoritmo tradicional é inicializada compreendendo as linhas 6 até 17. A partir da linha 18, é realizada a etapa de finalização da tarefa e não está relacionada com o algoritmo do problema da aplicação. As linhas 18 e 19 indicam o ponto onde ocorre a autoconfiguração da aplicação. O valor p, que representa o número de novas tarefas a serem criadas, é determinado de acordo com informações de carga do sistema fornecidas pelo EasyGrid AMS (autoconfiguração). O conjunto S é então particionado entre p subconjuntos S_i e eles são enviados (linha 22) para as novas tarefas criadas na linha 21. A última linha procede o envio da solução encontrada pela tarefa corrente para a tarefa mestre.

Combinado ao *middleware* EasyGrid AMS, o algoritmo paralelo descrito em 6 possui a capacidade de autoconfigurar-se e também adquire um sofisticado meio de auto-otimização e autorrecuperação. A granularidade das tarefas é fina (por exemplo, indicando os valores

Algoritmo 6: Algoritmo *branch-and-prune* paralelo e autônomo.

Entrada: N - número de níveis na árvore.
MAX_TIME - tempo máximo de execução de uma tarefa.

1 $solucao \leftarrow 0$
2 $S \leftarrow \varnothing$
3 Receber M do nó pai
4 **para cada** $no \in M$ **faça**
5 $\quad push(Pilha, no)$
6 **enquanto** $Pilha \neq \varnothing$ **faça**
7 $\quad no \leftarrow pop(Pilha)$
8 $\quad$ **se** $no.nivel = N$ **então**
9 $\quad\quad solucao \leftarrow solucao + 1$
10 $\quad$ **senão**
11 $\quad\quad$ Calcular os filhos de no e inserir no conjunto C
12 $\quad\quad$ **para cada** $no_c \in C$ **faça**
13 $\quad\quad\quad$ **se** no_c ***não*** *é poda* **então**
14 $\quad\quad\quad\quad push(Pilha, no_c)$
15 $\quad$ **se** *tempo de execução* $>$ *MAX_TIME* **então**
16 $\quad\quad$ Guardar todos os nós de $Pilha$ em S
17 $\quad\quad Pilha \leftarrow \varnothing$
18 Determinar p
19 Particionar S em p subconjuntos e considerar S_i como cada subconjunto
20 **para** $i \leftarrow 0$ *a* p **faça**
21 $\quad$ Criar nova tarefa i
22 $\quad$ Enviar S_i para a tarefa filha i
23 Enviar *solucao* para o processo mestre

5 ou 10 segundos para MAX_TIME) e o número de tarefas é ajustado para que sempre existam novas tarefas disponíveis em cada máquina.Ainda, o algoritmo proposto encontra-se também no padrão mestre-trabalhador, assim como no algoritmo paralelo tradicional. A diferença entre os dois algoritmos está apenas na introdução da autoconfiguração representadas pelas linhas 15, 16 e 17 (reconfiguração da granularidade) e linhas de 18 a 23 (reconfiguração do número de tarefas). No entanto, a estrutura da aplicação sobrepõe a do EasyGrid AMS que é hierárquica. Logo, a aplicação herda a estrutura hierárquica de gerenciamento do *middleware*, eliminando o problema do gargalo.

5.3 Caso de Estudo: o Problema Discreto de Geometria de Moléculas

O problema de encontrar a estrutura tridimensional de moléculas de proteína é conhecido como PGDM – Problema da Geometria da Distância Molecular, uma aplicação da área de bioquímica. O objetivo do problema é encontrar todas as possíveis estruturas tridimensionais da cadeia principal ou *backbone* de uma molécula de proteína, dadas informações limitadas sobre distâncias atômicas disponibilizadas pela Ressonância Nuclear Magnética (RNM) [78]. A estrutura tridimensional é muito importante porque propriedades físicas e químicas da molécula estão associadas a ela.

A definição do PGDM é: dado um grafo ponderado e não direcionado $G = (V, E, d)$, com $d : E \to \mathbb{R}_+$, deseja-se encontrar $x : V \to \mathbb{R}^3$ de tal modo que $\|x_u - x_v\| = d_{uv}$ ("$\|$" representa a distância euclidiana) para cada $\{u, v\} \in E$. O conjunto V representa os átomos, o conjunto E são os pares de átomos $\{u, v\}$ cuja a distância d_{uv} é conhecida e x representa a posição tridimensional de um átomo. Este problema é NP-Completo [77].

A versão discreta do PGDM é conhecida como PDGDM – Problema Discreto da Geometria da Distância Molecular. Esta versão resolve o problema PGDM considerando uma ordem $v_1, ..., v_n$ de V satisfazendo os seguintes requisitos:

1. E contém todas as cliques em quádruplas de vértices consecutivos, *i. e.*, $S \subset E$ onde $S = \{\{j, k\} \in E \ \forall i \in \{4, ..., n\} \ \forall j, k \in \{i - 3, ..., i\}\}$;
2. os ângulos de ligação $\theta_{i-2,i}$ entre átomos v_{i-2}, v_{i-1} e v_i não são múltiplos de π, para $i \in \{3, ..., n\}$.

Em outras palavras, considere qualquer sequência de átomos consecutivos a, b, c, d na cadeia principal da proteína como na Figura 5.3. No processo de discretização, duas hipóteses são assumidas: existe um conhecimento *a priori* das distâncias entre 4 átomos consecutivos representados na figura por x_{ab}, x_{ac}, x_{ad}, x_{bc}, x_{bd} e x_{cd}; e o ângulo de torção entre os vetores $\overrightarrow{a, b}$ e $\overrightarrow{b, c}$, representado na figura por ω, não é múltiplo de π.

As duas hipóteses são aplicáveis a várias proteínas. NMR é capaz de computar distâncias de átomos que estão próximos entre si, e átomos em grupos de quatro consecutivos na sequência do *backbone* são geralmente mais próximos do valor limite 6Å[2]. Além disso, não são conhecidas proteínas com ângulos múltiplos de π. Ainda, a sequência de átomos pode

[2]O valor de 1Å (angströn) equivale a 10^{-10}m (metro).

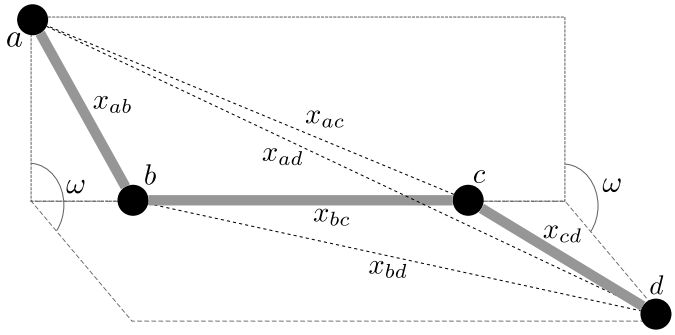


Figura 5.3: Exemplo de sequência de átomos consecutivos no *backbone* de uma proteína.

ser encontrada usando um algoritmo de complexidade polinomial. PDGDM é também NP-completo [78].

Um algoritmo *branch-and-prune* pode ser usado para resolver este problema. A fase de ramificação ocorre por causa de dois possíveis valores do ângulo de torção (o ângulo ω na Figura 5.3): positivo e negativo. Um ramo representa um possível valor do ângulo. Logo, a árvore é binária. No nível igual ao comprimento da árvore, dado pelo número de átomos na cadeia principal da molécula, soluções são encontradas. Uma solução é representada por uma sequência de átomos com suas coordenadas tridimensionais. Um conjunto $F = \{E - S\}$ contém os pares de vértices que não são cliques em quádruplas de vértices consecutivos (isto é, contém os outros pares de vértices não-cliques). Este conjunto permite a fase de poda através da exclusão de ramos cuja a distância, representada por um par de F, é inviável.

5.4 Avaliação Experimental

Os experimentos foram executados, assim como no Capítulo 4, no *cluster* Oscar na Universidade Federal Fluminense. Foram utilizadas, então, 16 máquinas de 2 processadores de quatro núcleos, cada, totalizando 128 CPUs.

Os testes foram conduzidos com 4 instâncias reais disponíveis no banco de dados PDB [100] - *Protein Data Bank*. A Tabela 5.1 refere-se a cada instância chamada de 1AYK, 1BMW, 2ADN e 2AIV. A segunda coluna apresenta o número de soluções encontradas por cada instância que são as mesmas para o algoritmo paralelo. Colunas 3 e 4 representam o número de vértices ou átomos da cadeia principal da moléculas e o número de arestas extra F. As duas últimas colunas mostram a média de tempo do algoritmo sequencial com 10 rodadas e o intervalo de confiança de 96%.

Tabela 5.1: Informação sobre as instâncias PDGDM usadas nos experimentos.

Instância	Número de Soluções	$\|V\|$	$\|F\|$	Tempo Sequencial Média (s)	Intervalo de Confiança
1AYK	2560	507	1280	14651,02	0,43
1BMW	3840	282	620	735648,73	16,57
2ADN	720896	216	541	41635,48	0,46
2AIV	131072	447	1120	332831,97	5,60

Todas as instâncias tem um tempo longo de execução e 1BMW e 2AIV são os maiores com tempo de 8,5 e 4 dias, respectivamente, enquanto 1AYK e 2ADN demoram cerca de 4 e 11 horas, cada. As soluções, representada pelas estruturas tridimensionais possíveis da cadeia principal da proteína, são geradas, mas não apresentadas aqui.

Três versões, uma sequencial e duas paralelas, do algoritmo *branch-and-prune* para o PDGDM são comparadas nesta seção. Todos os algoritmos foram implementados com as mesmas estruturas de dados. O *algoritmo sequencial*, usado para gerar os resultados na Tabela 5.1 e calcular os *speed-ups*, foi implementado em [78] e é similar ao Algoritmo 5. O *algoritmo paralelo tradicional* foi implemendado em MPI e ele considera uma abordagem mestre-trabalhadora *work-stealing* como descrito na Seção 5.1.2. A terceira versão é o método proposto neste capítulo: o *algoritmo autônomo com o EasyGrid AMS*, que será chamado de *versão EasyGrid AMS*. Embora o mecanismo de tolerância a falhas não tenha sido avaliado neste trabalho, o método verificação de falhas encontrava-se ativado durante os experimentos. Com isso, é possível avaliar a sobrecarga do método.

O objetivo dos experimentos é comparar a execução do algoritmo paralelo tradicional e com o EasyGrid AMS. Com um ambiente dedicado e outro com cargas externas em relação a aplicação, é possível avaliar o comportamento dos dois algoritmos considerando características de heterogeneidade e compartilhamento de recursos dinâmico.

5.4.1 Descrição dos Cenários Paralelos

Os cenários para avaliar a proposta são listados a seguir:

Cenário 1 Todas as máquinas são dedicadas.

Cenário 2 Metade de 12 máquinas possui cargas externas estáticas.

Cenário 3 Metade de 12 máquinas possui cargas externas dinâmicas.

No cenário 1, a variação do número de máquinas é 2, 4, 6, 8, 10, 12, 14, e 16, lembrando que cada máquina possui 8 núcleos no total. Todas as máquinas estão dedicadas para avaliar os *speed-ups* e a eficiência da proposta usando cada instância. O segundo e o terceiro cenário tem o número de máquinas fixado em 12 e estes experimentos simulam um ambiente heterogêneo, compartilhado e dinâmico. Uma *carga externa* é um processo que faz uso intensivo de CPU e executa nas máquinas concorrentemente com a aplicação. No cenário 2, 8 (todos os núcleos de processamento de uma máquina) cargas externas com uso intensivo de CPU são estaticamente alocadas em cada 6 máquinas e permanecem nelas até o final da aplicação. A mesma estratégia é conduzida no cenário 3, mas as cargas são dinâmicas. As 8 cargas rodam por 10 segundos em cada 6 máquinas e alternam sua execução nas outras máquinas. Então, cada 6 máquinas tem 8 cargas por 10 segundos, depois as outras 6 máquinas tem 8 cargas por 10 segundos e assim sucessivamente.

A relevância dos cenários 2 e 3 é a possibilidade de análise do desempenho das abordagens em um sistema compartilhado e dinâmico, como grade e nuvem de computadores. Neste caso, para o balanceamento de carga, não só o desbalanceamento da árvore é considerado, como também as mudanças de carga no sistema.

5.4.2 Resultados no Cenário 1

O objetivo do primeiro experimento com o cenário 1 é mostrar e analisar o desempenho do algoritmo *branch-and-prune* autônomo com o EasyGrid AMS comparado com a abordagem tradicional. A Tabela 5.2 apresenta a média de tempo de execução (de 10 rodadas) e seus intervalos de confiança (96%) para cada abordagem e variando o número de processadores P. Cada tabela contém os resultados de cada instância experimental. Considerando o intervalo de confiança, todas as médias de tempo de execução para a versão EasyGrid AMS foram menores em relação a versão tradicional.

Os quatro gráficos na Figura 5.4 mostram, para cada instância, a eficiência em relação à execução do algoritmo sequencial. Este valor foi calculado considerando a equação: $eficiencia = \frac{speedup}{P} \times 100\%$, onde $speedup = \frac{tempo\ sequencial}{tempo\ paralelo}$ é o *speed-up* da execução paralela em relação à sequencial. O eixo vertical do gráfico indica o valor da eficiência em porcentagem e o eixo horizontal indica o número de processadores (número total de núcleos de CPU). Em cada gráfico, a barra cinza clara representa a abordagem do EasyGrid AMS e a barra cinza escura representa a abordagem tradicional.

Geralmente, a eficiência varia de acordo com o tamanho da instância. 1AYK e 2ADN são as menores instâncias (em termos de tempo de execução sequencial) e possuem as

Tabela 5.2: Principais resultados de desempenho no cenário 1 para o PDGDM.

1BMW				
	Versão Tradicional		EasyGrid AMS	
P	Tempo Total	I. C.	Tempo Total	I. C.
16	53937,34	5,03	49866,33	4,00
32	27789,47	3,21	24144,23	2,66
48	18335,37	1,63	15957,42	0,95
64	13830,19	1,25	11889,54	0,72
80	11062,09	0,56	9502,51	0,22
96	9228,04	0,59	7927,93	0,40
112	7927,72	0,28	6813,17	0,11
128	6937,88	0,37	5982,82	1,14

2AIV				
	Versão Tradicional		EasyGrid AMS	
P	Tempo Total	I. C.	Tempo Total	I. C.
16	25609.79	2.07	22889.60	1.14
32	13310.98	0.77	11131.95	1.98
48	9052.96	0.47	7363.00	0.77
64	7001.42	0.33	5503.18	0.48
80	5697.64	0.38	4408.43	0.38
96	4897.96	0.38	3683.76	0.28
112	4259.94	0.55	3133.19	0.08
128	3785.09	0.42	2744.40	0.14

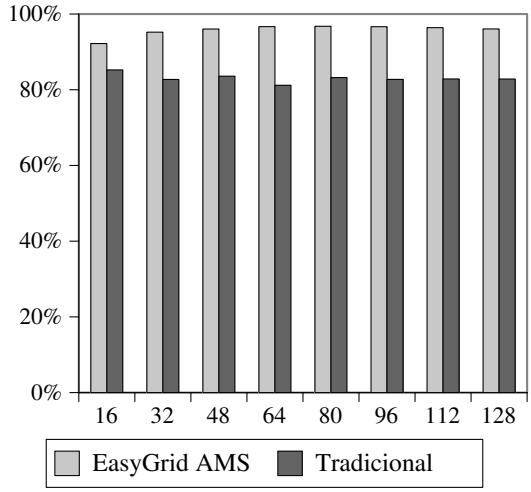
1AYK				
	Versão Tradicional		EasyGrid AMS	
P	Tempo Total	I. C.	Tempo Total	I. C.
16	1138.76	0.12	1024.33	0.12
32	591.62	0.04	512.87	0.06
48	403.90	0.05	352.67	0.06
64	310.77	0.04	270.32	0.04
80	256.00	0.04	228.64	0.03
96	218.43	0.03	195.93	0.05
112	193.80	0.02	177.04	0.06
128	174.53	0.03	159.28	0.04

2ADN				
	Versão Tradicional		EasyGrid AMS	
P	Tempo Total	I. C.	Tempo Total	I. C.
16	3027.83	0.30	2868.12	0.29
32	1534.02	0.18	1404.28	0.21
48	1031.58	0.11	942.80	0.14
64	780.43	0.07	714.76	0.12
80	632.58	0.06	584.47	0.14
96	532.23	0.04	497.50	0.15
112	463.62	0.03	421.89	0.06
128	407.69	0.03	373.85	0.07

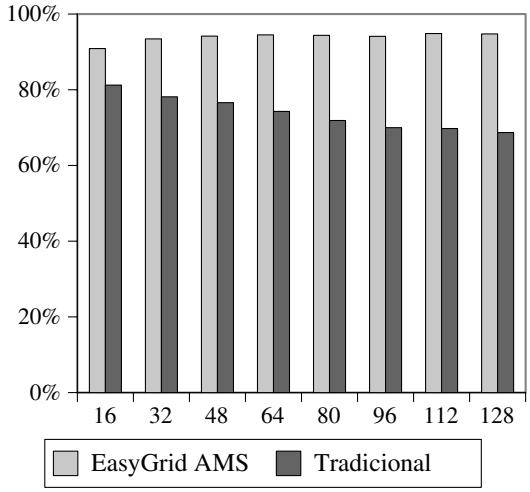
menores eficiências enquanto 1BMW e 2AIV são as instâncias maiores e tem uma eficiência maior. As instâncias pequenas apresentam um baixa quantidade de trabalho em relação às instâncias maiores e, conforme o número de processadores paralelos aumenta, a quantidade de trabalho para ser distribuída diminui.

Um comportamento relevante que pode ser visto nos gráficos da Figura 5.4 é a variação da eficiência de acordo com o aumento do número de processadores. Se a eficiência é constante ou aumenta, a proposta é escalável. Caso contrário, a proposta não escala. Para a instância 1AYK (Figura 5.4(c)), a eficiência varia de 66% a 80% usando a versão tradicional e de 72% a 90% usando a versão EasyGrid AMS. Estes valores diminuem conforme o número de processadores aumenta e isto mostra uma baixa escalabilidade para ambas as versões, embora a abordagem com o EasyGrid AMS é mais eficiente que a tradicional. Na Figura 5.4(d), com a instância 2ADN, a diminuição da eficiência em relação ao número de processadores é mais tênue comparada à instância 1AYK. Isto indica uma melhor escalabilidade especialmente para a versão EasyGrid AMS que tem um sequência constante de valores de eficiência (88%) desde 96 processadores. A execução com a versão tradicional tem uma variação do valor de eficiência entre 80% e 86% e para o EasyGrid AMS, tal variação encontra-se no intervalo entre 88% e 93%.

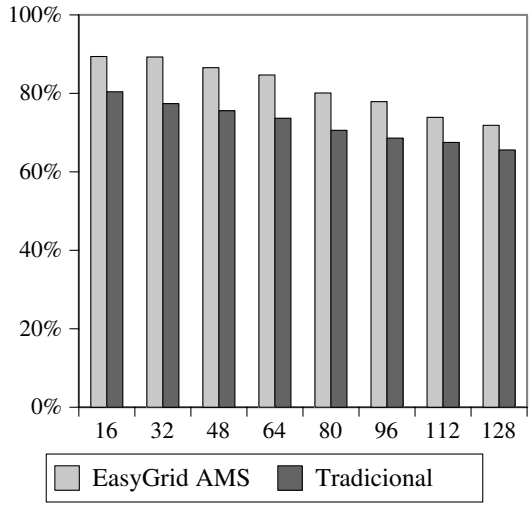
A Figura 5.4(b) e a Figura 5.4(a) mostram a comparação da eficiência entre o EasyGrid AMS e a versão tradicional no cenário dedicado para as instâncias maiores 2AIV e 1BMW. Com a instância 2AIV, a eficiência da abordagem tradicional diminui enquanto aumenta para a abordagem com o EasyGrid AMS. Em comparação, o EasyGrid AMS apresenta um



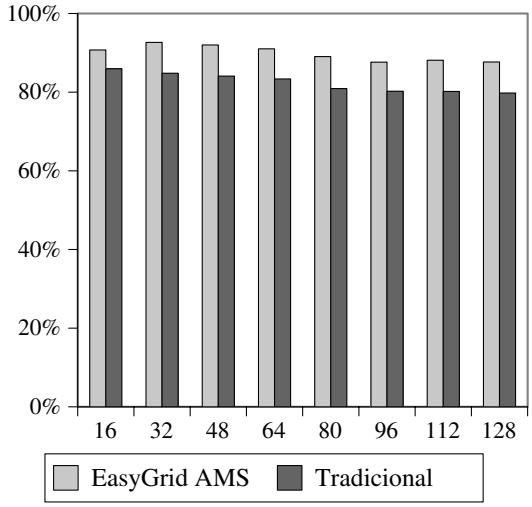
(a) 1BMW



(b) 2AIV



(c) 1AYK



(d) 2ADN

Figura 5.4: Comparação da eficiência entre as versões EasyGrid AMS e tradicional no cenário 1 para a aplicação PDGDM.

desempenho muito bom não apenas porque ele tem alta eficiência (com diferença de 26% para 128 processadores) mas também porque possui uma escalabilidade significantemente melhor. No caso da instância 1BMW, existe valor constante de eficiência a partir de 32 processadores. Para a versão EasyGrid AMS, ele valor é aproximadamente igual a 97% e para a versão tradicional, este valor é de 83% (diferença de 14%). Embora este último resultado indique uma boa escalabilidade para ambas as propostas, a eficiência do EasyGrid AMS é visivelmente melhor. A instância 1BMW é a maior dentre as avaliadas neste trabalho e possui um alto grau de paralelismo. Por esta razão, a escalabilidade é menos afetada.

A explicação dos resultados no cenário 1 é dada pela sobrecarga do escalonamento de

Tabela 5.3: Média do número de tarefas da aplicação PDGDM com o EasyGrid AMS.

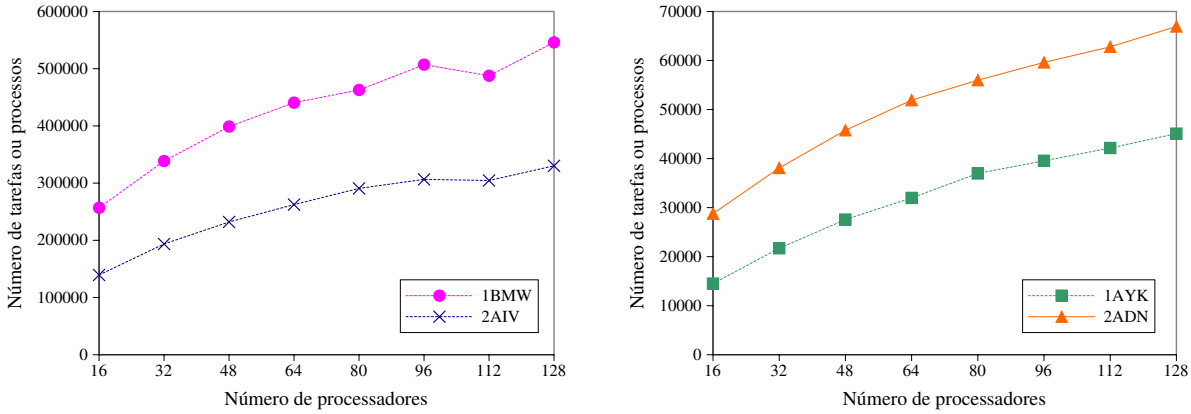
		16	32	48	64	80	96	112	128
1BMW	Tarefas	256940	338648	398869	440678	462823	507084	487722	545966
	I. C.	186	296	505	452	602	770	496	481
2AIV	Tarefas	139436	193617	232092	262500	290781	306525	304555	330256
	I. C.	271	285	292	264	244	134	141	151
1AYK	Tarefas	14519	21709	27567	31982	36972	39556	42165	45090
	I. C.	17	19	31	38	32	37	15	22
2ADN	Tarefas	28786	38087	45782	51912	55988	59617	62788	66939
	I. C.	24	19	29	51	33	49	57	11

tarefas e pela estratégia de particionamento. Na versão tradicional, o particionamento das tarefas é realizado sempre que uma máquina ociosa é detectada e tarefas de uma máquina ocupada são roubadas e alocadas para a máquina ociosa. Este é um escalonamento *work-stealing* reativo executado por um processo mestre-trabalhador centralizado. EasyGrid AMS usa um escalonamento proativo onde as tarefas são particionadas mesmo antes da detecção de desbalanceamento. A estrutura do escalonador é distribuído e hierárquico e, portanto, apresenta menos sobrecarga em relação a um escalonador mestre-trabalhador. Devido ao escalonador reativo da versão tradicional, sua eficiência é menor comparada ao escalonador do EasyGrid AMS, como pode ser visto no gráfico da Figura 5.4. A baixa escalabilidade da versão tradicional, demostrada pelo decréscimo da eficiência conforme o número de processadores aumenta nas Figuras 5.4(b), 5.6(c) e 5.6(d), é uma consequência do escalonador mestre-trabalhador. Um processo centralizado é responsável por simultaneamente gerenciar várias requisições de tarefas, causando um gargalo.

Os gráficos da Figura 5.5 e a Tabela 5.3 mostram a média da quantidade total de tarefas geradas em cada instância avaliada e variando-se o número de processadores (intervalo de confiança de 96%). O número de tarefas sempre aumenta conforme o aumento do número de processadores, exceto para as instâncias maiores 1BMW e 2AIV (ver Figura 5.5(a)) com 112 processadores, onde o valor é menor comparado ao uso de 96 processadores. De modo geral, isto mostra que o EasyGrid AMS consegue lidar com um grande número de tarefas ou processos durante a execução da aplicação.

5.4.3 Resultados nos Cenários 2 e 3

Os gráficos na Figura 5.6 representam os principais resultados obtidos com o cenário 2 e 3, comparando a eficiência entre as versões tradicional e EasyGrid AMS para cada instância (e também comparando com o cenário 1). Os valores da eficiência foram calculados da mesma maneira que no experimento com o cenário 1, porém eles foram normalizados.



(a) Instâncias 1BMW e 2AIV

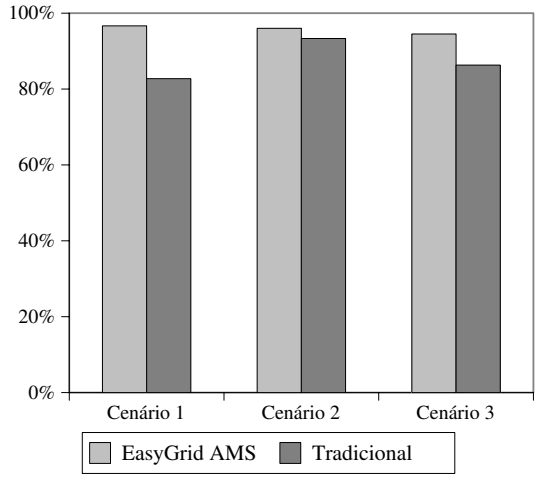
(b) Instâncias 1AYK e 2ADN

Figura 5.5: Quantidade de tarefas da aplicação PDGDM com o EasyGrid AMS.

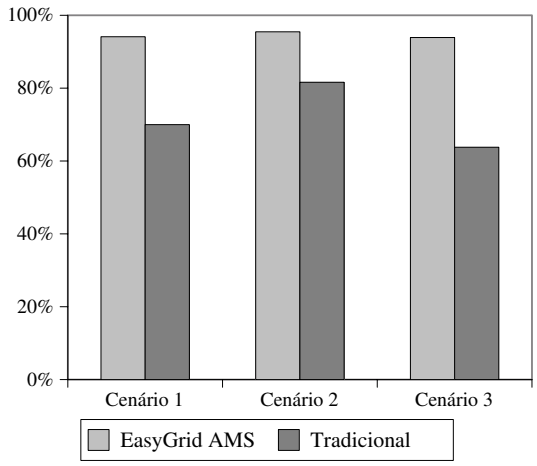
Como já mencionado, ambos os cenários 2 e 3 possuem metade das máquinas com 50% de carga externa. Logo, o valor esperado da eficiência é de 75%. No entanto, o valor mostrado nos gráficos da figura estão normalizados em relação ao cenário 1 com eficiência esperada de 100%. Este valor pode variar de acordo com as cargas e o escalonamento de processos dos sistemas operacionais de cada máquina.

Pela Figura 5.6, a eficiência da execução com o EasyGrid AMS é melhor que a execução com a versão tradicional em todos os cenários. As diferenças de eficiência entre eles alcançam cerca de 24%, 14% e 30% para cada respectivo cenário com a instância 2AIV (Figura 5.6(b)). O balanceador de carga *work-stealing*, que é centralizado e reativo às mudanças do sistema, tem um bom desempenho mas introduz atrasos nas decisões de escalonamento de tarefas. O escalonador dinâmico do EasyGrid AMS, que é hierárquico e proativo, tem um desempenho melhor não apenas em ambientes dedicados, mas também em sistemas heterogêneos, compartilhados e dinâmicos. A diferença de desempenho entre as propostas fica evidente no cenário 3 que simula um ambiente mais parecido com uma grade computacional. Além disso, as eficiências mantém-se relativamente constantes em todos os cenários com a versão EasyGrid AMS, especialmente para as instâncias 1BMW, 2ADN e 2AVI. Isto mostra a estabilidade e confiabilidade do escalonador dinâmico do EasyGrid AMS conforme as variações do ambiente.

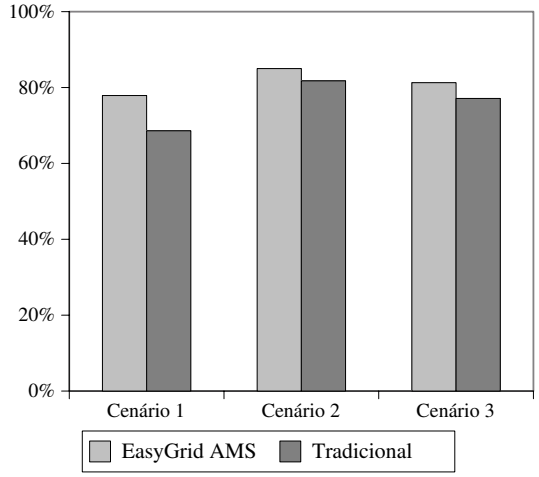
Embora o tempo de execução para todas as instâncias e versões (tradicional e EasyGrid AMS) sejam sempre maiores para para o cenário 2 e 3 em relação ao cenário 1, a eficiência tem um comportamento diferente (ver Figura 5.6). A eficiência do tempo de execução total em relação à quantidade de processadores, mesmo normalizada, é maior no cenário 2 comparado aos outros dois cenários. Comparado ao cenário 3, os resultados



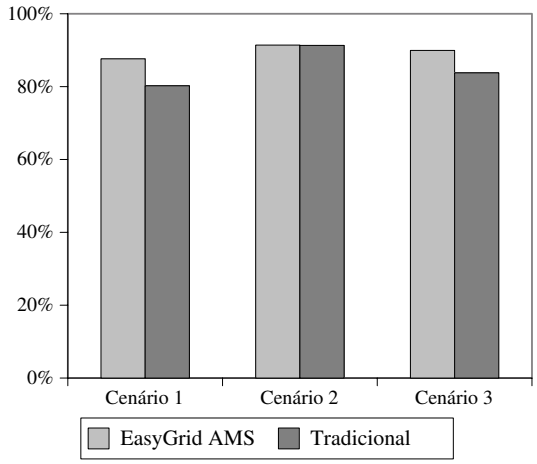
(a) 1BMW



(b) 2AIV



(c) 1AYK



(d) 2ADN

Figura 5.6: Comparação da eficiência normalizada entre as versões EasyGrid AMS e tradicional nos cenários 1, 2 e 3 para a aplicação PDGDM.

de eficiência mais baixa no cenário 2 são esperados já que a introdução de cargas externas dinâmicas atrasam o desempenho dos escalonadores *work-stealing* em do EasyGrid AMS. Em relação ao cenário 1, na maioria das vezes a diferença na eficiência entre os cenários 2 e 1 é maior especialmente para a execução com a versão tradicional. A carga externa é estática e, para os escalonadores, a introdução destas cargas é levemente similar à redução do número de máquinas disponíveis. Então, o escalonador lida com menos máquinas simultaneamente já que a sobrecarga nas máquinas atrasa o trabalho das tarefas e faz o escalonador associar mais tarefas para as máquinas mais rápidas.

5.5 Resumo

Este capítulo apresentou um algoritmo *branch-and-prune* autônomo eficiente também capaz de tirar proveito de um ambiente compartilhado, heterogêneo, dinâmico e distribuído de forma eficiente. A proposta indica o uso de recursos de auto-otimização, autorrecuperação e autoconfiguração fornecidos pela EasyGrid AMS e um algoritmo *branch-and-prune* autoconfigurável. Quando o algoritmo está ligado ao *middleware*, a aplicação herda características autônomas e é capaz de explorar os sistemas distribuídos.

Os experimentos foram realizados através de 3 cenários: 1) em máquinas dedicadas de um *cluster* com até 128 CPUs; 2) em um ambiente compartilhado de 96 CPUs com cargas externas estáticas e 3) em um ambiente compartilhado de 96 CPUs com cargas externas dinâmicas. Resultados no cenário 1 indicam que o algoritmo proposto (com o EasyGrid AMS) resolve o problema *branch-and-prune* com valores de eficiência nos intervalos de 91% a 97% para instâncias maiores e de 72% a 93% para instâncias menores. A versão tradicional alcança apenas os valores de eficiência nos intervalos de 69% a 85% para instâncias maiores e 66% a 86% para instâncias menores. A eficiência superior do EasyGrid AMS é devido ao seu particionamento autoconfigurável e ao escalonamento proativo e hierárquico, enquanto a versão tradicional usa um particionamento sob demanda e um escalonamento centralizado. Finalmente, resultados com os cenários 2 e 3 mostram novamente um bom desempenho para o EasyGrid AMS mesmo em ambientes distribuídos heterogêneos, compartilhados e dinâmicos. Nestes cenários, o *middleware* mantém a eficiência obtida em um ambiente dedicado. A versão tradicional com a abordagem *work-stealing* apresenta uma boa eficiência nos 3 cenários mas o EasyGrid AMS consegue ser representativamente melhor.

Algoritmos *branch-and-bound*, cujo objetivo é encontrar a melhor solução, podem ser implementados usando a mesma proposta deste capítulo. Basicamente, a única alteração seria na propagação do limite superior/inferior e usar este valor para acelerar a execução da fase de poda de ramos da árvore de busca. Em trabalhos futuros, pretende-se avaliar a implementação de um algoritmo *branch-and-bound* e analisar o desempenho considerando os possíveis *speed-ups* super lineares.

Capítulo 6

Decomposição de Domínio Fortemente Acoplado – N-corpos

Aplicações que possuem um domínio de dados fortemente acoplado requerem uma alta taxa de comunicação entre seus processos em paralelo. Essa classe de aplicações apresenta uma alta complexidade nos projetos paralelos e poucas propostas de solução existem em ambientes heterogêneos e dinâmicos [104]. Devido a esta dificuldade no desenvolvimento de soluções paralelas para aplicações de domínio fortemente acoplado, a proposta deste trabalho estuda e analisa o algoritmo *Ring*, representante desta classe, aplicado ao problema *N-corpos*.

Em relação ao particionamento, aplicações de decomposição de domínio particionam seus dados associados a um problema. Tais dados são divididos em pequenos pedaços de aproximadamente tamanhos iguais, se for possível. As operações são feitas sobre cada partição, sendo que elas geralmente requerem dados de outras. Sendo assim, a comunicação intensiva é necessária para mover informações entre tais partições [44] e por isto são chamadas de fortemente acopladas.

6.1 Caso de Estudo: o Problema N-corpos

O problema N-corpos ou *N-body* simula a evolução de um sistema físico com vários corpos (ou partículas) onde a força gravitacional de Newton de um corpo influencia todos os outros. Esta força causa movimentação contínua dos corpos. Este clássico problema astrofísico ajuda a entender o movimento de diversos corpos celestes desde galáxias com poucos corpos até sistemas de larga estrutura universal.

Para resolver o problema, algoritmos numéricos são utilizados para representar a si-

mulação do problema N-corpos em cada passo evolutivo (chamado também de *time step*). Os algoritmos numéricos podem ser divididos em duas categorias: uma abordagem direta ou um esquema aproximado. A abordagem direta [2] calcula as N^2 forças que são exercidas entre os corpos para cada passo da evolução na simulação, tem uma complexidade temporal de $O(N^2)$. A outra abordagem indica o uso de aproximações para as interações de partículas longínquas, possuindo assim uma complexidade inferior, porém com perda de precisão em relação à abordagem direta. A abordagem direta é utilizada para simular tanto regiões de pouca densidade como de alta densidade [57], *globular star clusters* [55] ou *galatic nuclei* [82], uma vez que nessas regiões há a necessidade de alta precisão nos cálculos. No entanto, devido a sua complexidade temporal de $O(N^2)$, simulações com milhares e, especialmente milhões de estrelas, ainda são um desafio.

Na literatura, existem duas implementações paralelas para o cálculo da força entre as partículas N-corpos pela abordagem direta: o algoritmo de cópia e o algoritmo *Ring* [79]. No algoritmo de cópia, a cada evolução ou *time step*, todas as partículas são copiadas para cada processador utilizado. Desta forma, esse algoritmo requer grande quantidade de memória em cada máquina quando é usado um grande número de partículas. No algoritmo *ring* cada processador precisa armazenar apenas $\frac{N}{P}$ partículas, onde P é o número de processadores utilizados, e, após computar as suas $\frac{N}{P}$ partículas, cada processador as envia para seu vizinho. Esses algoritmos foram projetados para ambientes paralelos, homogêneos e dedicados, como *clusters*, por serem os mais utilizados na época de seus desenvolvimentos e pela simplicidade de implementar algoritmos para esses ambientes.

6.1.1 Algoritmo *Ring*

A aplicação N-corpos particiona seus dados usando a técnica de decomposição de domínio. O domínio é composto pelas partículas ou corpos que são aglomeradas em tarefas e computadas pela Equação 6.1, onde r_i é a posição no espaço tridimensional da partícula i, G é a gravidade e m_i é a massa da partícula i. Toda partícula calcula a soma de todas as forças gravitacionais que agem sobre ela, o que faz realizar o somatório da força de todas as partículas (já que, segundo o modelo do problema, todos os corpos exercem força entre si). A comunicação entre as partições é dada pela estrutura de anel ou *ring* conforme mostra a Figura 6.1. Por esta razão, o algoritmo usado chama-se *algoritmo Ring* [58].

$$\frac{d^2 r_i}{dt^2} = \sum_{j=1; j \neq i}^{N} \frac{G m_j (r_j - r_i)}{|r_j - r_i|^3} \tag{6.1}$$

A estrutura de comunicação é dada pelo particionamento de N em P pedaços e pela propagação dos dados em forma de anel. Cada processador apresenta um estágio que compreende $\frac{N}{P}$ partículas. No primeiro estágio, a interação entre as $\frac{N}{P}$ partículas de da processador é calculada paralelamente e as partículas locais são enviadas para seu vizinho a direita no anel. São necessários mais $P-1$ estágios para receber as partículas vizinhas, calcular a interação entre suas partículas locais e as remotas e propagar as partículas recebidas para os vizinhos a direita no anel (este último envio não é feito no último estágio). Após o P-ésimo estágio, a interação entre todas as partículas do domínio estão calculadas. Na Figura 6.1, as $N = 16$ partículas são divididas entre 4 processos.

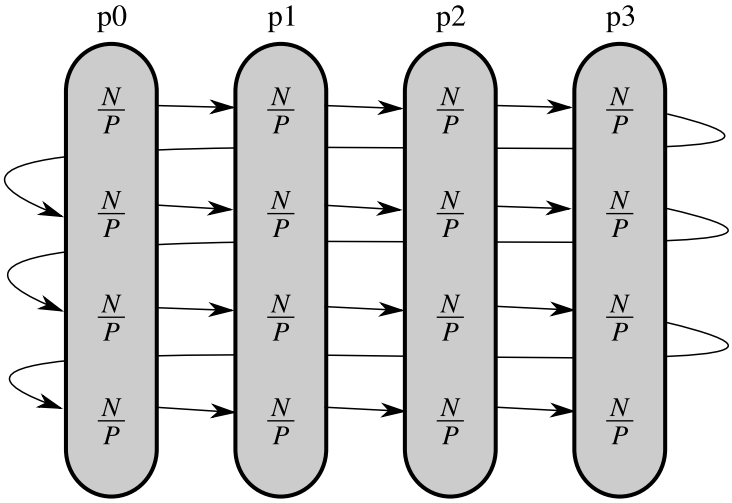


Figura 6.1: Estrutura dos processos para o algoritmo *Ring* usado com a aplicação N-corpos em 1 *time step*.

O Algoritmo 7 representa os passos do método *Ring*. As entradas são o número de partículas N, o número de processos P e o identificador do processo p_k. Cada processo, então, calcula a interação das forças entre suas $\frac{N}{P}$ partículas previamente particionadas (linha 1). Em seguida, envias essas $\frac{N}{P}$ partículas para o processo $p_{(k+1 \mod P)}$, que é o processo imediatamente a sua direita no anel (linha 2). A partir da linha 3, o processo p_k espera receber $P-1$ mensagens dos processos restantes (linha 4), calcula, em seguida, a interação da força das partículas recebidas sobre suas próprias partículas (linha 5) e envia as partículas recebidas para o processo à direita no anel (linha 6). No fim de cada *time step*, as soluções são reunidas centralizadamente, sincronizando os processos para que um novo *time step* possa ser iniciado.

Como já dito, nessas simulações de astrofísica o tempo é discretizado em evoluções ou passos de tempo ou *time steps*. Em cada *time step*, o algoritmo *Ring* é executado para que se obtenha a movimentação das partículas no decorrer do tempo. Por exemplo, em

Algoritmo 7: Algoritmo *Ring* para o problema N-corpos.

Entrada: N - número de partículas
p_k - identificador do processo
P - número de processos

1 Calcula interação entre suas $\frac{N}{P}$ partículas
2 Envia suas $\frac{N}{P}$ partículas para $p_{(k+1 \mod P)}$
3 **para** $i \leftarrow 1$ *a* $P-1$ **faça**
4 Recebe $\frac{N}{P}$ partículas de $p_{(P+k-1 \mod P)}$
5 Calcula interação das partículas recebidas sobre suas próprias partículas
6 Envia partículas recebidas para $p_{(k+1 \mod P)}$

uma execução com I *time steps*, cada i $(1 \leq i \leq P-1)$ *time step* é representado por uma figura semelhante a Figura 6.1, sendo que o próximo *time step* $i+1$ faz parte do processo do *time step* anterior i. Entre cada mudança de *time step*, um processo centralizador realiza a soma total dos resultados obtidos e inicia o novo *time step*.

6.2 Algoritmo *Ring* Autônomo

A solução proposta neste trabalho segue a implementação do modelo 1Ptask e o uso de *middleware* EasyGrid AMS. As partículas são particionadas por uma determinada largura W e agrupadas em W porções de $\frac{N}{W}$. Ao invés do grupo de N partículas ser dividido pelo número de processadores, este é dividido por W. Os estágios também são divididos, gerando comunicação vertical (ver Figura 6.2). As tarefas possuem, portanto, granularidade fina conhecida. Assim, o número de tarefas total passa a ser W^2 (W tarefas horizontais multiplicado por W tarefas verticais). A Figura 6.2 mostra esta estrutura do algoritmo usando o modelo 1Ptask em 1 *time step*.

Na Figura 6.2, existem 16 tarefas equivalentes àquelas da Figura 6.1. Cada tarefa possui $\frac{N}{W}$ partículas e apresenta um identificador p_i, onde $0 \leq i < P$. A estrutura das tarefas pode ser vista em forma de matriz quadrada com suas respectivas linhas e colunas. A primeira linha contém os identificadores no intervalo $0 \leq i \leq W-1$, a segunda em $W \leq i \leq 2W-1$, a terceira em $2W \leq i \leq 3W-1$ e a N-ésima em $(N-1)W \leq i \leq NW-1$. Desta forma, a comunicação continua sendo em forma de anel mas agora é introduzida uma troca de mensagens entre as tarefas na mesma coluna, de cima para baixo.

O Algoritmo 8 apresenta os passos de execução para o então chamado algoritmo *Ring AMS*. Ele apresenta como entrada o número N de partículas, o identificador p_i da tarefa e a largura W. Na primeira linha, é verificado se a tarefa não faz parte das W primeiras.

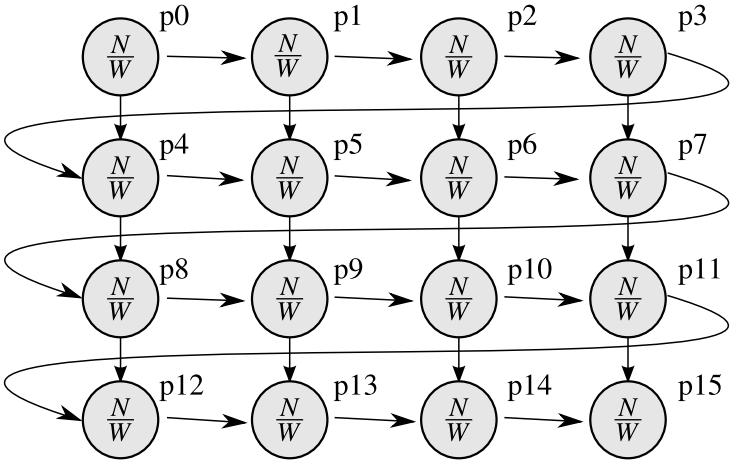


Figura 6.2: Estrutura das tarefas para o algoritmo *Ring* AMS usado com a aplicação N-corpos em 1 *time step*.

Caso não faça parte, ela deve receber $\frac{N}{W}$ partículas da tarefa p_{i-W} imediatamente acima na matriz (linha 2). Em seguida, se a tarefa pertencer a primeira coluna (linha 3), ela irá receber as $\frac{N}{W}$ partículas da tarefa de identificador anterior ao dela p_{i-1} (linha 4). Caso contrário, a tarefa de identificador i receberá $\frac{N}{W}$ partículas de p_{i-W-1}, isto é, da tarefa da linha acima e coluna anterior (linha 6). Na linha 7 encontra-se o passo da tarefa que calcula a força das partículas recebidas sobre suas próprias partículas (no caso das primeiras tarefas, esse cálculo é feito apenas com as próprias partículas). A partir da linha 8, iniciam-se os passos de envio das partículas. Se a tarefa possui um identificador menor que $W^2 - W$, isto é, todas as tarefas que não estão na última linha, ela enviará suas $\frac{N}{W}$ partículas para p_{i+W} (a tarefa imediatamente abaixo) na linha 13 do algoritmo. Antes disso (linha 9), o envio das partículas recebidas é feito. Se o resto da divisão de i por W for $W - 1$ (última coluna), o envio é feito para a tarefa de identificador $i + 1$ linha 10. Caso contrário, as partículas são enviadas para a tarefa p_{i+W+1} (linha 12).

Para introduzir múltiplos *time steps* ao algoritmo *Ring* AMS, as últimas W tarefas irão enviar a informação resultante para um processo centralizador R, que realizará a soma total dos resultados obtidos. Esse processo R irá passar as novas partículas para o próximo *time step* e as W primeiras tarefas irão recebê-las. São etapas extras e representam um momento de sincronismo da mesma forma que no algoritmo *Ring*. A Figura 6.3 mostra a introdução deste processo centralizador R no algoritmo em cada *time step* I.

O parâmetro W influencia na performance do algoritmo quando exposto a um ambiente computacional heterogêneo, compartilhado e/ou dinâmico. O W pode ser variado dinamicamente de modo que aumente a quantidade de tarefas caso haja menos poder computacional e diminua a quantidade de tarefas caso aumente o poder computacional.

Algoritmo 8: Algoritmo *Ring* AMS para o problema N-corpos.

Entrada: N - número de partículas
p_i - identificador da tarefa W - largura

1 **se** $i \geq W$ **então**
2 Recebe $\frac{N}{W}$ partículas de p_{i-W}
3 **se** $i \mod W = 0$ **então**
4 Recebe $\frac{N}{W}$ partículas de p_{i-1}
5 **senão**
6 Recebe $\frac{N}{W}$ partículas de p_{i-W-1}
7 Calcula a força das partículas recebidas sobre suas próprias partículas
8 **se** $i < W^2 - W$ **então**
9 **se** $i \mod W = W - 1$ **então**
10 Envia as $\frac{N}{W}$ partículas recebidas para p_{i+1}
11 **senão**
12 Envia as $\frac{N}{W}$ partículas recebidas para p_{i+W+1}
13 Envia suas $\frac{N}{W}$ partículas para p_{i+W}

Esse processo de aumentar e diminuir a largura W é chamado de *maleabilidade* [41, 36]. É uma técnica autônoma de autoconfiguração. A Figura 6.3 apresenta um exemplo de maleabilidade. As 4 primeiras tarefas são estruturas de acordo com $W = 2$, no *time step* $I = 1$, enquanto as próximas, o $W = 4$, no *time step* $I = 2$.

Para aplicar a maleabilidade no algoritmo *Ring* AMS, pontos de reconfiguração precisam ser estabelecidos. Um bom indício de ponto de reconfiguração seria entre a cada *time step*. Assim, cada *time step* teria sua largura independente, sem a necessidade de alterações complexas, o que aconteceria se fosse no meio de um *time step*. Portanto, toda vez que alcançar um ponto do algoritmo em que o W deverá ser recalculado, ou seja, a cada *time step*, um novo escalonamento das tarefas pode ser realizado. Para fazer um novo escalonamento, o bem conhecido algoritmo heurístico HEFT [116] – *Heterogeneous Earliest Finish Time* – foi escolhido.

O HEFT é um algoritmo que tem complexidade temporal de $O(n^2 \times p)$ (onde n é o número de tarefas e p é o número de processadores) e serve principalmente para escalonar um conjunto de tarefas dependentes em processadores heterogêneos e distribuídos, considerando o tempo de comunicação. Em geral, ele tem como entrada o conjunto de tarefas, o DAG com as relações de precedência, o conjunto de processadores, o tempo de execução de cada tarefa em cada processador e o tempo de comunicação entre os processadores.

Nos pontos de reconfiguração, existem duas etapas custosas: a escolha do W e o

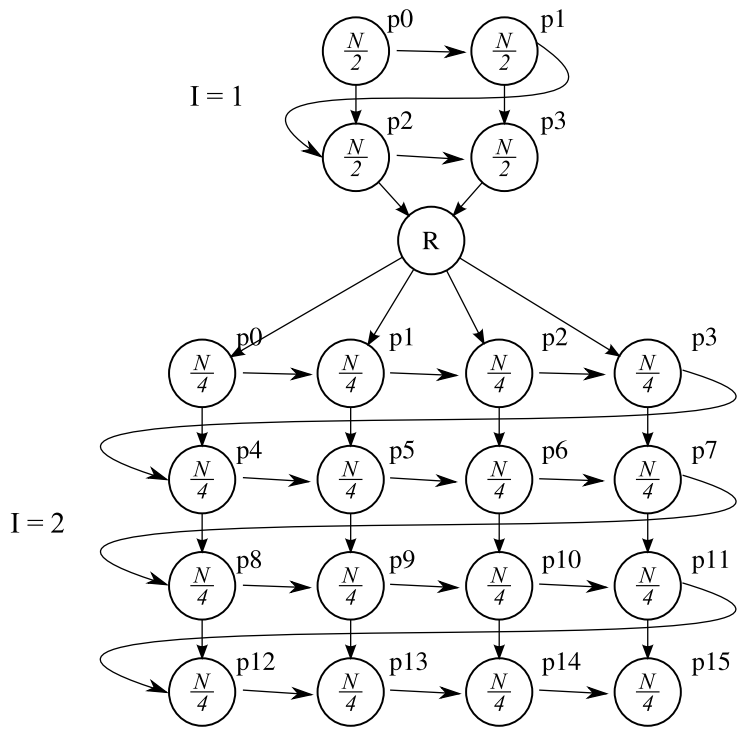


Figura 6.3: Exemplo da estrutura das tarefas para o algoritmo *Ring* AMS em 2 *time steps* com $W = 2$ no *time step* $I = 1$ e $W = 4$ no *time step* $I = 2$.

escalonamento das tarefas. Para reduzir o tempo do algoritmo HEFT, duas otimizações podem ser feitas. 1) Como as tarefas seguem uma relação de precedência simples, a leitura do DAG em arquivo não é necessária. Isto pode ser substituído por funções matemáticas que definem o sucessor e o predecessor de cada tarefa para a aplicação N-Corpos, segundo a sua lei de formação. 2) Foram retiradas ou simplificadas algumas etapas do algoritmo que são desnecessárias a aplicação em questão. A mais importante foi a retirada da etapa inserção de tarefas em espaços ociosos durante o processo de escalonamento. O HEFT analisa o escalonamento realizado e busca por buracos ou *gaps* em que possa inserir uma tarefa e diminuir, assim, o *makespan* (o tempo total do escalonamento). Como o tempo de computação tende a ser maior que o tempo de comunicação, os períodos ociosos no processador praticamente não existem, evitando que possam ser ocupados por alguma tarefa. Com esta alteração, a complexidade temporal do algoritmo para esse tipo de aplicação fica igual a $O(N \times P)$ e, portanto, melhora consideravelmente o desempenho.

A escolha do novo W é calculada através de uma heurística para selecionar o melhor W. Existem diversos algoritmos que podem ser utilizados: métodos analíticos (o que não existe para esta aplicação e, portanto é inviável), descida de encosta (*hill climbing*), *simulated annealing* e métodos de busca por eliminação. Dentre os métodos de busca por eliminação encontram-se a busca por dicotomia, métodos Fibonacci e métodos da

seção áurea. Pela simplicidade, baixo custo e eficiência, o método da *seção áurea* [2] foi considerado para ser empregado nesta parte do trabalho.

6.2.1 Escalonamento Dinâmico

Um escalonamento dinâmico eficiente é muito importante para esta aplicação com o domínio fortemente acoplado, devido a uma grande quantidade de comunicações entre tarefas. Em termos de qualidade, seria uma boa solução utilizar o HEFT para fazer o escalonamento dinâmico. No entanto, em termos de eficiência no tempo de execução, esta não é uma boa solução, já que sua complexidade temporal é de $O(n \times p)$ e o algoritmo heurístico deveria ser executado em intervalos curtos de tempo. Haveria uma grande sobrecarga. É por esta razão que o HEFT é apenas executado a cada *time step*.

O escalonador dinâmico padrão do EasyGrid AMS foi feito inicialmente para aplicações que não apresentam dependências entre si. Basicamente, existem duas filas de tarefas: as prontas e as pendentes. Quando ocorre um evento de desbalanceamento, apenas as tarefas prontas são reescalonadas. Em um desbalanceamento recorrente, tarefas de uma mesma coluna (ver Figura 6.2) são reescalonadas uma por vez, já que uma tarefa mais abaixo depende da tarefa mais acima. No entanto, tarefas de uma mesma coluna poderiam ser reescalonadas juntas se o escalonador dinâmico do EasyGrid AMS usasse, no cálculo do escalonamento, as tarefas pendentes também. Logo, existem dois possíveis tipos de escalonamento para a aplicação N-corpos: o *Escalonamento Dinâmico para tarefas Independentes* – EDI – e o *Escalonamento Dinâmico por Colunas* – EDC.

6.3 Avaliação Experimental

Os experimentos foram executados no *cluster* Oscar na Universidade Federal Fluminense. Foram selecionadas 16 máquinas de 8 núcleos, cada, para os experimentos deste capítulo. Três testes diferentes foram realizados de acordo com os seguintes objetivos: 1) conhecer a sobrecarga do algoritmo *Ring* AMS em relação ao *Ring* (foram usadas apenas 4 máquinas); 2) avaliar o algoritmo *Ring* AMS em um ambiente heterogêneo, compartilhado e dinâmico (foram usadas as 16 máquinas); 3) testar os escalonamentos dinâmicos para tarefas independentes e por colunas em um ambiente também heterogêneo e compartilhado (foram usadas também as 16 máquinas). Como não havia disponível uma versão sequencial, a avaliação foi conduzida comparando apenas as execuções paralelas.

6.3.1 Teste de Sobrecarga

No intuito de avaliar a sobrecarga do algoritmo *Ring* AMS, foram usadas 4 máquinas dedicadas, totalizando 32 CPUs, para executar o algoritmo tradicional *Ring* e o proposto *Ring* AMS. A ideia é investigar o quanto de sobrecarga o algoritmo *Ring* AMS introduz na aplicação N-corpos, em um ambiente de execução favorável ao algoritmo *Ring*. O algoritmo *Ring* AMS particiona as tarefas do algoritmo *Ring* em tarefas menores, inserindo mais comunicação durante a execução. Isto certamente ocasiona sobrecarga e, portanto, é necessário conhecer a sua intensidade conforme o tamanho da instância aumenta.

Tabela 6.1: Médias dos tempos de execuções de um *time step* para os algoritmos *Ring* e *Ring* AMS no teste de sobrecarga.

		Número de partículas			
		250 mil	500 mil	750 mil	1 milhão
Ring	Média:	132,63	532,02	1192,63	2119,90
	I.C.	0,002	0,083	0,025	0,005
Ring AMS	Média:	155,22	557,26	1218,35	2145,51
	I.C.	0,077	0,051	0,028	0,024

A Tabela 6.1 apresenta a média de tempo de execução e seu intervalo de confiança (I.C.) para um *time step* usando o algoritmo *Ring* e *Ring* AMS. A confiança do intervalo é de 96% e as médias foram obtidas através de 8 rodadas de execução. Os números de partículas avaliados neste experimento foram 250 mil, 500 mil, 750 mil e 1 milhão. É possível perceber que a quantidade de trabalho de um *time step* aumenta exponencialmente conforme aumenta-se o número de partículas. Ainda, a diferença de tempo entre o algoritmo *Ring* AMS e o *Ring* é aproximadamente fixo (varia de 23s a 26s). Esta sobrecarga fixa ocorre devido ao tempo de criação dos processos no EasyGrid AMS (cerca de 10 milissegundos por processo) e ao atraso da troca de mensagens (que é menor em relação ao tempo de criação do processo).

A Figura 6.4 mostra o comportamento da sobrecarga do algoritmo *Ring* AMS em relação ao algoritmo tradicional *Ring* conforme aumenta-se o número de partículas da aplicação N-corpos. Quando a instância é pequena, a sobrecarga é alta, em cerca de 17%. Conforme o número de partículas for aumentando, a sobrecarga cai exponencialmente. Através deste experimento é possível concluir que a sobrecarga é baixa (cerca de 1,2% para 1 milhão de partículas) para instâncias grandes, que são o foco deste trabalho.

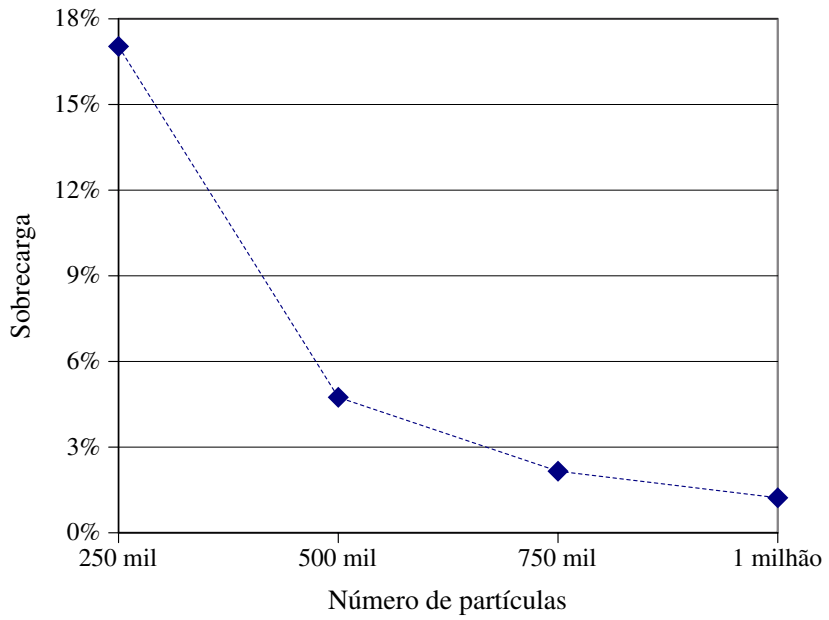


Figura 6.4: Sobrecarga do algoritmo *Ring* AMS em relação ao algoritmo *Ring*.

6.3.2 Teste em Ambiente Compartilhado

O objetivo dos experimentos é comparar quatro configurações diferentes do algoritmo *Ring* AMS para a aplicação N-corpos em um ambiente compartilhado e dinâmico. Aqui, o uso do escalonamento dinâmico e da maleabilidade são variados. A configuração chamada SDSM é aquela *sem escalonamento dinâmico e sem maleabilidade* ativados; CDSM é *com escalonamento dinâmico e sem maleabilidade*; SDCM é *sem escalonamento dinâmico e com maleabilidade*; e CDCM é *com escalonamento dinâmico e com maleabilidade*. O escalonamento usado neste experimento é para tarefas independentes (EDI). O valor de N é de 1,5 milhões de partículas.

Primeiramente, o número de tarefas e o valor de W, conforme a variação dos *time steps*, podem ser analisados através do gráfico da Figura 6.5. O valor de W é a raiz quadrada do número de tarefas (eixo vertical do gráfico). Em todas as execuções sem maleabilidade, o valor de W mantém-se constante em 128. Em todas as execuções com maleabilidade, o valor de W varia conforme os pontos plotados no gráfico da figura. Quanto maior é o número de processadores ociosos, maior é o valor de W, já que o escalonador do EasyGrid AMS necessita gerar mais tarefas para compensar as tarefas demoradas (que estão nas máquinas ocupadas).

Para fazer a avaliação e tornar o ambiente compartilhado e dinâmico, cargas extras foram inseridas nas máquinas. Foram usadas 16 máquinas de 8 núcleos (no total, 128

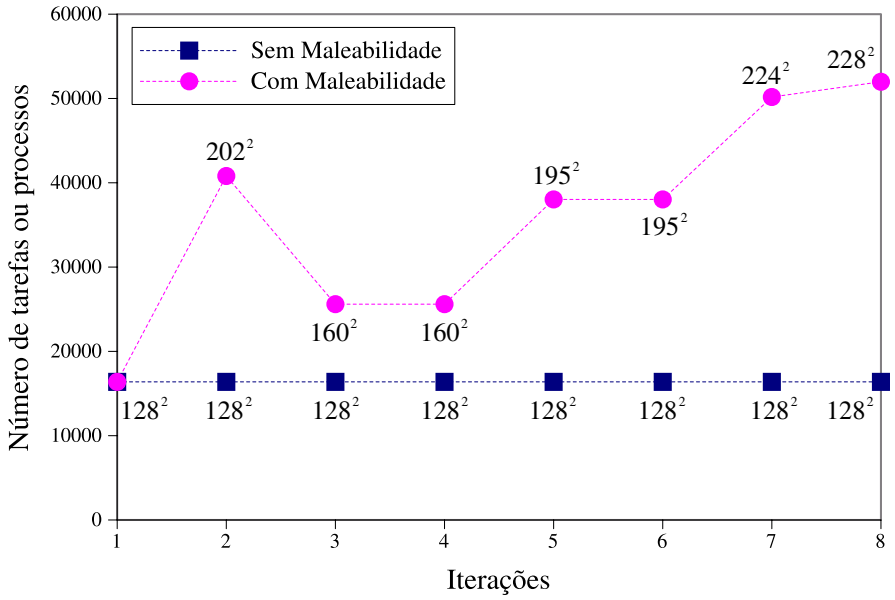


Figura 6.5: Número de tarefas por iteração para a aplicação N-corpos.

Tabela 6.2: Configuração do ambiente compartilhado para experimentos com o N-corpos.

Time steps	Máquinas Ociosas	Máquinas Ocupadas	Poder Computacional
1,2	8	8	50%
3,4	4	12	50%
5,6	8	8	33%
7,8	12	4	50%

CPUs). De dois em dois *time steps* do algoritmo *Ring* AMS, as cargas são alteradas. São 8 *time steps*: nos dois primeiros *time steps* 8 máquinas possuem 50% de seu poder computacional (isto é, foram inseridas 8 cargas extras em cada 8 máquinas); no terceiro e quarto *time steps* 12 máquinas possuem 50% de seu poder computacional; no quinto e sexto *time steps*, 8 máquinas ficam com 33% de seu poder computacional (neste caso, foram inseridas 16 cargas em cada 8 máquinas); e nos dois últimos *time steps*, 4 máquinas ficam com 50% de seu poder computacional. A Tabela 6.2 resume esta configuração. A primeira coluna mostra o *time step*, a segunda e terceira colunas apresentam a quantidade de máquinas ociosas e ocupadas, respectivamente, e a quarta coluna exibe o poder computacional das máquinas ocupadas.

A Tabela 6.3 sumariza os tempos de execução em segundos de cada iteração para cada combinação de uso de escalonamento dinâmico e maleabilidade: SDSM, SDCM, CDSM e CDCM. Em um ambiente homogêneo e dedicado, seria esperado obter os tempos de execução de cada iteração semelhantes. Mas como o ambiente é compartilhado e dinâmico, o tempo de cada iteração varia de acordo com as cargas inseridas dinamicamente nas

Tabela 6.3: Tempo de execução em segundos de cada iteração para cada combinação de uso de escalonamento dinâmico e maleabilidade.

Time step	SDSM	SDCM	CDSM	CDCM
1	2386,44	2390,20	2393,97	2405,36
2	2406,25	2123,49	2749,02	1863,43
3	2400,38	2046,71	2478,48	1999,13
4	2933,01	2302,58	2530,73	2210,33
5	3533,48	1964,54	2362,70	2059,90
6	2703,55	2459,02	2646,76	2261,56
7	2387,07	1527,64	1679,36	1504,47
8	2387,22	1521,81	1670,85	1490,93

máquinas. A Figura 6.6 mostra graficamente tais resultados obtidos. O eixo X representa cada *time step* e no eixo Y são encontrados o tempo em segundos de cada iteração. A linha azul, com um losango, representa o SDSM. A linha rosa, com um quadrado, representa o SDCM. A linha vermelha, com um triângulo, representa o CDSM. A linha verde, com um círculo, representa o CDCM.

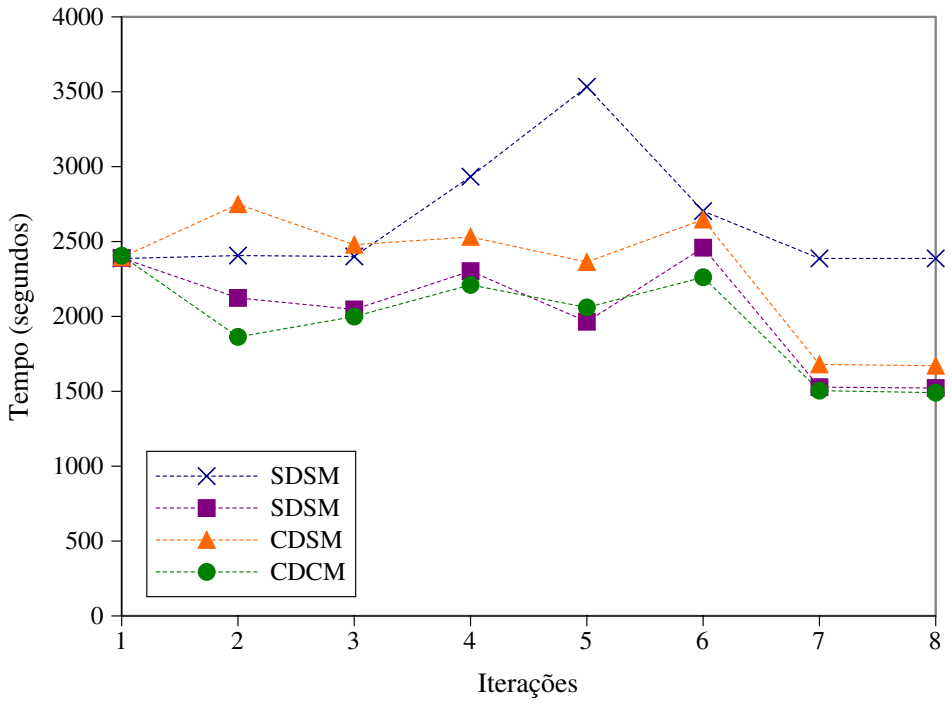


Figura 6.6: Comparação das propostas *Ring* em um ambiente compartilhado e dinâmico.

Pela Figura 6.6 é possível perceber que a solução sem escalonamento dinâmico e sem maleabilidade obtém os piores resultados em quase todos os *time steps* (exceto no primeiro *time step*). Este resultado era esperado uma vez que, conforme a mudança do ambiente, ele não consegue se adaptar. Com apenas o escalonamento dinâmico ativado, o resultado dos tempos de cada *time step* melhora, mas nas duas primeira iterações o tempo é ruim.

Isto ocorre devido ao fato do CDSM ter poucas possibilidades de escalonamento pois apresenta a mesma quantidade de tarefas desde o início até o fim da execução. A execução com apenas maleabilidade consegue obter um melhor desempenho devido à propriedade de aumentar e diminuir a largura W. Com isso, o algoritmo consegue se adaptar melhor ao ambiente. Quando junta-se o escalonamento dinâmico e a maleabilidade o resultado é ainda melhor já o escalonamento apoia a maleabilidade, balanceando as tarefas. Utilizando também o escalonamento dinâmico, a adaptação ao ambiente é mais eficiente, melhorando cerca de 3% o tempo de execução em relação a execução SDCM. Em relação à execução SDSM, a melhoria chega a ser de 34%. Em relação ao pico do SDSM na Figura 6.6, a iteração 5 apresenta a maior quantidade de carga (duas vezes o número de carga extra em metade das máquinas) e por isso ocorre esse tempo elevado em relação às outras iterações. É interessante ressaltar também que as cargas são introduzidas no momento em que cerca de 60% de uma iteração é concluída.

6.3.3 Teste do Escalonamento Dinâmico por Colunas

Em todos os experimentos até aqui, o escalonador dinâmico usa uma política padrão de execução de tarefas da aplicação compartilhando recursos com outras aplicações. A *política ociosa* do EasyGrid AMS indica que o escalonador dinâmico irá remover as máquinas ocupadas e usará para a execução apenas as máquinas ociosas. Uma máquina é considerada ociosa quando apresenta acima de 80% de seu poder computacional disponível.

Através da política ociosa, é possível simular a situação de remover e inserir máquinas no sistema, além de ser uma política útil para algum cenário de execução. Neste experimento, esta política de escalonamento foi utilizada para avaliar os dois escalonamentos para a aplicação N-corpos descritos na Seção 6.2.1: o escalonamento dinâmico para tarefas independentes – EDI – e o escalonamento dinâmico por colunas – EDC.

Tabela 6.4: Configuração do ambiente para o teste de comparação entre EDI e EDC.

Time steps	Máquinas Ociosas	Máquinas Ocupadas
1,2	8	8
3,4	12	4
5,6	16	0
7,8	12	4
9,10	8	8

O objetivo é avaliar os escalonamentos obtidos por cada uma das versões considerando um ambiente compartilha e dinâmico. Foram utilizadas, novamente, 1,5 milhões de

partículas e 16 máquinas do *cluster* Oscar. Cargas com processamento intensivo foram inseridas considerando a seguinte distribuição: nos *time steps* 1 e 2, 8 máquinas possuem 50% de seu poder computacional; nos *time steps* 3 e 4, 4 máquinas possuem 50% de seu poder computacional; nos *time steps* 5 e 6, nenhuma máquinas possui carga; nos *time steps* 7 e 8, 4 máquinas possuem 50% de seu poder computacional; e nos *time steps* 9 e 10, 8 máquinas possuem 50% de seu poder computacional. A Tabela 6.4 resume esta configuração do experimento.

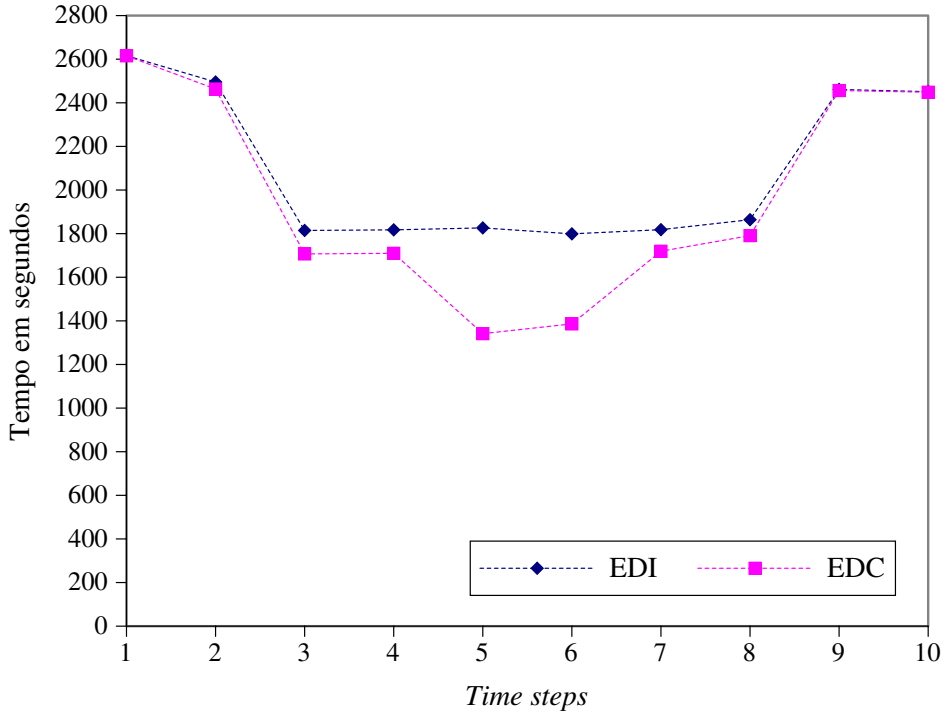


Figura 6.7: Comparação dos tempos de execução de cada *time step* para EDI e EDC.

O gráfico da Figura 6.7 compara os tempos de execução de cada *time step* entre as duas versões de escalonamento: EDI e EDC. Os pontos em forma de quadrado representam os tempos dos *time steps* do EDC e os pontos em forma de losango representam os tempos dos *time steps* do EDI. Os resultados mostram que o EDC apresenta melhores resultados em relação ao EDI. Para a maioria das *time steps*, os resultados são semelhantes entre EDI e EDC, embora o EDC seja um pouco mais eficiente. Exceto nos *time steps* 5 e 6, quando ocorre a mudança do *time step* 4 para o 5 (isto é, todas as máquinas tornam-se ociosas), o escalonador tem a visão de apenas 128 tarefas prontas. Por esta razão (ter poucas tarefas) e pelo fato do EDI só escalonar tarefas prontas e ignorar as dependentes, ocorre um desempenho pior comparado ao EDC. O EDC tem um comportamento mais próximo ao esperado, com desempenho melhor. Isto ocorre porque ele considera as tarefas dependentes e as reescalonam junto com as prontas. Embora isso pareça trazer uma

sobrecarga maior, já que um grande número de tarefas é migrado, esta é uma solução mais eficiente para o cenário indicado.

Na verdade, o ideal poderia ser um misto entre o EDI e EDC: mover um grupo de tarefas, isto é, a pronta e algumas de suas sucessoras imediatas. O EDC basicamente faz com que a aplicação retorne para o modelo com granularidade grossa, tendo a diferença de poder migrar as tarefas quando for apropriado.

6.4 Resumo

Aplicações de domínio fortemente acoplado particionam os dados associados ao problema entre tarefas que são dependentes entre si e precisam constantemente se comunicar para trocar dados comuns. Por elas requerem uma alta taxa de comunicação, o desenvolvimento de seus processos paralelos tende a ser mais complexo em relação a aplicações com dados menos acoplados. Por este motivo, um algoritmo específico chamado *Ring* aplicado ao problema N-corpos é usado como representante desta classe de aplicações.

O algoritmo *Ring* divide N partículas em P partições associadas a P tarefas ou processos (modelo 1Pproc). Como é preciso calcular a força exercida entre cada par de partículas, os dados são propagados em forma de anel, de modo todas as tarefas troquem suas partículas. O algoritmo *Ring* AMS é a versão autônoma do algoritmo tradicional. Ao ser colocado no modelo 1Ptask, as tarefas são particionadas por uma determinada largura W horizontalmente e verticalmente e, portanto, apresentam granularidade fina. Ao invés do grupo de N partículas ser dividido pelo número de processadores, este é dividido por W, e cada N estágios que pertenciam a um processador são divididos entre W tarefas. O EasyGrid AMS é usado para adicionar auto-otimização, autoconfiguração e autorrecuperação à aplicação. Ele é responsável por realizar o escalonamento destas tarefas com dependências predeterminadas e ajustar dinamicamente a largura W, de modo a garantir uma boa otimização da execução em um ambiente heterogêneo e compartilhado.

Os primeiros resultados mostraram que, quanto maior for a instância, menor será a sobrecarga do algoritmo *Ring* AMS em relação ao *Ring*. Por exemplo, para uma instância de 1 milhão de partículas, a sobrecarga é de 1,2%. O segundo experimento considera um ambiente heterogêneo e compartilhado e seus resultados mostram que a maleabilidade junto com o escalonamento dinâmico conseguem melhorar até 34% em relação a execução com a ausência deles. Ainda, uma modificação no escalonamento dinâmico do EasyGrid AMS pode trazer melhores resultados.

Capítulo 7

Workflow Científico – Modelagem de Nicho Ecológico

Os *workflows* científicos podem ser vistos como uma coleção de tarefas relacionadas entre si que são processadas em recursos distribuídos em uma ordem bem definida para alcançar um objetivo específico. As tarefas apresentam decomposição funcional, dependências representadas por um DAG, granularidade grossa, o que difere do modelo do EasyGrid AMS mas não interfere seu uso. Um exemplo de *workflow* pode ser encontrado na modelagem de nicho ecológico que tem o objetivo de prever a distribuição de uma espécie em um espaço geográfico usando modelos matemáticos e probabilísticos sobre sua distribuição já conhecida no ambiente. Geralmente, este processo de modelagem inclui etapas como a criação do modelo, o teste do modelo e a projeção do modelo em um ambiente geográfico que possuem dependências entre si e formam um fluxo de trabalho, ou seja, um *workflow*.

Em um sistema computacional paralelo e distribuído, o gerenciamento de *workflows* científicos é necessário uma vez que os trabalhos apresentam dependência e precisam de uma ordem de execução estática e dinâmica, no caso de ambientes de execução compartilhados e dinâmicos. Geralmente isto é feito por sistemas gerenciadores na forma de *middleware*, uma camada entre a aplicação e a infraestrutura.

Neste capítulo, o problema de modelagem de nicho ecológico é descrito, assim como os *workflows* utilizados para processar os dados ecológicos e ambientais e retornar informação útil ao pesquisador. Uma comparação prática e conceitual entre diferentes tipos de gerenciadores é realizada de modo a induzir uma execução mais eficiente para uma determinada aplicação de *workflow* científico. Como esta aplicação é diferente das outras já tratadas nesta tese, porque apresentam tarefas de granularidade grossa, a avaliação é comparativa entre sistemas gerenciadores existentes e representativos.

Figura 7.1: Demonstração da modelagem de nicho ecológico [93].

7.1 Caso de Estudo: Modelagem de Nicho Ecológico

A modelagem de nicho ecológico – ENM (*Ecological Niche Modelling*) – vem tornando-se um procedimento comum para determinar a extensão da distribuição geográfica das espécies, provendo uma ferramenta poderosa para prever a distribuição de espécies em diferentes contextos geográficos e temporais, assim como para estudar outros aspectos da evolução e ecologia. ENM tem sido bastante usado em várias situações como por exemplo: busca de espécies raras ou ameaçadas de extinção; identificação de regiões adequadas para a (re-)introdução de espécies; previsão do impacto das mudanças climáticas sobre a biodiversidade; definição e avaliação de áreas protegidas; prevenção da propagação de espécies invasoras, auxiliando no estudo dos impactos das mudanças climáticas sobre a biodiversidade, entre outras aplicações importantes. Basicamente, a modelagem de nicho precisa de algoritmos e também dois tipos de dados: bióticos e abióticos [95].

Dados bióticos representam informações sobre as espécies com dados biológicos e geográficos. A rede *SpeciesLink* [26] é um sistema distribuído de informação que integra dados primários de coleções científicas. O *Global Biodiversity Information Facility* [52] – GBIF – é um portal web que disponibiliza informações sobre um grande número de espécies de várias partes do mundo. Dados abióticos são dados ambientais que influenciam a vida das espécies. O *Worldclim* [127] é um conjunto de informações climáticas mensais de todo o mundo incluindo temperatura, volume de chuva e altitude, o que constitui um exemplo de conjunto de dados abióticos. Outro exemplo é o Instituto Nacional de Pesquisa Espacial [65] (INPE). Ele oferece um catálogo de imagens de satélite baseadas em leituras de diferentes sensores e câmeras.

Aplicações de ENM combinam informações sobre a ocorrência de espécies (bióticas) com bases de dados ambientais (abióticas) na forma de camadas rasterizadas georeferenciadas (tais como temperatura, precipitação e salinidade) para gerar potenciais modelos de distribuição, como mostrado na Figura 7.1. Este processo inclui uma combinação de fases (as partições): criação, teste e projeção do modelo. Uma vez que as fases

de teste e projeção requerem a criação de um modelo, e a projeção pode depender da fase de teste, esta sequência de passos se torna um *workflow*. Outras fases além das já descritas, como conversão de imagem ou geração de outros dados, podem ser adicionadas ao *workflow*.

7.1.1 OpenModeller

O openModeller – OM – é um *framework* de modelagem de nicho ecológico importante desenvolvido por um centro brasileiro de referência em informações ambientais: o CRIA (Centro de Referência em Informação Ambiental), junto com outros parceiros nacionais e internacionais [93, 84]. O *openModeller* trabalha com modelos de distribuição potencial e inclui mecanismos para a leitura de dados ambientais e de ocorrência de espécies, seleção de camadas ambientais sobre as quais o modelo deve se basear, criação de um modelo de nicho fundamental, projeção de modelo em um cenário ambiental e escrita de dados em vários tipos de arquivo relativos a grandes imagens. Vários algoritmos são providos na forma de *plugins*, incluindo:

- BIOCLIM: envoltórios bioclimáticos (*Bioclimatic envelopes*).
- CSMBS: modelo de clima espacial (*Climate Space Model*).
- GARP: algoritmo genético para produção de conjunto de regras (*Genetic Algorithm for Rule-set Production*).
- GARP_BS: GARP com melhores subconjuntos (*Best Subsets*)
- DG_GARP: implementação do GARP para *desktop*.
- DG_GARP_BS: implementação do GARP_BS para *desktop*.
- ENFA: análise de fator do nicho ecológico (*Ecological-Niche Factor Analysis*).
- ENVSCORE: pontuação de envoltório (*Envelope Score*).
- ENVDIST: distância ambiental (*Environmental Distance*).
- MAXENT: entropia máxima (*Maximum Entropy*).
- NICHE_MOSAIC: mosaico de nicho (*Niche Mosaic*).
- ANN: rede neural artificial (*Artificial Neural Network*).

- RF: florestas aleatórias (*Random Forests*).
- SVM: máquina de vetores de suporte (*Support Vector Machines*).
- AQUAMAPS: algoritmo para organismos aquáticos.

A maior parte dos métodos ENM utilizam uma abordagem dirigida a dados baseada em correlação. Estes métodos tentam construir uma representação dos requisitos ecológicos fundamentais de uma espécie com base nas características ambientais dos locais de ocorrência conhecidos. Neste caso, como mostrado na Figura 7.1, o *openModeller* necessita de informações ambientais e parâmetros específicos de cada algoritmo para gerar os dados de instância do modelo. Os resultados da projeção em uma região geográfica mostram as áreas nas quais as espécies tem possibilidade de ocorrência [84].

No *openModeller*, para obter o modelo de distribuição baseado em um algoritmo, existem três ferramentas básicas de linha de comando: o *om_model*, que gera modelos de distribuição de acordo com um algoritmo selecionado (*i.e.*, ele recebe os dados ambientais e sobre a ocorrência de uma espécie, e modela a distribuição potencial de acordo com um algoritmo especificado); o *om_test*, que testa o modelo gerado pelo *om_model* e pode gerar ambas a matriz de confusão e a curva de características de operação do receptor (ou ROC - *Receiver Operating Characteristic*) em sequência, ou apenas uma delas; e o *om_project*, que é usado para projetar o modelo e criar como saída um mapa com os dados de distribuição.

Através de, pelo menos, estas três ferramentas são determinados *workflows* OpenModeller. A fase de modelagem (*om_model*) é executada inicialmente para que o modelo seja gerado de acordo com algum algoritmo. A partir deste modelo, pode-se executar uma fase de teste (*om_test*) ou, diretamente, uma fase de projeção. A projeção pode ser dependente do teste também. Um exemplo de *workflow* OpenModeller pode ser visto na Figura 7.2. Este *workflow* foi fornecido pelo projeto EuBrazilOpenBio [39] e será tratado como uma das instâncias de experimentos deste capítulo. Neste caso, apenas existe dependência em dois níveis: entre projeção/teste e modelagem. No entanto, um *workflow* mais real (que não será tratado aqui por motivo de simplificação) teria três níveis, onde a fase de projeção depende das fases de testes e as fases de teste dependem de cada modelagem associado.

Além destas três ferramentas fundamentais, existem outras como: *om_console* (executa modelagem, teste e projeção em uma única ferramenta), *om_algorithm* (exibe informações sobre os algoritmos), *om_sampler* (mostra os dados ambientais), *om_points* (lê

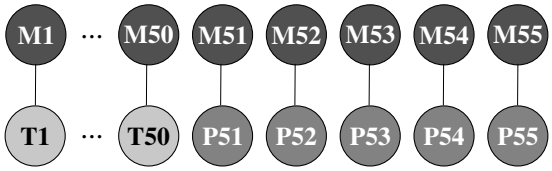


Figura 7.2: *Workflow* OpenModeller.

dados de pontos de ocorrência de espécies de vários formatos), *om_pseudo* (gera pontos aleatórios dentro de uma dada máscara de região ambiental), *om_viewer* e *om_niche* (com apropriada interface gráfica, visualizam mapas de distribuição e modelos, respectivamente).

7.1.2 Ferramenta de Projeção Paralela

A ferramenta de projeção do *openModeller*, *om_project*, tem uma versão paralela baseada em MPI. Ela utiliza o modelo (gerado pelo *om_model* com o algoritmo desejado) para projetar a viabilidade de um dado cenário ambiental para uma determinada espécie. O resultado é uma imagem representando a distribuição potencial das espécies selecionadas na região escolhida. A princípio, a paralelização do *om_project* é simples: os pontos internos à região de interesse selecionada podem ser particionados em sub-conjuntos menores e distribuídos aos recursos disponíveis para a avaliação do modelo. Esta versão emprega o paradigma de mestre-trabalhador, no qual um mestre é alocado para distribuir os subconjuntos pelos vários processos trabalhadores que, por sua vez, calculam a projeção.

Tabela 7.1: *Speed-ups* da versão paralela do *om_project*.

	2	3	4	5	6	7	8
proj51	1,78	2,46	2,90	3,31	3,74	4,12	4,19
proj52	1,68	2,41	3,08	3,83	4,27	4,77	5,18
proj53	1,78	2,54	3,11	3,70	4,40	4,58	4,87
proj54	1,79	2,46	2,93	3,32	3,78	4,24	4,33
proj55	1,76	2,32	2,76	3,11	3,53	3,80	4,06

Resultados de experimentos utilizando esta versão paralela simples com 4 processadores de dois núcleos são apresentados na Figura 7.3 e na Tabela 7.1. As configurações exatas das máquinas e as instâncias de projeção do *openModeller* estão descritas na Seção 7.4. Nesta versão paralela, o processo mestre também atua como trabalhador. O processo mestre distribui os dados aos processos trabalhadores sob demanda (*i.e.*, os processos trabalhadores pedem dados ao mestre), recebe dados já calculados (e salva estes

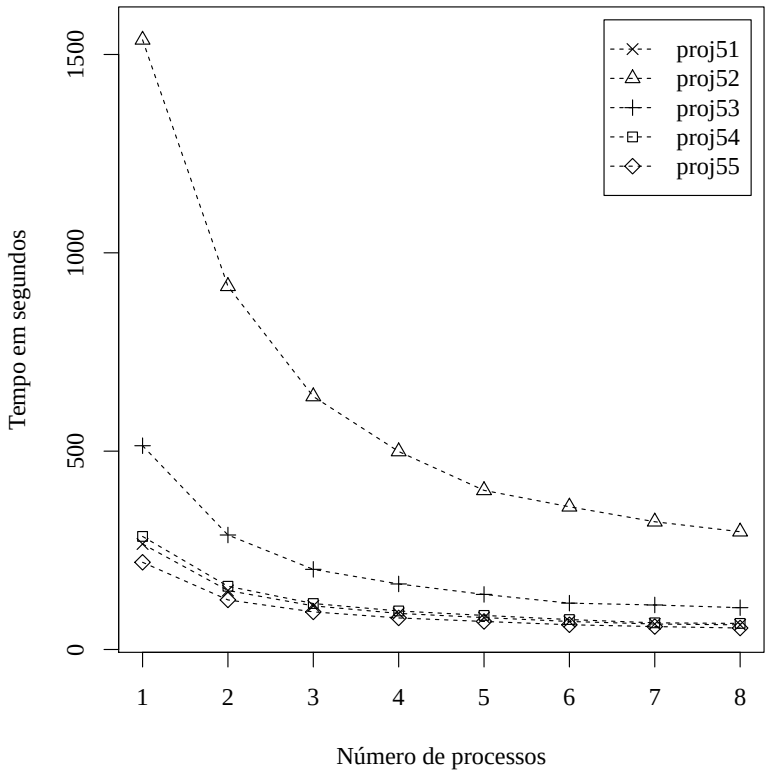


Figura 7.3: Tempo de execução (em segundos) do algoritmo paralelo original *om_project*.

dados em um arquivo) e, quando não há mais demanda dos trabalhadores (porque estes estão executando suas respectivas computações), calcula parte dos dados como um trabalhador. O tempo total de projeção para cada instância – proj51, proj52, proj53, proj54 and proj55 – foi plotado no gráfico da Figura 7.3 e a Tabela 7.1 representa os *speed-ups*. Os resultados mostram que o tempo de execução diminui a medida que o número de processos aumenta, *i.e.*, há um ganho em termos de tempo de processamento com o aumento do número de nós. No entanto, os valores de *speed-up* mostram que este ganho não é tão grande quanto poderia ser. Isto ocorre por causa do gargalo no processo mestre, causado pela recepção e escrita de dados em arquivo. De toda forma, esta versão simples com seus problemas específicos será usada neste trabalho para aumentar a complexidade do DAG do *workflow*. Uma implementação nova foi desenvolvida com o objetivo de resolver estas questões no Capítulo 8.

7.2 Sistemas de *Workflow* Científico

Os *workflows* científicos têm se tornado cada vez mais populares, com mais e mais cientistas e pesquisadores utilizando sistemas computacionais para conduzir suas análises e descobertas a cada dia. Um cenário típico de experimentos é o ciclo repetitivo de mo-

ver dados para super computadores para análise e simulações, realizando a computação necessária e administrando o armazenamento dos resultados. Sistemas de gerenciamento de *workflow* científicos procuram automatizar estes ciclos, de forma que o cientista ou pesquisador possa manter o foco na sua pesquisa, ao invés do gerenciamento dos experimentos [31].

Uma quantidade considerável de trabalho tem sido desenvolvida no uso de sistemas de *workflow* para aplicações científicas. Yu e Buyya [132] proveem uma taxonomia completa dos sistemas de gerenciamento de *workflow* baseada no tipo de projeto, gerenciamento de falhas e movimentação de dados. Eles caracterizam o que constitui um bom sistema de gerenciamento e classificam as diferentes abordagens existentes para a construção e execução de *workflows* em grades. Eles também estudam os sistemas de *workflow* existentes para grades, destacando suas principais características e diferenças. Dentre os métodos estudados, pode-se citar o DAGMan (*Directed Acyclic Graph Manager*) [62, 113], o Pegasus [30], o Taverna [90] e o GridBus [131].

Neste trabalho, considera-se a existência de três filosofias de sistemas de gerenciamento: os sistemas de gerenciamento de recursos (*Resource Management Systems*, ou RMS), os sistemas de gerenciamento de usuário (*User Management Systems*, ou UMS) e os sistemas de gerenciamento de aplicação (*Application Management Systems*, ou AMS). Estes três tipos de sistemas gerenciadores são apresentados no Capítulo 2. Para *workflows* científicos, os RMSs são os mais comuns de serem empregados. O HTCondor DAGMan, citado na taxonomia de Yu e Buyya [132], é considerado um RMS. Um *script* meta-escalonador usado neste trabalho é um UMS e o EasyGrid AMS, que pode lidar com tarefas de *workflow*, representa um AMS.

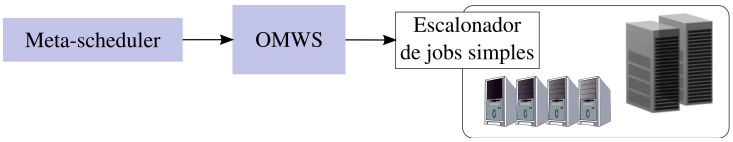
7.3 ENM e Sistemas de *Workflow* Científico

O problema de escalonamento de *workflows* em um sistema distribuído é uma área de pesquisa relevante em computação [80, 133] e vários esforços tem sido feitos visando o desenvolvimento de sistemas de gerenciamento de *workflows* para *clusters* e grades, como apresentado na Seção 7.2. Os três tipos gerais de sistemas de gerenciamento de *workflow* – UMS, RMS e AMS – têm suas características próprias e, dependendo da aplicação, podem tirar vantagem do sistema distribuído (ver Seção 2.3 do Capítulo 2).

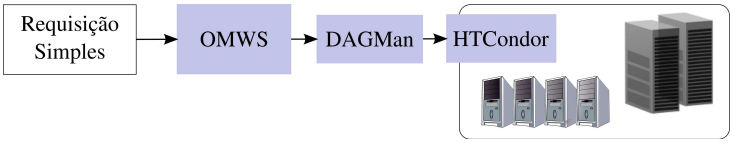
O problema de e-ciência da biodiversidade ENM possui um conjunto de testes que foi criado para avaliar o desempenho de uma instância do serviço web *openModeller* – *open-*

Modeller Web Service, ou OMWS. É um experimento relacionado aos dados de ocorrência de espécies. As espécies são de maracujá (*Passiflora*) que ocorrem no Brasil. Os registros de dados foram obtidos a partir do speciesLink [26] – um sistema distribuído de informação que integra, em tempo real, dados primários sobre coleções biológicas – e todos passaram por uma série de filtros de qualidade. A aplicação é baseada na API versão 1.0 da OMWS e usa pontos de ocorrência para uma única espécie (*Passiflora luetzelburgii*) para gerar modelos com cinco algoritmos diferentes (MAXENT, ENFA, one-class SVM, ENVDIST e GARP BS). Modelos foram gerados para cada algoritmo para uma validação cruzada com 10 *folds* com diferentes combinações de pontos de ocorrência. Finalmente, o modelo final foi gerado e todos os pontos e as probabilidades foram projetadas. Logo, tem-se um grupo de *jobs* ou experimentos que pode ser representado como um DAG com relações de dependência entre modelagem, teste e projeção.

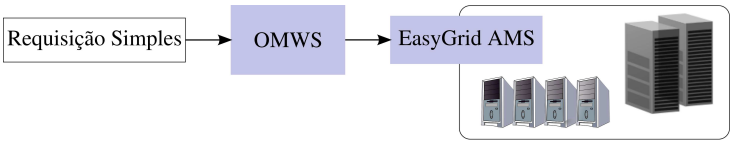
Para permitir a recepção de experimentos ENM com múltiplos estágios e parâmetros, como é o caso do conjunto de teste deste trabalho, e também para tentar melhorar o escalonamento do *workflow*, foi proposto o uso dos mecanismos de um *script* de meta-escalonamento (como UMS), do DAGMan (como RMS) e do EasyGrid (como AMS). O objetivo é comparar e analisar conceitualmente estes três tipos de gerenciadores. A Figura 7.4 mostra a diferença conceitual em relação às requisições de execução do *workflow* realizadas ao servidor OMWS e elas são tratadas nas próximas subseções.



(a) Requisição do meta-escalonador.



(b) Requisição do HTCondor.



(c) Requisição do EasyGrid AMS.

Figura 7.4: Requisições OMWS de cada sistema de gerenciamento de *workflow*.

7.3.1 Script Meta-escalonador

Um meta-escalonador pode ser empregado para controlar o gerenciamento do *workflow* como um UMS e utilizar um escalonador de *jobs* para associá-los a uma máquina. Um cliente escrito na linguagem Perl foi desenvolvido pelo CRIA para submeter requisições para o servidor web do *openModeller* (OMWS) e controlar o *workflow* do cenário descrito na Seção 5.4.1.

O servidor *openModeller* utiliza o protocolo SOAP (*Simple Object Access Protocol*)) para receber arquivos XML contendo todos os parâmetros necessários para a criação, teste e projeção de modelos de nicho ecológicos. Para controlar a dependência entre *jobs*, este cliente continuamente verifica o progresso dos *jobs* predecessores que constituem dependências. Toda a manipulação do DAG é realizada antes que este chegue ao servidor, ainda no lado cliente, pelo *script* meta-escalonador. O HTCondor é usado pelo servidor meramente como um escalonador de *jobs* que simplesmente seleciona uma máquina disponível e inicia os processos. O servidor constrói uma requisição HTCondor para cada submissão e chama o processo do HTCondor. Tarefas paralelas são escalonadas da mesma maneira, utilizando o ambiente MPI.

A Figura 7.4(a) mostra como uma requisição é feita por este UMS. O *script* meta-escalonador é executado no lado do cliente e faz requisições quando uma máquina está disponível. Ele verifica continuamente o progresso da execução perguntando ao servidor periodicamente, o que traz custo associado. Como mencionado anteriormente, o servidor envia a tarefa para o HTCondor que é o RMS do UMS e é responsável por selecionar uma máquina para execução. Neste caso, o meta-escalonador usado não tem controle sobre os recursos existentes, e não interfere na maneira como os *jobs* são executados. Estas tarefas são completamente transparentes para o usuário.

7.3.2 HTCondor DAGMan

HTCondor DAGMan é um sistema gerenciador de recursos que foca nos requisitos dos recursos como carga média da máquina, uso de memória, uso de I/O (entrada e saída de disco) e outros. Diferentemente da abordagem UMS, HTCondor DAGMan é usado para tratar DAGs de *workflow* e fazer a alocação de cada *job* nas máquinas disponíveis. Toda a manipulação de DAG é feita no lado do servidor, como pode ser visto na Figura 7.4(b). O usuário faz uma requisição simples especificando informações sobre o DAG e os *jobs* (como o custo baseado no tempo de execução). O servidor recebe esta requisição, chama

o DAGMan para gerar o escalonamento do *workflow* e o HTCondor para executar a sequência de *jobs* determinada pelo DAGMan. No caso de um *job* que requisita um ambiente paralelo (que executa em um ambiente paralelo ou MPI) de x máquinas, ele será executado apenas quando existirem as x máquinas disponíveis.

Como já comentado, DAGMan gerencia as dependências entre as execuções dos *jobs*. Enquanto HTCondor é responsável por realmente iniciar um *job*, DAGMan gerencia o *workflow*, associando um *job* pronto a uma máquina ociosa de acordo com as dependências, e o entrega ao HTCondor. Este processo tende a ser ineficiente pois nele é incluído o uso do bem conhecido escalonador FCFS – *First Come First Served*. Considerando todo o DAG, a sequência de execução é um importante quesito na redução do tempo de toda a execução do experimento, o *makespan*. Para melhorar isto, é possível configurar a prioridade (custo associado ao tempo de execução) dos *jobs* e tal ação resulta em um melhoramento do escalonamento e, consequentemente, do *makespan*.

7.3.3 EasyGrid AMS

O EasyGrid AMS [86] foi projetado para tratar uma única aplicação MPI, que vai executar diversas tarefas em paralelo. Como o EasyGrid AMS é um gerenciador voltado para a aplicação, existe um único *middleware* para cada uma delas e a coordenação entre eles é feita indiretamente através de medições do sistema (que permitirá um gerenciador perceber a existência de outra aplicação). No caso do *workflow* OpenModeller, o objetivo é gerenciar um fluxo de programas ou *jobs* não MPI. Para tal, foi simplesmente implementado um programa que recebe os *jobs* sequenciais da aplicação e os executa de acordo com o DAG do fluxo de dados. Este programa, chamado de *wrapper*, está usando MPI e, portanto, pode ser utilizado pelo EasyGrid AMS. Deste modo, a aplicação OpenModeller consegue executar seu *workflow* de *jobs* não paralelos respeitando suas dependências e utiliza as funcionalidades autônomas do sistema gerenciador de aplicações.

A Figura 7.4(c) apresenta um visão geral de uma requisição openModeller usando EasyGrid AMS. Similarmente ao HTCondor, uma requisição simples é feita ao OMWS e o servidor chama o EasyGrid AMS para executar e gerenciar os processos nas máquinas. O AMS gerará um escalonamento inicial e iniciará as tarefas nas máquinas alocadas, podendo alterar o escalonamento dinamicamente de acordo com o sistema.

A principal diferença entre os gerenciadores DAGMan e EasyGrid AMS encontra-se no fato do HTCondor, que receberá um *job* do DAGMan, apenas iniciar um *job* em paralelo quando todas as máquinas requisitadas estiverem disponíveis (por exemplo, se um

job paralelo requisitar 4 máquinas, HTCondor irá alocá-lo somente quando 4 máquinas estiverem disponíveis) e o EasyGrid AMS pode iniciar assim que o *job* estiver pronto, adiantando o processamento. EasyGrid AMS não precisa alocar todos os recursos para iniciar os processos. Esta diferença será explorada e analisada nesta Seção. Além disso, com um AMS, é possível adicionar decisões inteligentes durante o escalonamento dinâmico. No caso de *workflows* ENM, algoritmos e tarefas (como criar, testar e projetar um modelo) podem ser customizados e, portanto, o *workflow* original pode mudar de acordo com variações dos recursos. Usando este tipo de gerenciador, as decisões ficam mais próximas da aplicação, permitindo o uso de autonomia neste nível. Esta característica ainda não está implementada no openModeller com EasyGrid AMS, mas é um trabalho futuro.

7.4 Experimentos e Avaliação

O objetivo dos experimentos nesta parte do trabalho é analisar o comportamento de um *workflow* ENM em um *cluster* empregando abordagens orientadas ao usuário, aos recursos e à aplicação. O *cluster* é dedicado e tem apenas 8 máquinas duo processadas. A razão para usar este ambiente paralelo é facilitar o controle do sistema e avaliar o desempenho de cada gerenciador de *workflow*. Neste caso, as máquinas são totalmente livres de intervenção externa, fazendo com que os resultados sejam confiáveis.

Os experimentos foram realizados em um *cluster* dedicado de 8 duo processadores Intel Xeon 3,06 GHz, devido à indisponibilidade do *cluster* Oscar previamente adotado. Três gerenciadores de *workflow* foram avaliados: o meta-escalonador (como UMS), DAGman do HTCondor (como RMS) e o EasyGrid AMS. A proposta é comparar a eficiência entre as três soluções e o escalonamento obtido, principalmente entre as abordagens baseadas em recurso e em aplicação. Todos os experimentos foram realizados com médias e intervalo de confiança de 95% de 5 rodadas. O tempo de *download* dos resultados não foram considerados. A memória total usada pela aplicação foi atendida pelas máquinas (que têm 2 GB cada).

7.4.1 Cenário do Experimento

A instância do openModeller que representa o *workflow* é composta por 55 *om_model*, 50 *om_test* e 5 *om_project*. Este *workflow* simples pode ser visualizado na Figura 7.5. As 50 fases de teste são dependentes das 50 primeiras fases de modelagem (ou criação do modelo) e as 5 projeções são dependentes das 5 últimas fases de modelagem. Cada grupo de 5

sequências é composto pelos algoritmos MAXENT, GARP_BS, ENFA, ENVDIST e SVM, isto é, $\{M1, M6, ..., M51, T1, T6, ..., T46, P51\}$ são MAXENT, $\{M2, M7, ..., M52, T2, T7, ..., T47, P52\}$ são GARP_BS, $\{M3, M8, ..., M53, T3, T8, ..., T48, P53\}$ são ENFA, $\{M4, M9, ..., M54, T4, T9, ..., T49, P54\}$ são ENVDIST e $\{M5, M10, ..., M55, T5, T10, ..., T50, P55\}$ são SVM. O *workflow* openModeller com *om_project* em paralelo está representado na Figura 7.6. Os *jobs* P51, P52, P53, P54 e P55 possuem $n1, n2, n3, n4$ e $n5$ números de processos MPI paralelos, respectivamente.

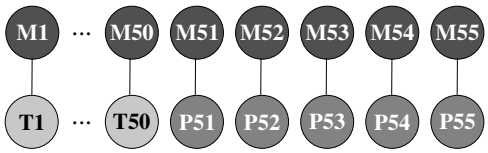


Figura 7.5: *Workflow* do *openModeller*.

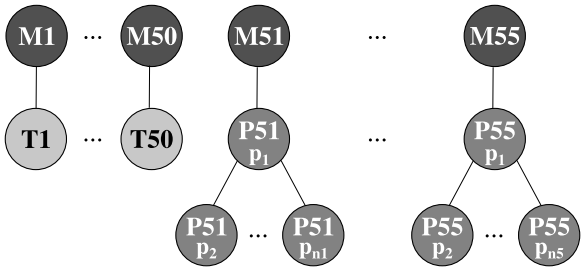


Figura 7.6: *Workflow* openModeller com projeção em paralelo.

A Tabela 7.2 mostra os pesos de cada algoritmo para cada fase, baseado no tempo total de execução sequencial em segundos. Os valores de tempo são médias das execuções. A fase de teste tem peso similar entre os cinco algoritmos. Os tempos de criação do modelo variam bastante, levando apenas 3 segundos para ENFA, ENVDIST e SVM, cerca de 29 segundos para MAXENT e cerca de 1426 segundos para GARP_BS, o mais custoso. Os tempos de projeção também variam e o GARP_BS é quase tão custoso quanto seu correspondente na fase de modelagem. O algoritmo ENFA tem um tempo de projeção próximo de 514 segundos e os outros algoritmos variam entre 220 e 285 segundos. Os custos de entrada para DAGMAN e EasyGrid AMS foram baseados nestes tempos. Basicamente, estes valores são usados para formular custos proporcionais para os *jobs* e eles são os mesmos para DAGMan e EasyGrid AMS enquanto o *script* meta-escalonador não requer especificação destes valores.

Tabela 7.2: Tempo sequencial em segundos de cada fase e cada instância do *workflow*.

Instâncias	Tempo de Modelagem	Tempo de teste	Tempo de Projeção
MAXENT	29	306	266
GARP_BS	1426	315	1537
ENFA	3	307	514
ENVDIST	3	299	285
SVM	3	302	220

7.4.2 Resultados

O *workflow* para *jobs* de projeção sequenciais tem um caminho crítico preestabelecido (modelagem + projeção para GARP_BS) e usando a projeção paralela, introduz-se uma topologia mais interessante já que o caminho crítico não é trivial de se determinar (apenas com técnicas de otimização combinatórias). Existem diversas combinações de *jobs* paralelos e sequencias para um *workflow* ENM.

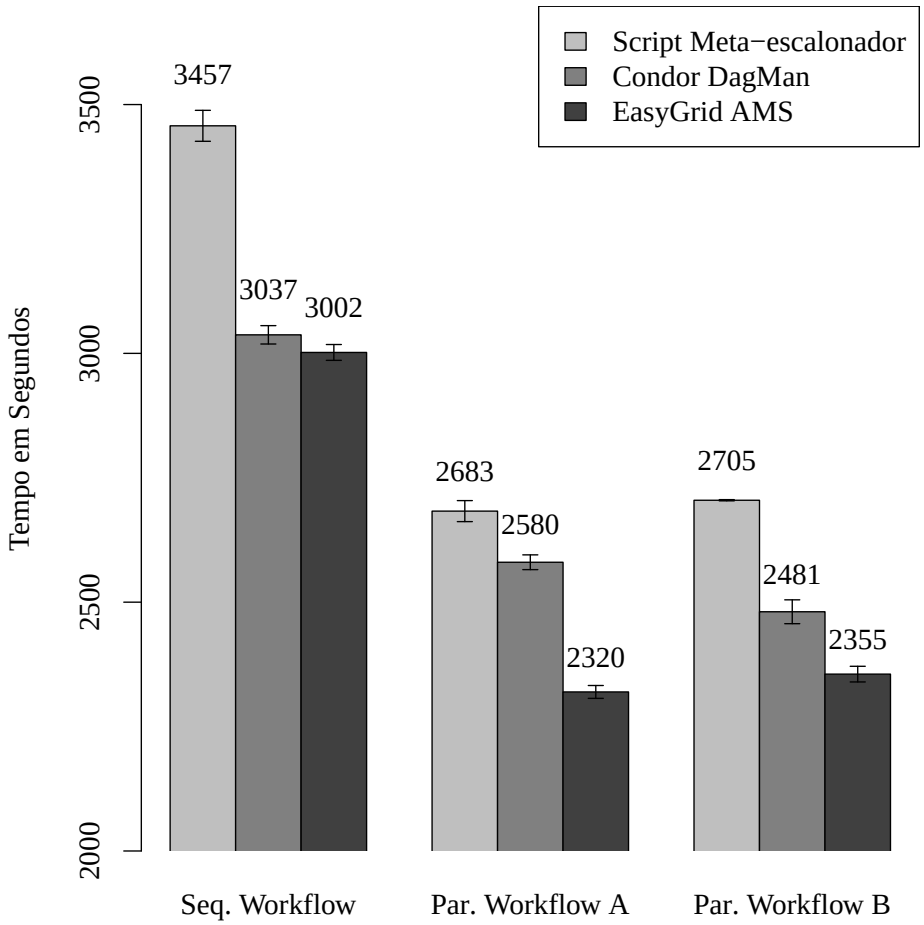


Figura 7.7: *Script* meta-escalonador *versus* HTCondor DAGMan *versus* EasyGrid AMS.

Neste experimento, três configurações de DAG foram escolhidas (de acordo com as melhores execuções) e avaliadas: 1) *workflow* com apenas processos da projeção sequen-

ciais, 2) *workflow* com projeção em paralelo usando 4 processos para cada algoritmo, chamado de *workflow* paralelo A, e 3) *workflow* com projeção paralela usando 8 processos para GARP_BS e 4 processos para ENFA (sendo os outros algoritmos sequenciais), chamado de *workflow* paralelo B. O gráfico da Figura 7.7 mostra a comparação ente o *script* meta-escalonador, HTCondor DAGMan e EasyGrid AMS usando estes DAG. Para cada configuração de DAG, duas barras foram plotadas usando a média do tempo total e seu intervalo de confiança de 95%. A barra cinza clara representa o *script* meta-escalonador, a barra cinza médio representa o DAGMan e a barra cinza escura presenta o EasyGrid AMS.

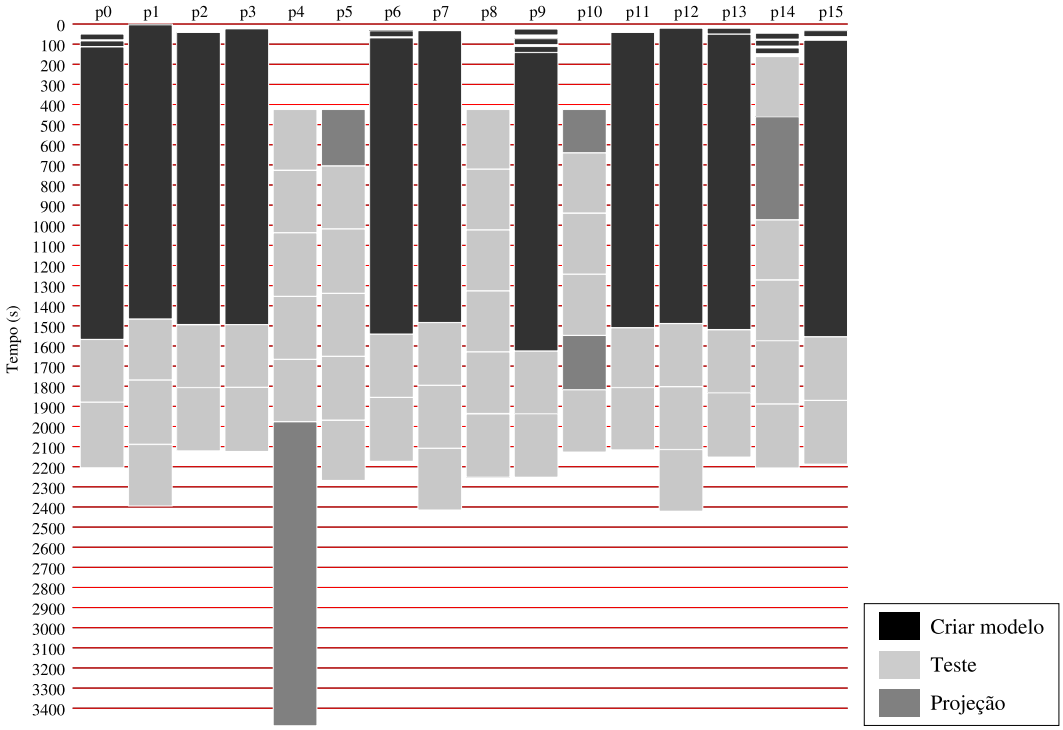


Figura 7.8: Escalonamento obtido pelo *script* meta-escalonador usando o *workflow* com processos sequenciais.

Primeiramente, o caminho crítico dos três *workflows* devem ser analisados. O *workflow* sequencial tem um caminho crítico composto por uma fase de modelagem com GARP_BS (cerca de 1426 segundos) e uma fase de projeção com também o algoritmo GARP_BS (cerca de 1537 segundos), para um total de 2963 segundos na soma (ver Tabela 7.2). Por esta razão, DAGMan e EasyGrid AMS não obtiveram um desempenho menor que este valor. Nas configurações de DAG usando implementações paralela, o caminho crítico é diferente. A Figura 7.8 mostra o escalonamento obtido pelo *script* meta-escalonador para o *workflow* com todos os *jobs* sequenciais. O processo da fase de projeção para o GARP_-BS é a barra cinza no processador $p4$ e ele apenas pode começar depois da maior barra

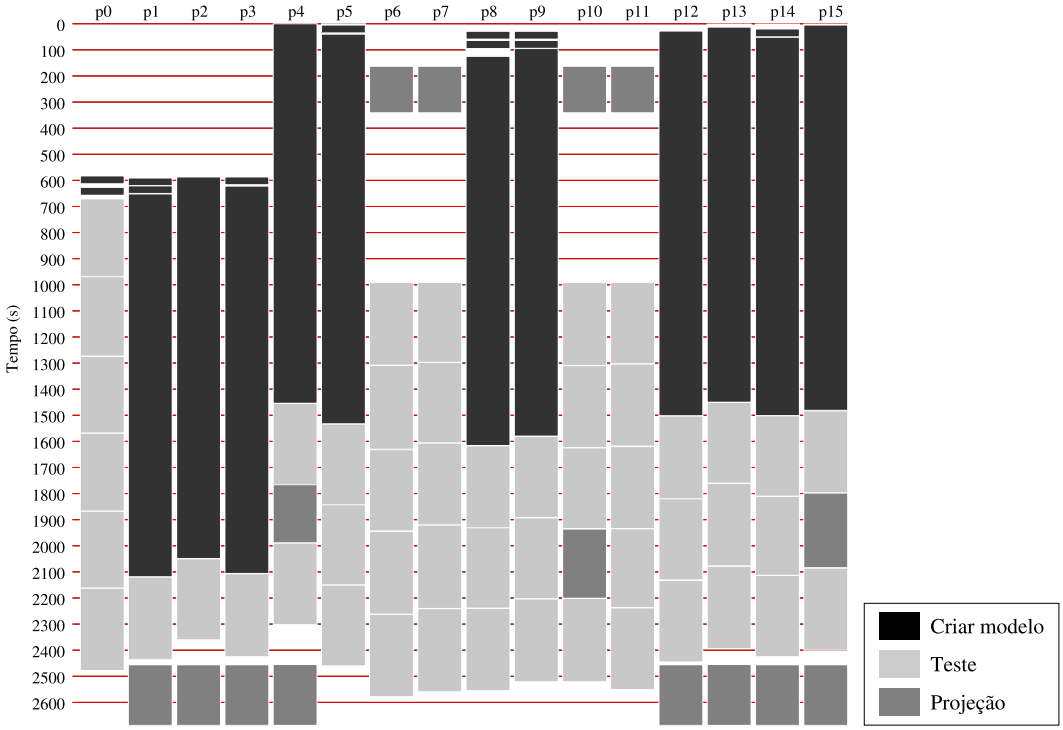


Figura 7.9: Escalonamento obtido pelo *script* meta-escalonador com o *workflow* paralelo B.

preta, representando a fase de modelagem com o GARP_BS. Esta figura também mostra o comportamento do escalonamento gerado a partir do *script* meta-escalonador. O *job* que faz a projeção deveria estar imediatamente após o *job* de modelagem mas resultou em um escalonamento pior, com atrasos no próprio caminho crítico. A projeção paralelo indica um novo e desconhecido caminho crítico[1] composto por outros nós do DAG.

A Figura 7.7 mostra o tempo de execução com o *workflow* sequencial e os valores são próximos entre eles assim como ao valor 2963 (soma da modelagem e projeção do GARP_-BS), embora o EasyGrid AMS tenha apresentado uma tempo médio menor. Os *workflows* paralelos tem o comportamento diferente mas ainda apresentam uma limitação de tempo: a modelagem com GARP_BS ainda possui o tempo de 1426 segundos. Obviamente, o *makespan* aumenta este tempo por causa das outras 50 fases de teste (durando cerca de 300 segundos) e as 5 projeções que são paralelas. Analisando a Figura 7.7 podemos ver que, para as execuções de *workflow* paralelo A e B, o tempo total de execução reduz em relação ao tempo total do *workflow* sequencial. O EasyGrid AMS apresentou o melhor desempenho em comparação com o DAGMan e o *script* meta-escalonador. O DAGMan RMS, por sua vez, teve melhor desempenho em relação ao *script* UMS. Para o *workflow*

[1]O caminho crítico do *workflow* sequencial fácil de obter devido à configuração do DAG enquanto os caminhos críticos para *workflows* usando projeção em paralelo precisam de uma análise de otimização combinatória para serem calculados.

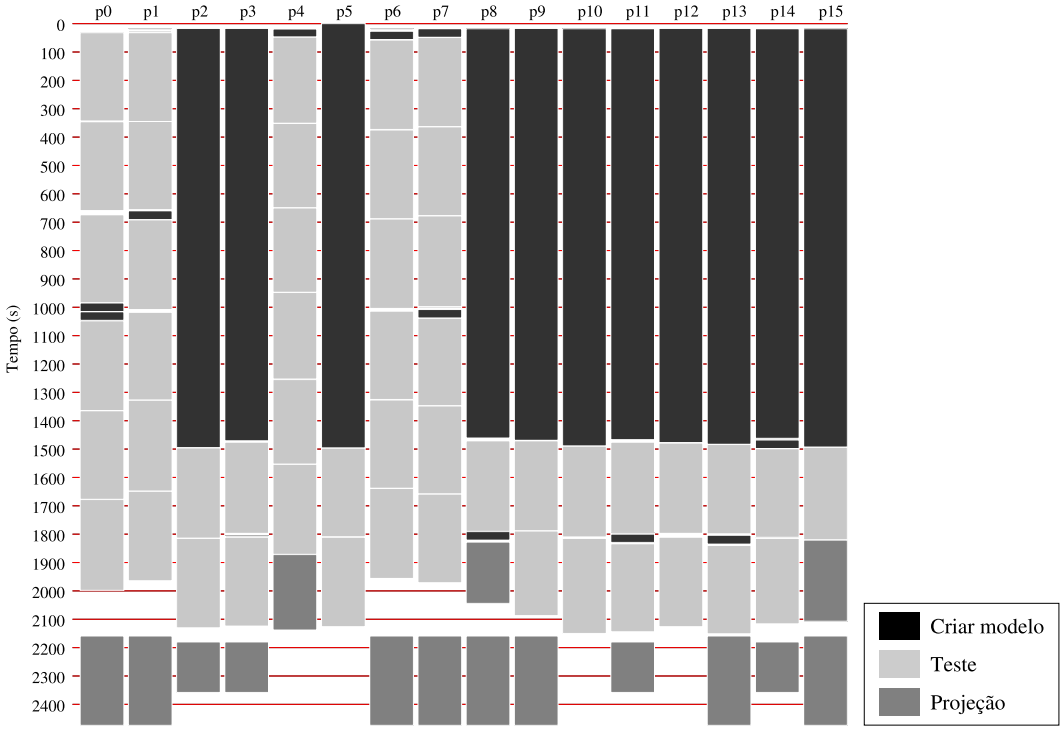


Figura 7.10: Escalonamento obtido pelo HTCondor DAGMan com o *workflow* paralelo B.

paralelo A, a diferença de tempos entre DAGMan e EasyGrid AMS foi de 260 segundos e para o *workflow* paralelo B, a diferença foi de aproximadamente 126 segundos. HTCondor DAGMan teve este desempenho porque ele espera por todos os 8 ou 4 máquinas tornarem disponíveis. Por esta razão, todas as projeções paralelas são empurradas para o final. Consequentemente, alguns nós ficam ociosos até o número de máquinas (determinado pelo número de processos paralelos na fase de projeção) torne-se disponível. No caso do *script* meta-escalonador, a baixa eficiência é explicada pelo escalonador FCFS, já comentado, que não considera o peso dos *jobs*.

As Figuras 7.9, 7.10 e 7.11 mostram o escalonamento total de uma das rodadas para o *script* meta-escalonador, HTCondor DAGMan e EasyGrid AMS para o *workflow* paralelo B. Cada retângulo representa um *job* do *workflow* openModeller (com 8 processos paralelos para o algoritmo GARP_BS). Os *jobs* cinza-claros são a fase de teste, os cinza-escuros são a fase de modelagem e os cinza-médios são as projeções em paralelo. A Figura 7.9 apresenta um escalonamento com vários buracos significantes, que representam o tempo ociosa das máquinas (no eixo x). Isto implica um desempenho ruim para o alocador de recursos HTCondor, que o *script* usa para alocar as máquinas. Muitos *jobs* iniciais demoram muito tempo para começar e não existe uma razão aparente para isto.

Na Figura 7.10, claramente podemos ver que uma fase de espera antes dos *jobs* de

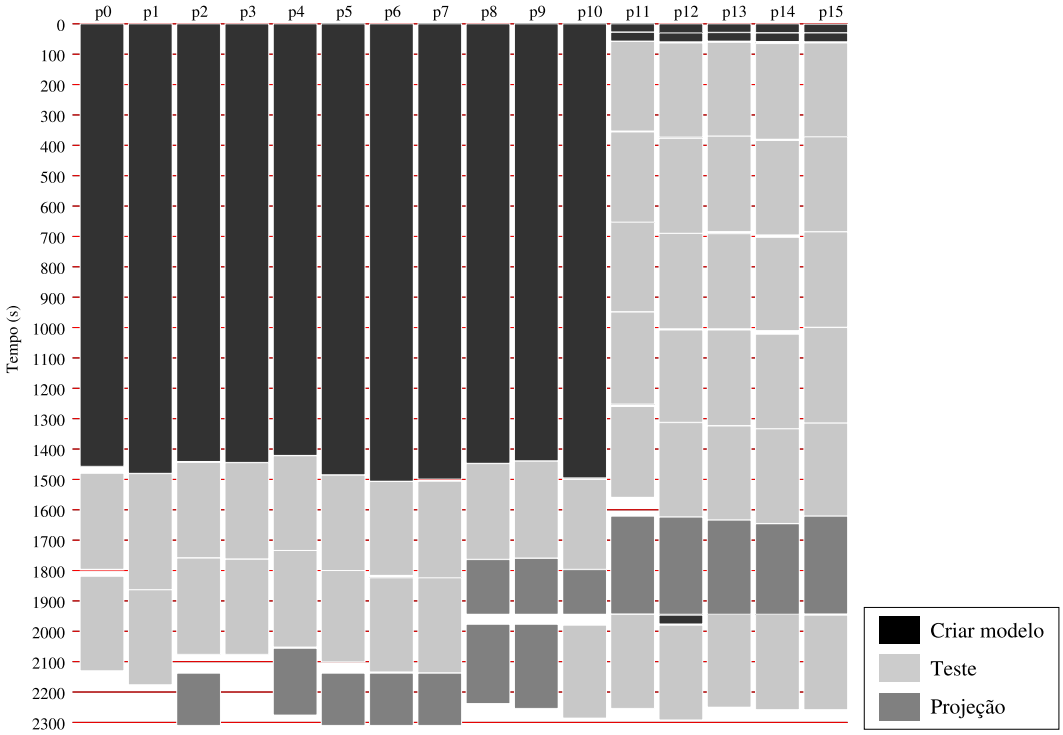


Figura 7.11: Escalonamento obtido pelo EasyGrid AMS com o *workflow* paralelo B.

projeção começarem e alguns atrasos para começar os *jobs* iniciais (por exemplo, o primeiro *job* em $p0$ não inicia no tempo 0). Estes efeitos causas uma perda de eficiência comparado ao desempenho do EasyGrid AMS. Da mesma maneira, a Figura 7.11 apresenta o escalonamento total feito pelo EasyGrid AMS. Existem poucos buracos e os *jobs* paralelos executam assim que eles estiverem prontos (isto é, o *job* antecessor terminou). Pelo fato dos *jobs* paralelos fazerem um balanceamento de carga interno (ver Seção 7.1.2), eles podem ter diferentes cargas e, portanto, diferentes granularidades, já que uma tarefa começou primeiro que a outra.

7.5 Resumo

Workflows científicos são muito usados para indicar o fluxo de trabalho realizado por alguma aplicação científica. Embora haja dependência entre as tarefas de trabalho, na maioria das vezes os *workflows* podem ser executados em paralelo, e um gerenciamento das tarefas é requerido. Este capítulo descreveu um *workflow* de uma aplicação de biodiversidade da *e*-ciência e três esquemas de gerenciamento que podem ser usados como escalonadores de trabalho em infraestruturas paralelas e distribuídas. Cada tipo de gerenciador tem sua peculiaridade em relação a sua orientação no gerenciamento. O UMS é orientado ao usuário, o RMS é orientado aos recursos computacionais e o AMS é orientado

à aplicação.

A análise do tempo de execução total em um ambiente dedicado (com a intensão de facilitar a análise) dos escalonamentos apontaram um melhor desempenho para o EasyGrid AMS em relação aos outros HTCondor DAGMan (como RMS) e o *script* meta-escalonador (como UMS). A submissão do *workflow* do OpenModeller, ferramenta ENM, para o ambiente paralelo é feita através de um servidor web OMWS e todos os gerenciadores devem se comunicar com ele. O meta-escalonador usa um escalonador simples do tipo FCFS que fica na frente do OMWS, isto é, a submissão passa primeiro pelo meta-escalonador, que decide qual tarefa irá executar, e a envia para o servidor. O servidor, então, aloca recursos disponíveis para esta única tarefa usando o HTCondor. Este mecanismo de submissão acarreta problemas como alta latência entre submeter e receber resultados e suas decisões de escalonamento não levam em consideração as características do ambiente de execução. Por outro lado, o HTCondor DAGMan recebe todo o *workflow* e realiza suas decisões considerando as características das máquinas disponíveis. No entanto, ele não considera ambientes compartilhados e dinâmicos e para executar tarefas em paralelo (como aquelas que usam MPI), é necessário ter todos os recursos requeridos disponíveis. No caso do EasyGrid AMS, a execução do *workflow* é dinamicamente gerenciada e qualquer mudança no ambiente é detectada e as tarefas são rebalanceadas. Ainda, sua habilidade de iniciar tarefas paralelas sem ter todos os recursos disponíveis também contribuem para um bom desempenho e um escalonamento sem ou com menos buracos.

Assim, para este específico problema ENM de biodiversidade, os resultados sugerem que a abordagem AMS, que faz o gerenciamento mais próximo da aplicação, pode realmente ser benéfico. Uma vantagem da abordagem EasyGrid AMS é que ao transformar o fluxo de trabalho em uma aplicação MPI paralelo, ele é capaz de tirar proveito dos recursos computacionais, sempre que há um disponível, tentando acelerar a execução e reduzir o número de *gaps* ou buracos no escalonamento gerado.

Capítulo 8

Projeção em Domínio Geoespacial – Projeção do Modelo de Nicho Ecológico

A compreensão da distribuição de dados geoespaciais provenientes de fenômenos ocorridos no espaço fazem parte de um grande desafio para elucidar questões de problemas como previsão do tempo, modelagem de nicho ecológico, planejamento de trânsito, fornecimento de serviços e controle de desastres. Entretanto, para serem usados, estes dados precisam ser analisados e interpretados. Estas atividades são muitas vezes trabalhosas e geralmente executadas por especialistas.

Tais estudos ganharam força, nos últimos anos, devido à propagação de ferramentas como *Sistemas de Informação Geográfico* (SIG ou GIS – *Geographic Information System*) capazes de permitir a visualização e análise simples dos dados espacial e suas informações através de mapas, facilitando o trabalho dos especialistas. Ainda, análises mais complexas podem ser realizadas computacionalmente através de bibliotecas disponíveis capazes de lidar com dados geográficos. A computação tende a ser custosa pois necessita-se analisar uma grande quantidade de dados, já que dados espaciais são de larga escala.

Este capítulo irá tratar da projeção em domínio geoespacial uma vez que o processamento deste tipo de dados é custoso. A paralelização da execução da análise computacional é uma etapa importante do desenvolvimento deste tipo de aplicações. Geralmente, ao se projetar informação em dados geoespaciais, o domínio é particionado em tarefas independentes de granularidade desconhecida. A informação a ser projetada é condicionada aos dados geoespaciais gerando tarefas com mais ou menos trabalho (*e.g.*, informações sobre vegetação terrestre são condicionadas a regiões continentais e regiões oceânicas possuem menos trabalho). A projeção do modelo de nicho ecológico, eferente à etapa de projeção do *workflow* ENM presente no Capítulo 7, será usado como estudo de caso de

uma aplicação de geoprocessamento espacial. Uma solução original e paralela para esta aplicação de estudo de caso será apresentada assim como uma solução proposta paralela e autônoma. Os experimentos servirão como forma de avaliação e comparação entre as abordagens existente e proposta.

8.1 Caso de Estudo: Projeção do Modelo de Nicho Ecológico

Em uma modelagem de nicho ecológico, como aquela apresentada no Capítulo 7, uma etapa muito importante e que requer grande processamento de dados geoespaciais é a projeção do modelo. Uma vez que o modelo de distribuição seja gerado, a projeção no mapa pode ser realizada. Tais modelos podem ser projetados em diferentes regiões geográficas, em diferentes cenários ambientais, tornando-se possível prever o impacto das mudanças climáticas sobre a biodiversidade, evitar a propagação de espécies invasoras, identificar aspectos geográficos e ecológicos da transmissão de doença, ajudar no planejamento de conservação, guiar campos de pesquisas, entre muitos outros usos [95].

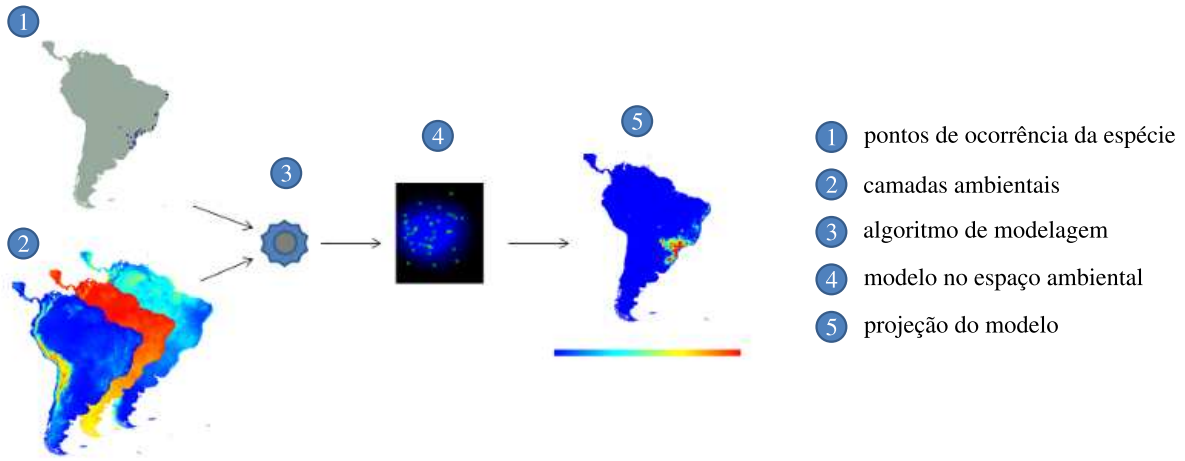


Figura 8.1: Exemplo de modelagem e projeção de um nicho ecológico [93].

OM project – OpenModeller *project* – é a ferramenta do openModeller capaz de fazer a projeção dos modelos de distribuição (já brevemente descrita no Capítulo 7). Ele irá gerar uma imagem com coordenadas (x, y) em uma determinada região geográfica cujos valores associados a cada ponto representam a probabilidade de se encontrar a espécie de acordo com os dados ambientais. A Figura 8.1 representa a modelagem e projeção de um nicho ecológico, onde o item 5 mostra o resultado da projeção de um modelo. A escala de cores que varia de azul a vermelho indica a adequação da espécie na região (vermelho

é alta e azul é baixa). O usuário seleciona camadas de dados ambientais para a projeção e pode, também, selecionar uma máscara geoespacial, para projetar o modelo em uma área diferente de onde foi gerado. A imagem gerada geralmente é de alta resolução de *pixels* que deve ser visualizada por programas específicos como os sistemas de informação geográfica (GIS – *Geografical Information System*) Quantum GIS ou QGIS[97], de código aberto, e ArcGIS [37], de proprietário ESRI.

Basicamente, o Algoritmo 9 sequencial da projeção de um modelo de nicho ecológico executa os seguintes passos para todo ponto (x, y):

1. adquire todos os dados das camadas ambientais no ponto (x, y) (corresponde à Linha 3 do algoritmo);
2. aplica o modelo no ponto (x, y) considerando os tais dados ambientais e, então, gera uma probabilidade p (corresponde à Linha 4 do algoritmo);
3. converte p em valor da imagem e o escreve no arquivo na posição (x, y) (corresponde à Linha 5 do algoritmo).

As entradas do algoritmo são o modelo, os dados ambientais e um arquivo contendo o mapa com as probabilidades. O modelo é previamente gerado pela etapa de modelagem através de um algoritmo descrito na Seção 7.1.1 do Capítulo 7. Assim, a ferramenta do OpenModeller *OM project* tipicamente faz a leitura, processamento e escrita de uma grande quantidade de dados geográficos. Isto faz com que haja a iniciativa da criação de versões paralelas. Atualmente, existe um algoritmo paralelo desenvolvido para *clusters* de computadores homogêneos.

Algoritmo 9: Algoritmo sequencial de projeção do modelo de um nicho ecológico.

Entrada: *modelo* - modelo previamente gerado por um algoritmo.
dados_ambientais - dados ambientais.
Saída: *mapa* - arquivo que contém a projeção.

1 **para** $x \leftarrow 0$ *a* $Dim_x - 1$ **faça**
2 ⎸ **para** $y \leftarrow 0$ *a* $Dim_y - 1$ **faça**
3 ⎸ ⎸ $amb \leftarrow$ *dados_ambientais* na posição (x, y)
4 ⎸ ⎸ aplica *modelo* a *amb* e coloca o valor em p
5 ⎸ ⎸ escreve p em *mapa* na posição (x, y)

8.2 Versão MPI Original

A paralelização original do *OM project* usa um modelo semelhante ao mestre-trabalhador. Além dos processos trabalhadores que computam a projeção, existem dois processos mestres (ao invés de um) responsáveis por distribuir e receber as soluções, respectivamente, o que difere do modelo mestre-trabalhador tradicional, mas que tem efeito semelhante. Denomina-se *mestreD* o processo que distribui os intervalos de dados para cada processo trabalhador e *mestreR* o processo que recebe as soluções de cada processo trabalhador e as escreve no arquivo imagem de saída (que contém a projeção).

O algoritmo de *mestreD* divide o número total de pontos a serem projetados (pontos da imagem) em blocos (valor padrão é 30.000) e os distribui aos processos trabalhadores sob demanda. Nesta aplicação, um bloco corresponde a um conjunto de linhas do mapa a ser projetado e, por isto, cada bloco é associado a (l_i, l_j), onde l_i é a linha inicial e l_j é a linha final. Para isso, um pedido de requisição deve ser feito pelos processos trabalhadores ao processo *mestreD* para que ele envie os dados informando qual o bloco a ser computado. Uma vez calculado, este bloco é enviado para o processo *mestreR* que irá receber os dados e escrevê-los no arquivo de saída. Quando todos os blocos forem calculados e escritos no arquivo pelo processo *mestreR*, o algoritmo termina. A separação de atividades do mestre entre 2 processos, *mestreD* e *mestreR*, é feita para que a distribuição de tarefas não seja sobrecarregada pela recepção dos dados, que é mais custosa.

Algoritmo 10: Algoritmo usado pelo processo *MestreD* da versão original MPI da aplicação de projeção ENM.

Entrada: Dim_x e Dim_y - dimensões X e Y do mapa total.
blocksize - tamanho de um bloco.

1 **para** $c \leftarrow 0$ *a* $\frac{Dim_x * Dim_y}{blocksize} - 1$ **faça**
2 Recebe requisição de bloco de s
3 Calcula bloco (l_i, l_j)
4 Envia (l_i, l_j) para s
5 Recebe requisição de bloco de s
6 Envia $(-1, -1)$ para todos os trabalhadores

Os Algoritmos 10, 11 e 12 descrevem detalhadamente os passos de cada processo. O processo *mestreD* é representado pelo Algoritmo 10. Como entrada, ele recebe as dimensões X e Y do mapa completo e o valor do tamanho fixo de um bloco. O mapa é, então, dividido entre os blocos na linha 1 e, para cada bloco, três procedimentos são realizados. O primeiro deles (linha 2) refere-se ao recebimento da requisição de um bloco de algum processo trabalhador s. O segundo procedimento (linha 3) calcula o intervalo

do bloco e o terceiro (linha 4) envia este bloco, representado por (x_{block}, y_{block}) para o processo s que o requisitou. Na linha 5 a última requisição é recebida e, na linha 6, é enviada a informação de terminação dos processos trabalhadores.

Algoritmo 11: Algoritmo usado pelo processo *MestreR* da versão original MPI da aplicação de projeção ENM.

Entrada: Dim_x e Dim_y - dimensões X e Y do mapa total.
blocksize - tamanho de um bloco.
Saída: *mapa* - arquivo que contém a projeção.

1 **para** $c \leftarrow 0$ *a* $\frac{Dim_x * Dim_y}{blocksize} - 1$ **faça**
2 Recebe *pacote* de s
3 **para cada** $\{(x, y, p)\} \in pacote$ **faça**
4 Escreve p em *mapa* na posição (x, y)

Como já explicado, a parte de recebimento das soluções é feita por um outro processo chamado *MestreR*. Da mesma maneira que ocorre no algoritmo do *MestreD*, o Algoritmo 11 tem como entrada as dimensões totais do mapa e o tamanho de bloco. A saída é representada por *mapa*, um arquivo imagem que contém o resultado da projeção calculada pelos processos trabalhadores. Assim, para cada bloco c (linha 1), o processo *MestreR* recebe *pacote* de um processo trabalhador s na linha 2. Este pacote contém as informações do bloco calcula pelo processo s na forma de uma tupla (x, y, p), onde x e y são respectivamente as posições de coordenadas no mapa e p é o valor da probabilidade calculada. Logo, para cada tupla (linha 3) calculada no bloco recebido, o valor p é escrito em *mapa* na posição (x, y), como mostra linha 4 do algoritmo.

As etapas de execução dos processos trabalhadores da versão original são apresentadas pelo Algoritmo 12. Neste caso, as entradas são o modelo, previamente gerado por um algoritmo na fase de modelagem, os dados ambientais e a dimensão Dim_y que denota o tamanho de uma linha. A variável fim controla o laço de repetição iniciado na linha 2. O término da repetição é feito nas linhas 5 e 6, onde é verificado se o l_i possui o valor -1 (bloco nulo). Caso positivo, a variável fim é alterada para verdadeiro e, então o laço de repetição termina. No início do laço, nas linhas 3 e 4, ocorre o envio da requisição para o processo *mestreD* e o recebimento do bloco requerido em (l_i, l_j) (corresponde ao envio do *mestreD* feito na linha 6 do Algoritmo 10). Se forem valores válidos, o cálculo da projeção é iniciado. Para tal, um conjunto chamado *pacote* é inicializado por $\varnothing$, na linha 8, e nele serão armazenados os valores já projetados. As linhas 9 e 10 indicam o percorrimento do bloco para que o modelo seja aplicado nos dados ambientais nas coordenadas (x, y) (linhas 11 e 12). O valor p resultante da projeção do modelo é guardado no conjunto

Algoritmo 12: Algoritmo usado pelos processos trabalhadores da versão MPI original para a aplicação de projeção ENM.

Entrada: *modelo* - modelo previamente gerado por um algoritmo.
dados_ambientais - dados ambientais.
Dim_y - dimensão Y do mapa completo.

```
1  fim ← FALSO
2  enquanto fim = FALSO faça
3      Envia requisição a mestreD
4      Recebe (l_i, l_j) de mestreD
5      se l_i = −1 então
6          fim ← VERDADEIRO
7      senão
8          pacote ← ∅
9          para x ← l_i a l_j − 1 faça
10             para y ← 0 a Dim_y faça
11                 amb ← dados_ambientais na posição (x, y)
12                 Aplicar modelo em amb e colocar o valor em p
13                 pacote ← pacote ∪ (x, y, p)
14         Envia pacote para mestreR
```

pacote (linha 13). Na linha 14, o envio do pacote completo para o *mestreR* é realizado.

8.3 Versão Autônoma com EasyGrid AMS

A versão MPI original apresenta basicamente 2 problemas: 1) o algoritmo é centralizado e o envio e recebimento de mensagens pelos mestres torna um gargalo de execução, prejudicando o desempenho. 2) o modelo de programação 1Pproc não é propício para ambientes distribuídos de larga escala. O modelo 1Ptask (ver Seção 2.3.7) é mais indicado para ambientes computacionais de larga escala, como *grids*, pois permite uma maior flexibilidade de execução as tarefas para fazer o balanceamento de carga da aplicação. A versão autônoma indica colocar o algoritmo da aplicação no modelo 1Ptask, realizar um processo de autoconfiguração da aplicação e incluir a utilização do *middleware* EasyGrid AMS para o gerenciamento autônomo das tarefas. Como ocorre com a maioria das aplicação projetadas com o EasyGrid AMS, os benefícios dessa abordagem incluem adotar políticas (escalonamento, comunicação, tolerância a falhas) adaptadas às necessidades específicas de cada aplicação, gerando assim um melhor desempenho.

Uma outra questão relacionada ao modelo de programação é a divisão do domínio da aplicação em blocos de tamanhos fixos realizada na versão MPI original. Através

da Figura 8.2, é possível visualizar este problema. A execução sequencial da projeção usando o algoritmo BIOCLIM foi conduzida em uma máquina dedicada, pertencente ao *cluster* Oscar (ver Seção 8.4), e, a cada contagem de blocos (1 linha do mapa), o tempo de execução da projeção nele foi exibido. Pelo gráfico da figura, pode-se verificar que os tempos variam bastante, com média de 0,28 segundos e desvio padrão de 0,15, indicando que a quantidade de trabalho de cada bloco é diferente. Por esta razão, a escolha de um tamanho fixo de bloco atrapalha a execução visto que o cálculo, para alguns blocos, pode ser rápido e, para outros, demorado.

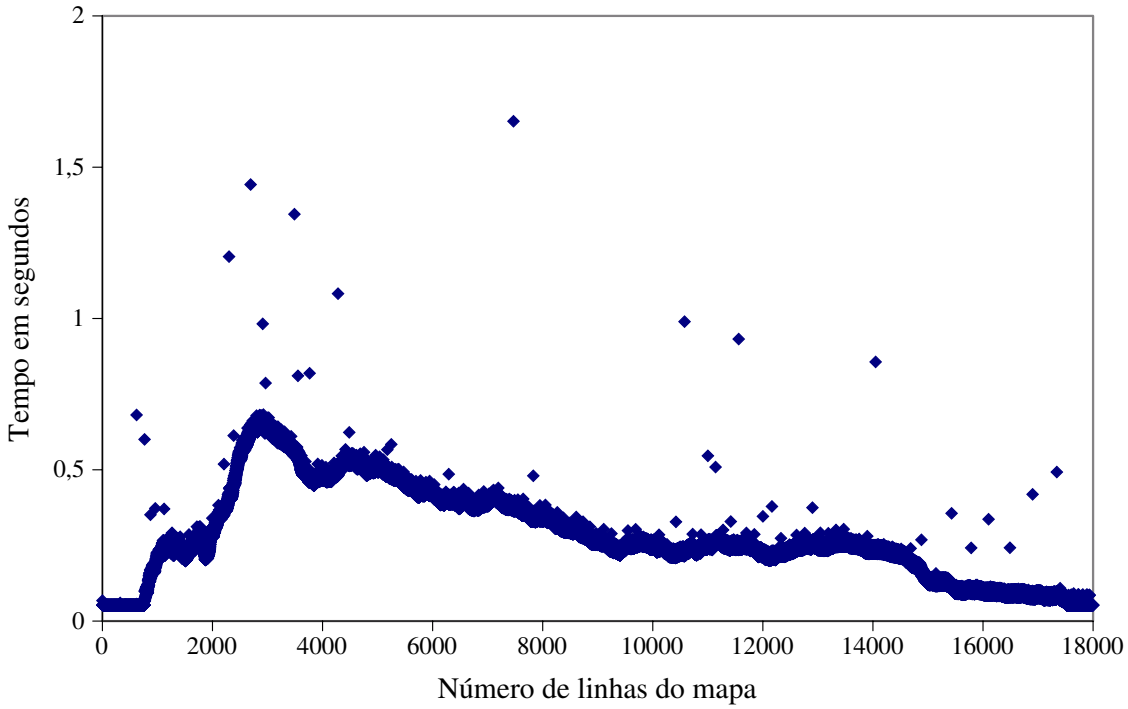


Figura 8.2: Tempo de execução de cada bloco fixo do domínio da projeção ENM.

A versão EasyGrid AMS usa tamanho de blocos variável, já que a granularidade das tarefas é desconhecida. Este é um processo de reconfiguração dinâmica. Cada tarefa (lembrando que no modelo 1Ptask cada tarefa é representada por um processo) recebe um bloco inicial fixo com o valor de, por exemplo, o número total de pontos no mapa dividido pelo número de processadores. A partir de então, a tarefa calcula a projeção do bloco da mesma maneira que no algoritmo sequencial, mas limita a execução deste cálculo por um temporizador *timeout*. Caso o tempo de execução da projeção do bloco ultrapasse *timeout*, uma nova tarefa de idêntico algoritmo é criada para que ela continue a projeção deste bloco. O particionamento do mapa é feito dinamicamente e, a cada nível de criação (tarefas iniciais estão no nível 0, seus filhos estão no nível 1 e assim sucessivamente) o valor do temporizador *timeout* é alterado para um valor inferior. Considerando que *timeout*

representa a granularidade de tempo da tarefa, a razão para a alteração do *timeout* é tornar a granularidade das tarefas que vão ficando para o final mais fina. Deste modo, tais tarefas com granularidade mais fina vão para a fila de tarefas do EasyGrid AMS e são distribuídas entre os processadores eficientemente [85].

O Algoritmo 13 corresponde aos passos de execução de cada tarefa da aplicação de projeção do OpenModeller, chamada de *map*. Como entrada, ele recebe o modelo produzido na fase de modelagem, os dados ambientais, a dimensão Y do mapa original, o bloco e o valor inicial do temporizador *timeout*. Cada tarefa terá uma saída representada por $mapa_{l_i}$, onde l_i é a linha inicial do bloco e é única (identifica a posição do bloco no mapa). As repetições encontradas na linha 1 e 2, onde a segunda é aninhada à primeira, representam o percorrimento de cada ponto no bloco de dados do mapa. O valor x varia de l_i a l_j, passados como argumento do algoritmo, que são as linhas inicial e final que delimitam o bloco. O valor y varia de 0 a Dim_y. Assim, os dados ambientais são capturados na posição (x, y) e colocados na variável *amb* (linha 3). O modelo é, então, aplicado a *amb* e o valor da probabilidade é colocado em p, na linha 4. Em seguida (linha 5), o valor p convertido para valor de imagem é escrito em $mapa_{l_i}$ na posição $(x - l_i, y)$. Como é uma imagem nova, ela deve começar na coordenada $(0, 0)$ e a expressão para a linha $x - l_i$ fornece esta configuração.

O trecho do Algoritmo 13 compreendido entre linhas 1 e 5 apresenta o mesmo comportamento existente no trecho entre as linhas 9 e 13 do Algoritmo 12. Este trecho corresponde ao processamento do problema de projeção ENM. A diferença está na forma como é feito o particionamento (autoconfiguração na versão EasyGrid AMS) e no gerenciamento das tarefas.

Na linha 6 do Algoritmo 13, após a repetição em relação ao eixo Y (isto é, uma linha completa), é verificado se o temporizador esgotou pelo valor de *timeout*. Caso negativo, o cálculo segue para a próxima linha. Em caso positivo, na linha 6, o número de novas tarefas é gerado de acordo com os dados que foram processados neste tempo *timeout*. Por exemplo, num total de 100 blocos, se 10 blocos foram calculados em 5 segundos por 1 tarefa, para calcular os restantes 90 blocos é preciso de 9 tarefas ($ntarefas = 9$). Neste caso, o valor de $ntarefas$ é estimado. Na linha 8, o valor de *timeout*, que será passado para as novas tarefas, é decrementado de acordo com o nível de criação das tarefas (tarefas iniciais estão no nível 0, seus filhos estão no nível 1 e assim sucessivamente). A razão para isto é tornar a granularidade das tarefas que vão ficando para o final mais fina e permitir, então, um melhor balanceamento de carga feito pelo EasyGrid AMS. Na linha 9, o restante

Algoritmo 13: Algoritmo usado pelas tarefas *map* da aplicação de projeção ENM com o EasyGrid AMS.

Entrada: *modelo* - modelo previamente gerado por um algoritmo.
dados_ambientais - dados ambientais.
Dim_y - dimensão Y do mapa completo.
(l_i, l_j) - bloco a ser calculado.
timeout - temporizador inicial.
Saída: $mapa_{l_i}$ - mapas parciais.

```
1  para x ← l_i a l_j faça
2      para cada y ← 0 a Dim_y faça
3          amb ← dados_ambientais na posição (x, y)
4          Aplica modelo a amb e coloca o valor em p
5          Escreve p em mapa_{l_i} na posição (x − l_i, y)
6      se temporizar por timeout então
7          Calcula ntarefas
8          Calcula novo timeout
9          Divide bloco restante (x + 1, l_j) por ntarefas e coloca cada sub-bloco em S
10         para cada k ∈ S faça
11             Cria tarefa com argumentos modelo, dados_ambientais, Dim_y, k e
               timeout
```

do bloco é dividido entre as $ntarefas$ tarefas e cada sub-bloco é adicionado ao conjunto S de $ntarefas$ elementos. Por fim, para cada $k \in S$ (linha 10), uma nova tarefa é criada com os argumentos *modelo*, $dados_ambientais$, Dim_y, k e *timeout* (linha 11).

No fim de todo esse processo de mapeamento, toda a imagem estará calculada mas dividida entre diversos arquivos de saída. Uma tarefa *reduce* é, então, usada para agrupar os dados em cada arquivo e gerar uma imagem final contendo todo o mapa de projeção. Esta etapa pode ser feita usando um agrupamento binário em paralelo (algoritmo de *merge* binário paralelo), o que acelera o processamento em relação ao processamento sequencial.

8.4 Avaliação Experimental

Dois tipos de experimentos foram realizados: 1) para comparar a versão MPI original com a versão autônoma com EasyGrid AMS e 2) para avaliar a escalabilidade da projeção do openModeller com o EasyGrid AMS. No primeiro tipo de experimento, um *cluster* com 2 máquinas Intel Xeon X5650 com 2 processadores de 6 núcleos, cada, totalizando 24 CPUs, e com 24 GB de memória principal. O segundo tipo de experimento usa um *cluster* chamado *Oscar* que possui 40 máquinas no total, sendo apenas 16 usadas, de

8 núcleos cada, totalizando 128 CPUs. Novamente, no experimento 1, o *cluster* Oscar não foi utilizado devido a sua indisponibilidade momentânea. Todos estes ambientes de execução estavam dedicados aos experimentos.

Para os valores apresentados nesta seção de experimentos, a eficiência é calculado considerando a equação: $eficiencia = \frac{speedup}{P} \times 100\%$, onde $speedup = \frac{tempo\ sequencial}{tempo\ paralelo}$ é o *speed-up* da execução paralela em relação à versão sequencial que estava disponível. A métrica eficiência indica o quanto um algoritmo paralelo utiliza bem os recursos envolvidos.

8.4.1 Experimento 1

No experimento 1, p processos trabalhadores são usados para ambas as implementações – original com MPI e autônoma com EasyGrid AMS – em p CPUs. Portanto, na verdade, $p + 2$ processos são usados para a versão original (existem os 2 processos extras). Os algoritmos BIOCLIM, ENVSCORE, GARP, DG_GARP, DG_GARP_BS, MAXENT, GARP_BS, SVM, RF e NICHE_MOSAIC foram escolhidos para serem usados neste experimento.

As Tabelas 8.1 e 8.2 apresentam os resultados comparativos entre a versão MPI original e a versão EasyGrid AMS usando múltiplos algoritmos. Para cada algoritmo, existem 2 conjuntos de colunas: uma para a versão MPI e outra para a EasyGrid AMS. Para cada versão paralela e cada número de processos, o tempo de execução total (em segundos), o *speed-up* e a eficiência (razão entre o *speed-up* e P) são indicados nas tabelas também. O *speed-up* é a razão entre o tempo sequencial e o paralelo.

O gráfico da Figura 8.3 provê uma melhor visualização da comparação dos resultados produzidos por ambas as versões paralelas. Cada grupo de barras com rótulos de 4, 8, 12, 16 e 24 processos em paralelo representam o ganho da versão EasyGrid AMS em relação a versão original MPI, isto é, o quanto o *speed-up* da versão em questão foi superior ao *speed-up* da versão MPI. Cada barra em cada grupo é um algoritmo usado na projeção do openModeller. A versão paralela da projeção com o EasyGrid AMS sempre obtém um *speed-up* maior em relação à versão original MPI e, na maior parte das vezes, é próximo a p. Para a maioria dos algoritmos, conforme o número de processos aumenta, o ganho do EasyGrid AMS em relação ao MPI aumenta também. Para BIOCLIM (barra azul escura), como $p = 24$, o *speed-up* da versão da projeção com EasyGrid AMS chega a ser 70% maior que o valor do *speed-up* da versão original MPI.

Tabela 8.1: Resumo dos resultados comparativos entre as versões original com MPI e autônoma com o EasyGrid AMS para a projeção ENM (parte 1/2).

		Versão Original com MPI			Versão com EasyGrid AMS		
	p	Tempo (s)	*Speed-up*	Efic.	Tempo (s)	*Speed-up*	Efic.
BIOCLIM	4	1008,67	2,58	65%	710,01	3,67	92%
	8	607,51	4,29	54%	363,97	7,15	89%
	12	469,83	5,54	46%	261,32	9,96	83%
	16	478,18	5,45	34%	199,86	13,03	81%
	24	456,50	5,70	24%	135,53	19,21	80%
ENVSCORE	4	964,75	2,65	66%	770,12	3,32	83%
	8	609,24	4,20	52%	394,29	6,48	81%
	12	465,31	5,49	46%	273,40	9,35	78%
	16	525,89	4,86	30%	201,76	12,67	79%
	24	413,31	6,18	26%	139,79	18,28	76%
GARP	4	1290,49	3,12	78%	1046,44	3,85	96%
	8	822,33	4,90	61%	531,70	7,57	95%
	12	587,99	6,85	57%	359,05	11,21	93%
	16	487,99	8,25	52%	275,92	14,59	91%
	24	501,21	8,03	33%	186,60	21,58	90%
DG_GARP	4	1112,83	2,90	72%	880,00	3,67	92%
	8	681,28	4,74	59%	453,76	7,11	89%
	12	522,67	6,17	51%	322,43	10,01	83%
	16	424,85	7,59	47%	246,25	13,10	82%
	24	430,63	7,49	31%	163,57	19,73	82%
DG_GARP_BS	4	2285,73	3,11	78%	1878,42	3,78	95%
	8	1225,75	5,80	72%	929,36	7,65	96%
	12	979,31	7,26	60%	654,08	10,87	91%
	16	734,02	9,68	61%	496,45	14,32	89%
	24	677,37	10,49	44%	330,90	21,48	90%

A escalabilidade é um ponto fraco da versão MPI. Para alguns algoritmos, os *speed-ups* crescem levemente até os 24 processos e, para outros algoritmos (como DG_GARP e NICHE_MOSAIC), o valor diminui com 24 processos comparado a 16 processos. Para uma boa escalabilidade, *speed-up* linear é esperado conforme o número p aumenta. Para a versão EasyGrid AMS, esta propriedade é melhor alcançada embora não seja perfeito. No caso do algoritmo NICHE_MOSAIC, a escalabilidade é mais fraca (o *speed-up* é 12 enquanto poderia ser próximo de 24, como acontece para os outros algoritmos) mesmo tendo o *speed-up* 64% melhor que a versão original MPI. Alguma característica do algoritmo não permite uma escalabilidade melhor. Mesmo assim, a eficiência, em termos de utilização dos recursos, é claramente melhor para o EasyGrid AMS uma vez que tem a vantagem devido a sua capacidade de atingir níveis de desempenho mais aceitáveis.

Tabela 8.2: Continuação da Tabela 8.1 (parte 2/2).

		Versão Original com MPI			Versão com EasyGrid AMS		
	p	Tempo (s)	*Speed-up*	Efic.	Tempo (s)	*Speed-up*	Efic.
MAXENT	4	3760,94	3,46	86%	3348,52	3,88	97%
	8	2142,53	6,07	76%	1706,24	7,62	95%
	12	1658,61	7,84	65%	1183,13	10,99	92%
	16	1252,91	10,38	65%	889,87	14,61	91%
	24	1135,80	11,45	48%	599,19	21,70	90%
GARP_BS	4	4274,29	2,98	74%	3304,26	3,85	96%
	8	2347,90	5,42	68%	1675,58	7,60	95%
	12	1696,72	7,50	63%	1162,09	10,96	91%
	16	1345,91	9,46	59%	876,84	14,52	91%
	24	1232,47	10,33	43%	584,80	21,77	91%
SVM	4	7054,70	3,32	83%	6011,95	3,90	97%
	8	4160,92	5,63	70%	3040,21	7,71	96%
	12	2928,81	8,01	67%	2099,60	11,17	93%
	16	2223,67	10,54	66%	1578,37	14,85	93%
	24	1862,63	12,59	52%	1054,04	22,24	93%
RF	4	13020,35	3,28	82%	11896,90	3,59	90%
	8	8552,22	4,99	62%	5977,23	7,14	89%
	12	5353,69	7,97	66%	4120,47	10,35	86%
	16	4201,71	10,15	63%	3096,43	13,78	86%
	24	3775,98	11,30	47%	2072,35	20,58	86%
NICHE_MOSAIC	4	22489,45	3,28	82%	18718,00	3,94	98%
	8	12983,28	5,68	71%	9443,91	7,81	98%
	12	9576,31	7,70	64%	7297,86	10,10	84%
	16	9842,26	7,49	47%	6423,85	11,48	72%
	24	15935,22	4,63	19%	5787,06	12,74	53%

8.4.2 Experimento 2

Neste experimento, a escalabilidade da versão autônoma com o EasyGrid AMS é avaliada. Os algoritmos utilizados são ENVDIST e SVM, sendo que ENVDIST é o mais demorado de todos, chegando a quase 20 dias de execução sequencial. Os números de CPUs p utilizados são 24, 48, 96 e 128.

Primeiramente, para se ter uma ideia do número de tarefas criadas durante a execução da projeção ENM, a quantidade total de tarefas para a instância SVM é de aproximadamente 5400, para todos os números de CPUs utilizados, apresentando baixa variância. Para a instância ENVDIST, o quantidade total de tarefas é de aproximadamente 16000, ara todos os números de CPUs utilizados, apresentando também baixa variância. O valor do número de tarefas não aumenta conforme o aumento de CPUs porque o objetivo do algoritmo é manter a maior quantidade de tarefas possível, independente do número de

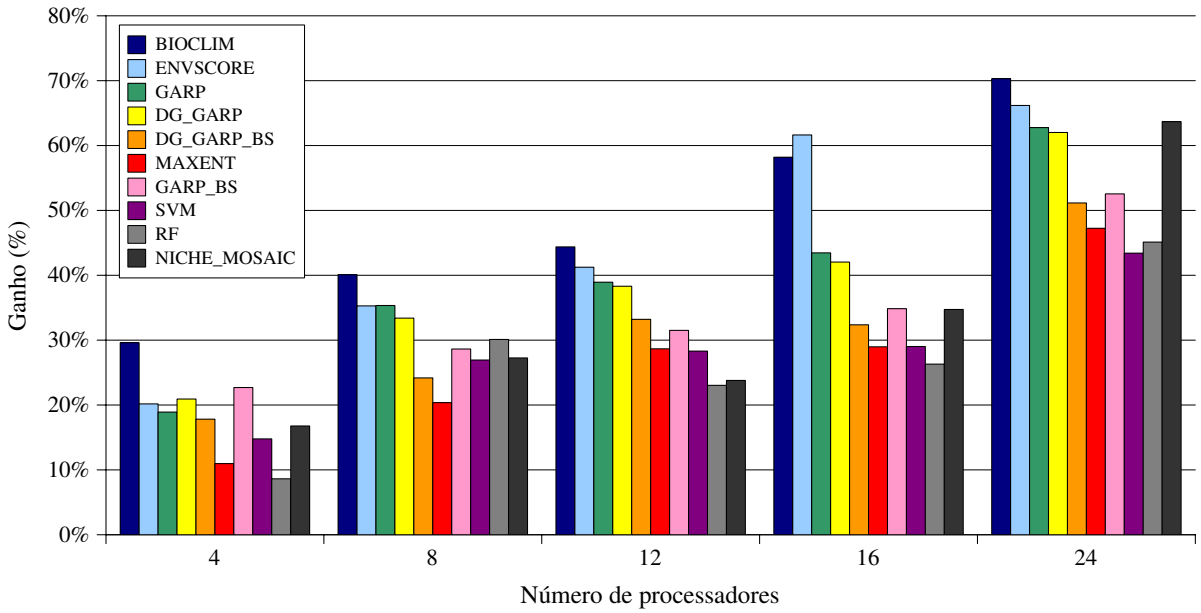


Figura 8.3: Comparação percentual do ganho da versão EasyGrid AMS em relação a versão original MPI.

CPUs, de modo que elas tenham granularidade que sobreponha a sobrecarga de gerenciamento do *middleware*.

A Tabela 8.3 mostra os tempos em segundos de execução da parte *map*, a mais custosa, e a parte *reduce* (concatenação das imagens geradas paralelamente) para o a projeção usando os algoritmos ENVDIST e SVM. Os tempos de *reduce* são próximos entre si para cada algoritmo e parecem não ter relação com o aumento do número p de CPUs para o algoritmo SVM e para o ENVDIST, os valores crescem ligeiramente. O tempo total é a soma entre *map* e *reduce*. Como o *reduce* não é paralelo, o tempo total pode representar uma grande porção em relação ao tempo do *map*.

Tabela 8.3: Tempos em segundos (*map* e *reduce*) obtidos para ENVDIST e SVM.

	ENVDIST		SVM	
p	*map*	*reduce*	*map*	*reduce*
24	70758,88	75,93	1186,28	41,18
48	35316,90	85,26	614,53	59,50
96	17758,27	88,44	313,29	47,33
128	13390,62	98,59	238,36	46,59

O gráfico representado na Figura 8.4 mostra a eficiência de toda a projeção, indicada por toda a barra (parte clara mais parte escura) para cada um dos dois algoritmos. Para cada p, existem duas barras, uma para o ENVDIST e outra para o SVM. A barra completa, somando o tom da cor mais escura com a mais clara, representa a eficiência

de apenas o *map* enquanto a parte mais escura representa a eficiência considerando a projeção por completa (*map* mais *reduce*). A parte clara indica a diferença causa na eficiência adicionando o *reduce*, que não está paralelizado. Para o algoritmo ENVDIST, a eficiência é alta tanto contando com a parte de *reduce* quanto sem ela. O *reduce* pouco prejudica o desempenho. Ao contrário, para o algoritmo SVM, a eficiência é prejudicada, tendo uma grande diferença entre a eficiência com e sem o *reduce*. Uma melhoria seria fazer o *reduce* em paralelo também, usando um algoritmo de redução hierárquica.

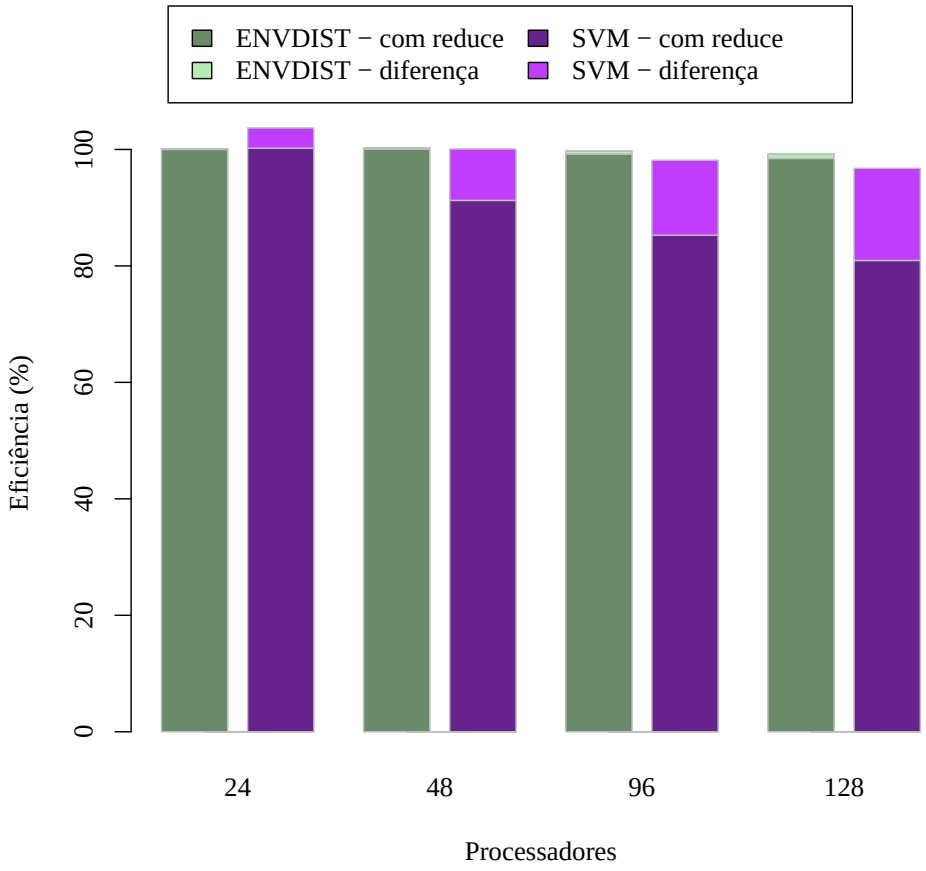


Figura 8.4: Eficiência da projeção para os algoritmos ENVDIST e SVM usando 24, 48, 96 e 128 CPUs.

8.5 Resumo

Neste capítulo foi proposto um novo algoritmo autônomo para um problema de projeção em domínio geoespacial: a projeção do modelo de nicho ecológico. Este tipo de aplicação apresenta as características de granularidade desconhecida e independência entre as tarefas, já que as operações sobre o domínio geoespacial são independentes.

A abordagem do algoritmo original usada é do tipo mestre-trabalhador e realiza um particionamento em blocos fixos possibilitando o balanceamento do trabalho entre tarefas no modelo 1Pproc. Já a nova abordagem do algoritmo autônomo coloca a aplicação no modelo 1Ptask através de um particionamento de tarefas em blocos de tamanho variável autoconfigurado e utiliza o *middleware* EasyGrid AMS para fazer o gerenciamento das tarefas provendo auto-otimização e autorrecuperação a aplicação. Além disso, o algoritmo original centraliza os resultados no mestre, o que gera uma grande sobrecarga, já que os resultados são referentes a pedaços de mapas e apresentam uma grande quantidade de dados. No caso do algoritmo proposto, cada tarefa gera seu próprio sub-mapa em arquivo e não apresenta este gargalo durante a execução. A junção dos sub-mapas é feita ao término da projeção.

Os resultados mostraram que o algoritmo autônomo com o EasyGrid AMS apresentou melhor desempenho em todos os experimentos comparado ao algoritmo original. A versão original MPI apresenta baixa escalabilidade enquanto a execução do algoritmo proposto é melhor escalável na grande maioria dos casos. Com 24 processadores, os *speed-ups* obtidos com a versão EasyGrid AMS chegam a ser de 43% a 70% maior que com a versão original MPI. Além disso, a versão paralela com o *middleware* escala bem com até 128 processadores, obtendo valores de eficiências acima de 94%. A junção dos sub-mapas precisa ser realizada de forma hierárquica futuramente.

Capítulo 9

Conclusão

As aplicações da *e*-ciência, que são muito importantes para a comunidade mundial, geralmente requerem um alto grau de poder computacional para resolver seus problemas científicos. Elas podem executar durante dias, meses ou, até mesmo, anos, e o uso das *e*-infraestruturas é indubitavelmente necessário para satisfazer esta demanda. No entanto, é preciso lidar com a complexidade de gerenciamento das *e*-infraestruturas que apresentam uma larga escala de recursos heterogêneos, compartilhados e suscetíveis a falhas. Além disso, a quantidade de recursos disponíveis pode variar durante a execução das aplicações e, se ainda for considerado o consumo de energia, o sistema torna-se ainda mais dinâmico.

A computação autônoma é vista, nesta tese, como uma solução para melhorar o gerenciamento de sistemas distribuídos de larga escala. Muitos trabalhos introduzem a autonomia em seus sistemas focados em otimizar a utilização de todos os recursos que o compõe e para qualquer tipo de aplicação. No entanto, o uso de um sistema que gerencie aplicações autonomicamente de acordo com suas características internas sugere uma melhor eficiência especialmente para aplicações da *e*-ciência, que requerem larga escala de poder computacional. A tese propõe, portanto, o projeto de aplicações da *e*-ciência autônomas em sistemas distribuídos de larga escala, através de uma metodologia, bem como sua avaliação. Os experimentos realizados com algumas classes de aplicações representativas da *e*-ciência mostraram que a introdução de componentes autônomos, através de um sistema gerenciador de aplicações, traz reais benefícios a essas soluções computacionais de problemas científicos. Portanto, conclui-se que é viável o projeto de aplicações autônomas.

A metodologia do projeto de aplicações autônomas se baseia na bem estabelecida metodologia de projeto de aplicações paralelas PCAM, onde as etapas de *particionamento*, atribuição de *comunicação*, *aglomeração* de tarefas e *mapeamento* são tipicamente utiliza-

das. Na proposta desta tese, a metodologia PCAM é realizada continuamente durante a execução da aplicação e seus parâmetros, como granularidade e número de tarefas, são reconfigurados dinamicamente através de um *middleware* específico, como o EasyGrid AMS que foi utilizado neste trabalho. Além do monitoramento, escalonamento dinâmico de tarefas e mecanismos de detecção e recuperação de falhas, a reconfiguração é integrada ao EasyGrid AMS e à aplicação. Deste modo, a aplicação assume propriedades autônomas de autoconhecimento, auto-otimização, autorrecuperação e autoconfiguração, sendo esta última contribuição da presente tese.

A implementação de aplicações autônomas torna-se mais simples através da metodologia proposta nesta tese, já que nem todas as propriedades autônomas precisam ser realizadas pelo desenvolvedor de controle. Apenas basta construir previamente um modelo de configuração da aplicação baseado no modelo de particionamento 1Ptask e utilizar um *middleware* AMS que seja capaz de tratar o modelo 1Ptask e fornecer as propriedades autônomas supracitadas, como o EasyGrid AMS. O modelo de particionamento tradicional *1Pproc* (1 processo por processador) não permite atingir bons desempenhos em ambientes de execução compartilhados e dinâmicos. Já o modelo de particionamento *1Ptask* (1 processo por tarefa) promove um particionamento diferente gerando uma grande quantidade de tarefas (em relação ao número de processadores) de granularidade fina, o que possibilita que as tarefas da aplicação sejam mais flexíveis e se ajustem ao ambiente através da autoconfiguração. Portanto, a metodologia proposta permitiu o desenvolvimento de aplicações autônomas com menor esforço, uma vez que foi mostrado que a introdução da computação autônoma no desenvolvimento de aplicações da *e*-ciência traz grandes benefícios no desempenho de sua execução em ambientes distribuídos. Ainda, a implementação da autoconfiguração nas aplicações é específica a cada uma, mas soluções apresentadas para determinada aplicação podem ser aproveitadas por outras de mesma classe.

A avaliação e validação da proposta foram conduzidas abordando algumas classes de aplicações de diferentes tipos e para cada uma delas, uma aplicação real da *e*-ciência foi utilizada. *Simulações Monte Carlo* empregam um método comumente utilizado para simular problemas que podem ser representados por processos estocásticos em diversas áreas como engenharia, física, biologia, medicina, finanças e negócios; aplicações de *decomposição de domínio fortemente acoplado* apresentam tarefas com intensa dependência de dados e são utilizadas em áreas como física e astrofísica; a *busca em árvore branch-and-prune* abrange problemas concentrados em áreas científicas como biologia, física, bioquímica, logística e geofísica; *workflows científicos* representam a execução experimental de aplicações de áreas como biologia, botânica e astronomia; e *aplicações de projeção em domínio*

geoespacial são bastante utilizadas em áreas científicas como geofísica, biologia e botânica. As classes foram descritas de acordo com a decomposição de dados, a comunicação e a granularidade das tarefas. O intuito foi analisar a aplicabilidade da abordagem proposta utilizando uma maior gama de problemas com características diversas.

Simulações Monte Carlo são de simples paralelização e apresentam as características de decomposição de domínio, tarefas independentes e granularidade fina. Por serem simples, não precisa de uma etapa de autoconfiguração da aplicação e utiliza apenas o modelo 1Ptask configurado inicialmente e o EasyGrid AMS para auto-otimização e autorrecuperação. Experimentos mostraram que uma aplicação Monte Carlo, chamada *Thermions*, obteve boa eficiência e escalabilidade com o EasyGrid AMS tanto em um cenário dedicado quanto compartilhado, enquanto a versão tradicional é debilitada. Houve uma melhora representativa de até 43%.

Uma aplicação de busca em árvore *branch-and-prune* apresenta decomposição exploratória, dependência entre tarefas e granularidade desconhecida. Os parâmetros granularidade e quantidade de tarefas são dinamicamente ajustados conforme mudanças no sistema, caracterizando um processo de autoconfiguração. A boa eficiência e escalabilidade da versão autônoma com o EasyGrid AMS também foram comprovadas através deste tipo de aplicação. A melhoria significativa da versão autônoma da aplicação chega a ser de 30% em relação à tradicional.

Aplicações com domínio fortemente acoplado precisam de constante comunicação. O algoritmo *Ring* para o problema N-corpos foi alterado, e chamado de *Ring* AMS, através da introdução do modelo 1Ptask, da autoconfiguração e do uso do EasyGrid AMS. Os primeiros resultados mostraram que para uma instância de 1 milhão de partículas, a sobrecarga é baixa, representando apenas 1,2% de todo o processamento. Em um ambiente heterogêneo e compartilhado, os resultados mostram que a autoconfiguração integrada ao escalonamento dinâmico consegue melhorar até 34% em relação a execução com a ausência deles.

Workflows científicos são aplicações diferentes das outras descritas anteriormente no trabalho. Sua decomposição funcional implica em tarefas de granularidade grossa e o fluxo de trabalho é definido por um DAG. Para este tipo de aplicação, a proposta indica apenas o uso do EasyGrid AMS para gerenciar as tarefas autonomicamente. Comparando três tipos diferentes de sistemas gerenciadores (UMS, RMS e AMS), tem-se que a versão AMS é mais eficiente e apresenta um escalonamento mais eficaz devido ao seu afinamento à aplicação.

Aplicações de projeção de informações em domínio geoespacial apresentam as características de decomposição de domínio, granularidade desconhecida e independência entre as tarefas. O emprego da proposta autônoma para este tipo de aplicação implica o uso do modelo *1Ptask*, da autoconfiguração e do EasyGrid AMS como *middleware* auto-otimizável e autorrecuperável. Os resultados dos experimentos realizados com a aplicação de projeção ENM mostraram que o algoritmo autônomo com o EasyGrid AMS é mais eficiente e escalável comparado ao algoritmo original. A melhoria da versão autônoma da aplicação alcança o valor significativo de 70% em relação a sua versão original.

9.1 Trabalhos Futuros

Constantemente, estudos são realizados para identificar a melhor forma de se tratar um determinado tipo de aplicação através do EasyGrid AMS. As versões específicas do *middleware* podem ser alteradas devido a modificações nos sistemas base (como sistema operacional e MPI) ou a novos requisitos do problema da aplicação. Logo, trabalhos futuros podem ser encontrados nesta linha, envolvendo modificações nos *middlewares* específicos.

Enquanto o controle descentralizado é mais escalável comparado às soluções RMSs, a coordenação de ações para obter uma melhor utilização é significantemente mais difícil sem uma visão global dos recursos. Um problema enfrentado pelo uso de AMS está o fato de existir uma disputa pelos recursos entre duas ou mais aplicações autônomas e isto pode atrapalhar a execução delas. Por exemplo, se a funcionalidade de auto-otimização emprega heurísticas gulosas cujo objetivo é executar a aplicação no menor tempo possível, as ações tomadas para uma determinada aplicação podem prejudicar outras. Usando uma regra comportamental simples e adotando metas de tempo de execução para aplicações [101], uma solução inicial permite que múltiplas aplicações executem cordialmente no ambiente distribuído. As aplicações são cientes de que fazem parte de uma sociedade e comportam-se altruistamente quando há folga em suas metas, de modo que todas saiam ganhando em desempenho. No entanto, é preciso incorporar as metas às aplicações avaliadas, incluir e descrever esta nova etapa na metodologia do projeto de aplicações autônomas.

Além das classes de aplicações científicas avaliadas neste trabalho, há inúmeras outras que merecem e necessitam de um projeto que possibilite sua autonomia diante de sistemas distribuídos de larga escala. Em relação a alguma das aplicações avaliadas neste trabalho, existem melhoramentos e experimentos mais confiáveis a serem realizados. O *workflow* de um ENM do Capítulo 7 deveria ser avaliado em um *cluster* maior, criando cenários

de heterogeneidade e compartilhamento. Deste modo, conclusões adicionais poderiam ser realizadas. Além disso, o particionamento dos trabalhos de um *workflow* seguindo o modelo 1Ptask poderia ser estudado. Uma solução seria o emprego de um *workflow* onde cada trabalho do fluxo seria uma aplicação autônoma como, por exemplo, a da projeção ENM desenvolvida nesta tese. Para a aplicação de projeção de um modelo de nicho ecológico descrita no Capítulo 8, a etapa de junção das imagens finais requer uma nova implementação de forma hierárquica. Isto melhoraria o desempenho da execução como um todo, embora não tenha sido o foco do trabalho. A aplicação N-corpos apresentada no Capítulo 6, possui um bom desempenho quando é empregada a maleabilidade e o escalonamento dinâmico do EasyGrid AMS juntos. No entanto, pode ser interessante rever o emprego da maleabilidade em cenários de constante modificações de carga entre as máquinas. Isto acontece porque a etapa de reconfiguração através da maleabilidade é realiza apenas a cada mudança de *time step*, porém um *time step* pode ser longo (cerca de 2000 segundos ou mais) e a execução seria afetada de modo não previsto. Experimentos em um ambiente de escala maior seria apreciado para todas as aplicações.

Referências

[1] Aad, G.; Abajyan, T.; Abbott, B.; Abdallah, J.; Abdel Khalek, S.; Abdelalim, A.; Abdinov, O.; Aben, R.; Abi, B.; Abolins, M., et al. Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc. *Physics Letters B 716*, 1 (2012), 1–29.

[2] Aarseth, S. J. From NBODY1 to NBODY6: The growth of an industry. *Publications of the Astronomical Society of the Pacific 111*, 765 (1999), pp. 1333–1346.

[3] Abramson, D.; Foster, I.; Giddy, J.; Lewis, A.; Sosiĉ, R.; Sutherst, R.; White, N. The nimrod computational workbench: A case study in desktop metacomputing. *Australian Computer Science Communications 19* (1997), 17–26.

[4] Abramson, D.; Sosiĉ, R.; Giddy, J.; Hall, B. Nimrod: a tool for performing parametrised simulations using distributed workstations. In *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on* (1995), IEEE, pp. 112–121.

[5] Amazon Company. Amazon web services. `http://aws.amazon.com/`, Aug. 2014.

[6] Ananth, G.; Gupts, A.; Karypis, G.; Kumar, V. *Introduction to Parallel computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[7] Andreo, P. Monte Carlo techniques in medical radiation physics. *Physics in Medicine and Biology 36*, 7 (1991), 861.

[8] Andronico, G.; Ardizzone, V.; Barbera, R.; Becker, B.; Bruno, R.; Calanducci, A.; Carvalho, D.; Ciuffo, L.; Fargetta, M.; Giorgio, E.; Rocca, G.; Masoni, A.; Paganoni, M.; Ruggieri, F.; Scardaci, D. e-Infrastructures for e-Science: A global view. *Journal of Grid Computing 9*, 2 (2011), 155–184.

[9] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; Zaharia, M. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.

[10] Barker, K. J.; Davis, K.; Hoisie, A.; Kerbyson, D. J.; Lang, M.; Pakin, S.; Sancho, J. C. Entering the petaflop era: The architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 1:1–1:11.

[11] Berman, F.; Chien, A.; Cooper, K.; Dongarra, J.; Foster, I.; Gannon, D.; Johnsson, L.; Kennedy, K.; Kesselman, C.; Mellor-Crummey, J.; Reed, D.; Torczon, L.; Wolski, R. The GrADS project: Software support for

high-level grid application development. *International Journal of High Performance Computing Applications 15*, 4 (2001), 327–344.

[12] Bilmes, J.; Asanovic, K.; Chin, C.-W.; Demmel, J. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing* (New York, NY, USA, 1997), ICS '97, ACM, pp. 340–347.

[13] Binder, K.; Heermann, D. W. *Monte Carlo simulation in statistical physics: an introduction*. Springer, 2010.

[14] Bird, G. Monte-carlo simulation in an engineering context. *Progress in Astronautics and Aeronautics 74* (1981), 239–255.

[15] Boeres, C.; Nascimento, A.; Rebello, V. E. F.; Sena, A. Efficient hierarchical self-scheduling for MPI applications executing in computational Grids. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing* (New York, NY, USA, 2005), ACM Press, pp. 1–6.

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

[17] Boinc Team. BOINC - open-source software for volunteer computing and grid computing. `http://boinc.berkeley.edu/`, Aug. 2014.

[18] Brailsford, S. C.; Potts, C. N.; Smith, B. M. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research 119*, 3 (1999), 557–581.

[19] Bruneton, E.; Coupaye, T.; Leclercq, M.; Quema, V.; Stefani, J.-B. An open component model and its support in java. In *Component-Based Software Engineering*, I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, Eds., vol. 3054 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 7–22.

[20] Buyya, R.; Abramson, D.; Giddy, J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on* (2000), vol. 1, IEEE, pp. 283–289.

[21] Buyya, R.; Venugopal, S. The gridbus toolkit for service oriented grid and utility computing: an overview and status report. In *Grid Economics and Business Models, 2004. GECON 2004. 1st IEEE International Workshop on* (Apr. 2004), pp. 19–66.

[22] Carrier, J.-F.; Archambault, L.; Beaulieu, L.; Roy, R. Validation of GEANT4, an object-oriented Monte Carlo toolkit, for simulations in medical physics. *Medical physics 31*, 3 (2004), 484–492.

[23] Chopra, I.; Singh, M. Analysing the need for autonomic behaviour in grid computing. In *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on* (Feb. 2010), vol. 1, pp. 535–539.

[24] Chrabakh, W.; Wolski, R. GrADSAT: A parallel sat solver for the grid. In *Proceedings of IEEE SC03* (2003).

[25] Cirne, W.; Marzullo, K. Open grid: A user-centric approach for grid computing. In *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing* (2001), Sep.

[26] CRIA. SpeciesLink. `http://www.splink.org.br/`, Aug. 2014.

[27] Cui, Y.; Olsen, K.; Jordan, T.; Lee, K.; Zhou, J.; Small, P.; Roten, D.; Ely, G.; Panda, D. K.; Chourasia, A.; Levesque, J.; Day, S. M.; Maechling, P. Scalable earthquake simulation on petascale supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for* (Nov 2010), pp. 1–20.

[28] da Silva, J. A. *Tolerância a Falhas para Aplicações Autônomas em Grades Computacionaiss*. Tese de Doutorado, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, 2010.

[29] da Silveira Filho, O. T. *Dispersão Térmica em Meios Porosos Periódicos. Um Estudo Numérico*. Tese de Doutorado, Instituto Politécnico do Rio de Janeiro, IPRJ/UERJ, RJ, Brasil, 2001.

[30] Deelman, E.; Blythe, J.; Gil, Y.; Kesselman, C.; Mehta, G.; Vahi, K.; Blackburn, K.; Lazzarini, A.; Arbree, A.; Cavanaugh, R.; Koranda, S. Mapping abstract complex workflows onto grid environments. *J. Grid Comput. 1*, 1 (2003), 25–39.

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

[32] Dobson, S.; Sterritt, R.; Nixon, P.; Hinchey, M. Fulfilling the vision of autonomic computing. *Computer 43*, 1 (Jan. 2010), 35–41.

[33] Dongarra, J.; Beckman, P.; Aerts, P.; Cappello, F.; Lippert, T.; Matsuoka, S.; Messina, P.; Moore, T.; Stevens, R.; Trefethen, A.; Valero, M. The international exascale software project: a call to cooperative action by the global high-performance community. *International Journal of High Performance Computing Applications 23*, 4 (2009), 309–322.

[34] Dongarra, J.; *et al.*, P. B. The international exascale software project RoadMap. *International Journal of High Performance Computing Applications 25*, 1 (2011), 3–60.

[35] Drummond, L. M. A.; Uchoa, E.; Gonçalves, A. D.; Silva, J. M. N.; Santos, M. C. P.; de Castro, M. C. S. A grid-enabled distributed branch-and-bound algorithm with application on the steiner problem in graphs. *Parallel Comput. 32*, 9 (Oct. 2006), 629–642.

[36] El Maghraoui, K.; Desell, T. J.; Szymanski, B. K.; Varela, C. A. Malleable iterative mpi applications. *Concurrency and Computation: Practice and Experience 21*, 3 (2009), 393–413.

[37] ESRI. ArcGIS. `http://www.arcgis.com/`, Aug. 2014.

[38] Esselink, K.; Loyens, L. D. J. C.; Smit, B. Parallel Monte Carlo simulations. *Phys. Rev. E 51* (Feb. 1995), 1560–1568.

[39] EUBrazilOpenBio Team. EU-Brazil open data and cloud computing e-infrastructure for biodiversity. `http://www.eubrazilopenbio.eu/`, Aug. 2014.

[40] Evangelinos, C.; Hill, C. N. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's EC2. *Ratio 2*, 2.40 (2008), 2–34.

[41] Feitelson, D. G.; Rudolph, L. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing* (1996), Springer, pp. 1–26.

[42] Fernandez-Baca, D. Allocating modules to processors in a distributed system. *Software Engineering, IEEE Transactions on 15*, 11 (Nov 1989), 1427–1436.

[43] FoldingAtHome Team. FoldingAtHome distributed computing. `http://folding.stanford.edu/`, Aug. 2014.

[44] Foster, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[45] Foster, I.; Kesselman, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications 11* (1996), 115–128.

[46] Foster, I.; Kesselman, C. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann, 2004.

[47] Foster, I.; Zhao, Y.; Raicu, I.; Lu, S. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08* (Nov. 2008), pp. 1–10.

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

[49] Frigo, M. A fast fourier transform compiler. *SIGPLAN Not. 39*, 4 (Apr. 2004), 642–655.

[50] Ganek, A. G.; Corbi, T. A. The dawning of the autonomic computing era. *IBM Syst. J. 42*, 1 (Jan. 2003), 5–18.

[51] Garlan, D.; Schmerl, B. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems* (New York, NY, USA, 2002), WOSS '02, ACM, pp. 27–32.

[52] GBIF. Global biodiversity information facility. `http://www.gbif.org/`, Aug. 2014.

[53] Geer, D. Chip makers turn to multicore processors. *Computer 38*, 5 (May 2005), 11–13.

[54] Gendron, B.; Crainic, T. G. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations Research 42*, 6 (1994), 1042–1066.

[55] Giersz, M.; Heggie, D. C. Statistics of N-Body simulations. II. equal masses after core collapse. *Monthly Notices of the Royal Astronomical Society 268* (1994), 257–275.

[56] Globus Team. The globus toolkit. `http://www.globus.org/toolkit/`, Aug. 2014.

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

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

[59] Hey, T.; Trefethen, A. E. The UK e-Science core programme and the grid. *Future Gener. Comput. Syst. 18*, 8 (Oct. 2002), 1017–1031.

[60] Hofstee, H. P. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2005), HPCA '05, IEEE Computer Society, pp. 258–262.

[61] Horn, P. Autonomic computing: IBM's perspective on the state of information technology. *Computing Systems 15*, Jan (2001), 1–40.

[62] HTCondor. DAGMan Application. `http://research.cs.wisc.edu/htcondor/dagman/dagman.html/`, Aug. 2014.

[63] Huebscher, M. C.; McCann, J. A. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv. 40*, 3 (Aug. 2008), 7:1–7:28.

[64] IBM. An architectural blueprint for autonomic computing. Tech. rep., Hawthorne, NY, 2004.

[65] INPE. INPE - image catalog. `http://www.dgi.inpe.br/CDSR/`, Aug. 2014.

[66] Jäckel, P.; Bubley, R. *Monte Carlo methods in finance.* J. Wiley, 2002.

[67] Jeffers, J.; Reinders, J. *Intel Xeon Phi Coprocessor High Performance Programming.* Elsevier Science, 2013.

[68] Juve, G.; Deelman, E.; Berriman, G. B.; Berman, B. P.; Maechling, P. An evaluation of the cost and performance of scientific workflows on amazon EC2. *Journal of Grid Computing 10* (2012), 5–21.

[69] Kacsuk, P.; Sipos, G. Multi-grid, multi-user workflows in the P-GRADE grid portal. *Journal of Grid Computing 3* (2005), 221–238.

[70] Kennedy, K.; Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[71] Kephart, J. O. Research challenges of autonomic computing. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on* (May 2005), pp. 15 – 22.

[72] Kephart, J. O.; Chess, D. M. The vision of autonomic computing. *Computer 36*, 1 (2003), 41–50.

[73] Kish, L. B. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A 305*, 3-4 (2002), 144–149.

[74] Krauter, K.; Buyya, R.; Maheswaran, M. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience 32*, 2 (2002), 135–164.

[75] Kurzak, J.; Buttari, A.; Luszczek, P.; Dongarra, J. The playstation 3 for high-performance scientific computing. *Computing in Science and Engineering 10*, 3 (2008), 84–87.

[76] Laure, E.; Gr, C.; Fisher, S.; Frohner, A.; Kunszt, P.; Krenek, A.; Mulmo, O.; Pacini, F.; Prelz, F.; White, J.; Barroso, M.; Buncic, P.; Byrom, R.; Cornwall, L.; Craig, M.; Meglio, A. D.; Djaoui, A.; Giacomini, F.; Hahkala, J.; Hemmer, F.; Hicks, S.; Edlund, A.; Maraschini, A.; Middleton, R.; Sgaravatto, M.; Steenbakkers, M.; Walk, J.; Wilson, A. Programming the Grid with gLite. *Computational Methods in Science and Technology 12*, 1 (2006), 33–45.

[77] Lavor, C.; Liberti, L.; Maculan, N. Molecular distance geometry problem. In *Encyclopedia of Optimization*, C. A. Floudas and P. M. Pardalos, Eds. Springer, 2009, pp. 2304–2311.

[78] Lavor, C.; Liberti, L.; Maculan, N.; Mucherino, A. The discretizable molecular distance geometry problem. *Computational Optimization and Applications 52*, 1 (2012), 115–146.

[79] Makino, J. An efficient parallel algorithm for $O(N^2)$ direct summation method and its variations on distributed-memory parallel machines. *New Astronomy 7*, 7 (2002), 373–384.

[80] Mandal, A.; Kennedy, K.; Koelbel, C.; Marin, G.; Mellor-Crummey, J.; Liu, B.; Johnsson, L. Scheduling strategies for mapping application workflows onto the grid. In *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on* (2005), pp. 125–134.

[81] Marco, C.; Fabio, C.; Alvise, D.; Antonia, G.; Francesco, G.; Alessandro, M.; Moreno, M.; Salvatore, M.; Fabrizio, P.; Luca, P.; Francesco, P. The glite workload management system. In *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing* (Berlin, Heidelberg, 2009), GPC '09, Springer-Verlag, pp. 256–268.

[82] Milosavljević, M.; Merritt, D. Formation of galactic nuclei. *The Astrophysical Journal 563*, 1 (2001), 34.

[83] MPI Forum. Message passing interface forum. `http://www.mpi-forum.org/`, Aug. 2014.

[84] Munoz, M.; Giovanni, R.; Siqueira, M.; Sutton, T.; Brewer, P.; Pereira, R.; Canhos, D.; Canhos, V. OpenModeller: a generic approach to species' potential distribution modelling. *Geoinformatica* (Aug. 2009).

[85] Nascimento, A.; Sena, A.; Boeres, C.; Rebello, V. E. F. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience 19*, 14 (Sept. 2007), 1955–1974. Published Online: 14 Nov 2006.

[86] Nascimento, A.; Sena, A.; da Silva, J.; Vianna, D. Q. C.; Boeres, C.; Rebello, V. E. F. On the advantages of an alternative MPI execution model for grids. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid* (Rio de Janeiro, Brazil, 2007), IEEE Computer Society, pp. 575–582.

[87] Nelson, Y. L.; Bansal, B.; Hall, M.; Nakano, A.; Lerman, K. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. *Parallel and Distributed Processing Symposium, International 0* (2008), 1–8.

[88] Nickolls, J.; Dally, W. J. The gpu computing era. *IEEE micro 30*, 2 (2010), 56–69.

[89] Oak Ridge National Laboratory. Introducing titan: Advancing the era of accelerated computing. `http://www.olcf.ornl.gov/titan/`, Aug. 2014.

[90] Oinn, T.; Addis, M.; Ferris, J.; Marvin, D.; Senger, M.; Greenwood, M.; Carver, T.; Glover, K.; Pocock, M. R.; Wipat, A.; Li, P. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics 20*, 17 (Nov. 2004), 3045–3054.

[91] Oliveira, F. G.; Rebello, V. E. F. Algoritmos branch-and-prune autônomos. In *SBRC '10, XXVIII Simpósio Brasileiro de Rede de Computadores e Sistemas Distribuídos* (Gramado, RS, 2010).

[92] Oliveira, F. G.; Rebello, V. E. F. Aplicações Autônomas + Sistemas Simples = Futuro Feliz? In *WoSida'11: Proceedings of the I Workshop on Autonomic Distributed Systems on XXIX SBRC* (Campo Grande, Mato Grosso do Sul, Brazil, 2011).

[93] OpenModeller Team. Openmodeller webpage. `http://openmodeller.sourceforge.net/`, Aug. 2014.

[94] Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[95] Peterson, A. T.; Soberón, J.; Pearson, R. G.; Anderson, R. P.; Martínez-Meyer, E.; Nakamura, M.; Araújo, M. B. *Ecological niches and geographic distributions (MPB-49).* Princeton University Press, 2011.

[96] Project, C. Basic features of the grid component model (assessed). Tech. rep., 2007. Deliverable D.PM.04.

[97] QGIS Project. QGIS - A Free and Open Source Geographic Information System. `http://www.qgis.org/`, Aug. 2014.

[98] Rambaut, A.; Grass, N. C. Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Computer applications in the biosciences: CABIOS 13*, 3 (1997), 235–238.

[99] Randall, A. L.; Walter, R. C. Overview of the small unit operations situational awareness system. In *Military Communications Conference, 2003. MILCOM 2003. IEEE* (2003), vol. 1, IEEE, pp. 169–173.

[100] Research Collaboratory for Structural Bioinformatics. RCSB Protein Data Bank. `http://www.rcsb.org/`, Aug. 2014.

[101] Rodrigues, H. Grid S.A.: Uma sociedade autônoma. Master's thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, 2009.

[102] Roure, D. D.; Goble, C.; Bhagat, J.; Cruickshank, D.; Goderis, A.; Michaelides, D.; Newman, D. myexperiment: Defining the social virtual research environment. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience* (Washington, DC, USA, Dec. 2008), ESCIENCE '08, IEEE Computer Society, pp. 182–189.

[103] Saïdi, H.; Levy, B. D. J.; Valdes, A. Self-regenerative software components. In *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems: in association with 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), SSRS '03, ACM, pp. 115–120.

[104] Sanjay, H. A.; Vadhiyar, S. S. A strategy for scheduling tightly coupled parallel applications on clusters. *Concurrency and Computation: Practice and Experience 21*, 18 (2009), 2491–2517.

[105] Schaller, R. R. Moore's law: past, present and future. *Spectrum, IEEE 34*, 6 (1997), 52–59.

[106] Seiler, L.; Carmean, D.; Sprangle, E.; Forsyth, T.; Abrash, M.; Dubey, P.; Junkins, S.; Lake, A.; Sugerman, J.; Cavin, R.; Espasa, R.; Grochowski, E.; Juan, T.; Hanrahan, P. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 18:1–18:15.

[107] Shoeb, M.; Hussain, A.; Bikas, M.; Hasan, M. An extended algorithm to enhance the performance of the gridbus broker with data restoring technique. In *Computer Engineering and Technology, 2009. ICCET '09. International Conference on* (Jan 2009), vol. 1, pp. 371–375.

[108] Si, M.; Ishikawa, Y.; Tatagi, M. Direct mpi library for intel xeon phi co-processors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International* (May 2013), pp. 816–824.

[109] Silva, J.; Rebello, V. E. F. Low Cost Self-healing in MPI Applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting* (2007), Springer, pp. 144–152.

[110] Souto, H. P. A.; da Silveira, O. T.; Moyne, C.; Didierjean, S. Thermal dispersion in porous media: computations by the random walk method. *Computational and applied mathematics 21* (2002), 513–544.

[111] Stokes-Ress, I.; Baude, F.; Doan, V.-D.; Bossy, M. Managing parallel and distributed Monte Carlo simulations for computational finance in a grid environment. In *Grid Computing*, S. Lin and E. Yen, Eds. Springer US, 2009, pp. 183–204.

[112] Tannenbaum, T.; Wright, D.; Miller, K.; Livny, M. Condor – a distributed job scheduler. In *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, Oct. 2001.

[113] Tannenbaum, T.; Wright, D.; Miller, K.; Livny, M. Condor – a distributed job scheduler. In *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, Oct. 2001.

[114] Tejedor, E.; Badia, R. Comp superscalar: Bringing grid superscalar and gcm together. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on* (May 2008), pp. 185–193.

[115] Top500.org. Top 500 supercomputer sites. `http://top500.org/`, Aug. 2014.

[116] Topcuoglu, H.; Hariri, S.; Wu, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on 13*, 3 (Mar 2002), 260–274.

[117] Truszkowski, W.; Hinchey, M.; Rash, J.; Rouff, C. Nasa's swarm missions: the challenge of building autonomous software. *IT Professional 6*, 5 (Oct. 2004), 47–52.

[118] van Nieuwpoort, R. V.; Kielmann, T.; Bal, H. E. User-friendly and reliable grid computing based on imperfect middleware. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07)* (Nov. 2007).

[119] Venugopal, S.; Buyya, R.; Winton, L. A grid service broker for scheduling e-science applications on global data grids. *Concurrency and Computation: Practice and Experience 18*, 6 (2006), 685–699.

[120] Vogelsberger, M.; Genel, S.; Springel, V.; Torrey, P.; Sijacki, D.; Xu, D.; Snyder, G.; Bird, S.; Nelson, D.; Hernquist, L. Properties of galaxies reproduced by a hydrodynamic simulation. *Nature 509*, 7499 (2014), 177–182.

[121] Wang, H. *Monte Carlo Simulation with Applications to Finance*. CRC Press, 2012.

[122] Weil, G.; Heus, K.; Faraut, T.; Demongeot, J. The cyclic genetic code as a constraint satisfaction problem. *Theoretical Computer Science 322*, 2 (2004), 313–334. Discrete Applied Problems - Florilegium for E. Goles.

[123] Weiss, A. Computing in the clouds. *netWorker 11*, 4 (Dec. 2007), 16–25.

[124] Whaley, R. C.; Petitet, A.; Dongarra, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2 (2001), 3–35.

[125] WLCG Team. Worldwide LHC computing grid. `http://wlcg.web.cern.ch/`, Aug. 2014.

[126] Wolfe, M. J. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

[127] Worldclim. Worldclim - global climate data. `http://www.worldclim.org/`, Aug. 2014.

[128] Wulf, W. A.; Johnsson, R. K.; Weinstock, C. B.; Hobbs, S. O.; Geschke, C. M. *The Design of an Optimizing Compiler*. Elsevier Science Inc., New York, NY, USA, 1975.

[129] Xiong, J.; Johnson, J.; Johnson, R.; Padua, D. Spl: a language and compiler for dsp algorithms. *SIGPLAN Not. 36*, 5 (May 2001), 298–308.

[130] Yang, C.; Wang, F.; Du, Y.; Chen, J.; Liu, J.; Yi, H.; Lu, K. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing* (Washington, DC, USA, 2010), CLUSTER '10, IEEE Computer Society, pp. 19–28.

[131] Yu, J.; Buyya, R. A novel architecture for realizing grid workflow using tuple spaces. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on* (2004), pp. 119–128.

[132] Yu, J.; Buyya, R. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec. 34*, 3 (Sept. 2005), 44–49.

[133] Yu, J.; Buyya, R.; Ramamohanarao, K. Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, F. Xhafa and A. Abraham, Eds., vol. 146 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, 2008, pp. 173–214.