Universidade Federal Fluminense

Fernanda Gonçalves de Oliveira

Aplicações Autônomas para Computação em Larga Escala

NITERÓI

2010

Universidade Federal Fluminense

Fernanda Gonçalves de Oliveira

Aplicações Autônomas para Computação em Larga Escala

Dissertação de **Mestrado** *submetida* ao "Programa de Pós-Graduação em Computação" da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Processamento Paralelo e Distribuído.

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

NITERÓI

2010

Aplicações Autônomas para Computação em Larga Escala

Fernanda Gonçalves de Oliveira

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

Aprovada por:

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

Prof. Célio Vinicius Neves de Albuquerque, Ph.D. / IC-UFF

Prof. Carlile Campos Lavor, D.Sc. / IMECC-UNICAMP

Niterói, 9 de abril de 2010.

"*Do not seek to follow in the footsteps of the men of old; seek what they sought.*"
(Matsuo Basho)

Aos meus pais, avós, irmã e amigos, que dão significado a todo o meu trabalho.

Agradecimentos

Agradeço primeiramente a Deus por me permitir chegar até este momento importante de minha vida.

Agradeço ao professor Vinod Rebello pelos seus ensinamentos e incentivo, à equipe do Projeto EasyGrid, desenvolvedores do EasyGrid SGA e aos professores do Instituto de Computação pelo imensurável conhecimento transmitido.

Agradeço também aos meus grandes amigos, principalmente a Diego Passos pela sua paciência, companhia e carinho, a Rebeca Lese e Ariela Souza pela ajuda em minha decisão de iniciar o doutorado e também pela grande amizade.

Finalmente, agradeço a minha família, especialmente aos meus pais, Aloisio e Estela, que mesmo não estando todo o tempo juntos a mim, estão sempre me apoiando.

Resumo

Este trabalho descreve uma estratégia para a paralelização de algoritmos do tipo *branch-and-prune* e *branch-and-bound* em ambientes distribuídos compartilhados e dinâmicos. Estas técnicas exaustivas são bastante utilizadas por aplicações de diversas áreas, como bioquímica, física e logística. Enquanto essas aplicações geralmente requerem uma grande quantidade de poder computacional, elas podem ser particionadas em sub-tarefas independentes e executadas em paralelo. No entanto, a distribuição da computação destas tarefas não é trivial já que elas não são conhecidas a *priori*. Além disso, ambientes computacionais distribuídos estão se tornando cada vez mais complexos e dinâmicos devido à colaboração e ao compartilhamento. A estratégia lida com estes problemas tornando aplicações *branch-and-prune* mais autônomas e portanto mais capazes de tirar proveito de ambientes computacionais dinâmicos e de larga escala eficientemente.

Palavras-chave: Algoritmos *Branch-and-Prune*, Computação em Grades, Computação Autônoma, Algoritmos *Branch-and-Bound*.

Abstract

This work describes a strategy to parallelize branch-and-prune and branch-and-bound based algorithms for shared dynamic distributed environments. These exhaustive search techniques are often required by applications from many areas, such as biochemistry, physics and logistics. While these applications typically demand huge quantities of computational power, they can be partitioned in independent sub-tasks and executed in parallel. However, the distribution of the non-uniform computational workloads of these tasks is not trivial since they may not be known a *priori*. Moreover, large scale distributed computing execution environments are becoming more and more complex and dynamic due to their collaborative and shared natures. The strategy addresses this problems by making branch-and-prune applications autonomic and thus better able to take advantage of these dynamic computing environments efficiently.

Keywords: Branch-and-Prune Algorithms, Grid Computing, Autonomic Computing, Branch-and-Bound Algorithms.

Palavras-chave

1. Algoritmos *Branch-and-Prune*.
2. Computação em Grades.
3. Computação Autônoma.
4. Algoritmos *Branch-and-Bound*.

Abreviações

AMS	:	*Application Management System*
BoT	:	*Bag-of-Tasks*
FLOPS	:	*FLoating point Operations Per Second*
GG	:	Gerenciador Global
GM	:	Gerenciador de Máquina
GrADS	:	*Grid Analysis and Display System*
GRAM	:	*Globus Resource Allocation Manager*
GS	:	Gerenciador de *Site*
GSI	:	*General Information about Security*
MPI	:	*Message Passing Interface*
PDB	:	*Protein Data Bank*
PDGDM	:	Problema Discreto de Geometria das Distâncias em Moléculas
PGDM	:	Problema da Geometria das Distâncias em Moléculas
RMN	:	Ressonância Magnética Nuclear
RMS	:	*Resource Management System*
SGA	:	Sistemas Gerenciadores de Aplicação
SGR	:	Sistema Gerenciador de Recursos

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Existem inúmeros problemas de várias áreas de pesquisa e desenvolvimento cujas soluções requerem um alto grau de processamento e memória geralmente disponível em uma instituição de pesquisa. Centros de supercomputação são núcleos de processamento de dados projetados especialmente para hospedar os melhores sistemas de computação no mundo. Porém, os custos de computação são altos devido não só ao preço de adquirir o sistema computacional mas também ao preço associado ao fornecimento de energia e refrigeração. Infelizmente, estes custos tornam tais sistemas computacionais pouco acessíveis para muitos cientistas e pesquisadores. As aplicações utilizadas para resolver tais problemas são, muitas vezes, chamadas de aplicações de larga escala, ou especificamente de tera, peta ou exa escala [12, 39], dependendo da quantidade de desempenho ou de memória necessários - por exemplo, TeraFLOPS/TeraBytes, PetaFLOPS/PetaBytes, etc.

Soluções eficientes requerem o desenvolvimento de aplicações especialmente de larga escala que podem ser classificadas em dois grupos. Algumas dessas aplicações são chamadas *data-intensive* pois são dependentes da análise e cálculo de uma grande volume de dados. Neste tipo de aplicações, muita memória é necessária e boa parte do tempo de processamento é devido à grande quantidade de dados a ser acessada e tratada. A área de mineração de dados possui bons exemplos de aplicações *data-intensive* que são de larga-escala como mineração em dados biológicos e mineração de texto [32]. Outro exemplo de aplicações com alta demanda de memória pode ser encontrado na área de física. São aplicações usadas para adquirir conhecimento pela análise de dados gerados pelo acelerador de partículas *Large Hardron Colider* [71], que se encontra no CERN (*Organisation Européenne pour la Recherche Nucléaire*), uma organização de pesquisas nucleares na Europa [14]. Este acelerador de partículas chega a gerar cerca de 1PBytes de dados por ano, e a tarefa de processar e analisar os dados requer a disponibilidade de bastante armazenamento e processamento para tratar os dados em um tempo razoável. Existem ainda aplicações de larga-escala que são chamadas *cpu-intensive*, onde a maior parte do processamento é baseada em relativamente pouca quantidade de dados que pode

estar toda em memória principal. Neste caso, o elevado tempo de processamento deve-se principalmente à grande quantidade de operações a serem efetuadas.

Na área de otimização combinatória [41], onde a maioria dos problemas são NP-difíceis [33], aplicações *cpu-intensive* na forma de meta-heurísticas são utilizadas para buscar uma boa solução de um dado problema. O problema de Steiner em grafos [20], usado em diversas aplicações na área de redes de computadores como a distribuição de conteúdo multimídia e teleconferências, e o Problema Discreto de Geometria das Distâncias em Moléculas (PDGDM) [44], usado para determinar a estrutura tridimensional de proteínas dos organismos vivos, são exemplos de problemas que necessitam intensivamente de processamento em CPUs. Além disso, na área de simulações, existem diversas aplicações de larga escala *cpu-intensive*. O projeto *Folding@home* [62, 69] tem o objetivo simular e compreender o "dobramento" das proteínas (*protein folding* - um desafio na área biológica), a falha no dobramento e as doenças relacionadas. Demora-se cerca de um dia para simular um nanosegundo de dobramento de uma proteína, sendo que as proteínas enrolam-se numa escala de dez microsegundos (10.000 nanosegundos). Para realizar isto em um tempo razoável, é necessária uma grande disponibilidade de processamento de um ambiente computacional distribuído [11]. Ainda na área de simulações, existe o clássico problema de simular aproximadamente a evolução de um sistema de corpos em que cada corpo continuamente interage com todos os outros corpos do sistema [1]. Aplicações na área astrofísica utilizam este problema para estudar a formação de constelações e até de galáxias. Como nestas aplicações a escala de entrada do problema é relativamente grande, é necessário o uso de bastante processamento computacional [31, 54].

Atualmente, existem diversos tipos de sistemas distribuídos para a execução de aplicações de larga escala [34], alguns com componentes construídos especificamente para computação em alto desempenho e outros simplesmente adaptados para este fim. Supercomputadores são sistemas de alto desempenho [30], alguns já capazes de alcançar capacidades de mais de 1 PetaFLOPS, como é o caso do *Blue Gene/P* [67], um grupo de diversos supercomputadores com cerca de 4096 processadores por *rack* [37]. No entanto, estes sistemas são caros para se adquirir e manter, tornando-se de difícil acesso para a maior parte da comunidade científica. Por outro lado, grades de computadores são aglomerações de recursos computacionais heterogêneos geograficamente distribuídos e tipicamente interconectados por uma rede compartilhada [22]. Podem compor uma grade recursos como PCs, *clusters* e até supercomputadores. Seus princípios de colaboratividade e compartilhamento de recursos entre instituições tornam este sistema potencialmente escalável (em relação ao aumento de poder computacional) especialmente para aplicações que necessitam de pouca comunicação entre processos paralelos. No entanto, tais ambientes de larga escala, homogêneos ou não, são mais suscetíveis a variações de poder computacional, banda e latência na rede, a falhas de recursos, por exemplo. Hoje em dia,

existem diversas ferramentas de gerenciamento de recursos e aplicações em grades computacionais que reduzem o esforço do desenvolvedor, tratando a maioria dos problemas comuns neste tipo de ambiente [24, 36, 5, 4].

Mais recentemente, um outro tipo de infraestrutura para sistemas distribuídos vem atraindo notícias - a computação nas nuvens (*cloud computing*) [23]. Seu objetivo é fornecer um ambiente abstrato, virtualizado, dinamicamente escalável, com gerenciamento do poder computacional, armazenamento, plataformas e outros serviços que serão entregues sob demanda para consumidores externos através da Internet. Hoje, várias companhias, como por exemplo Amazon [9], vendem poder computacional e armazenamento, tirando a necessidade de pesquisadores e instituições serem donos de seus próprios recursos. No entanto, seus usuários raramente possuem garantias na utilização de recursos como a comunicação e poder computacional.

Como pode ser visto, os ambientes distribuídos capazes de suportar aplicações de larga escala estão se tornando cada vez mais diversificados e complexos para se tirar proveito. Isto ocorre porque eles contem diferentes tipos de computadores multiprocessados (com processadores multicores) que se tornam mais dinâmicos, devido ao compartilhamento, e mais suscetíveis a falhas, devido a sua escala. Mais e mais a infraestrutura que compõe estes ambientes precisa de mecanismos sofisticados de gerenciamento para garantir o funcionamento eficiente do sistema e das aplicações. Isto é especialmente importante, dado que boa parte dos desenvolvedores de aplicações distribuídas geralmente não são aptos a lidar com a complexidade e peculiaridade associados a estas novas classes de ambientes. Além disso, a forma eficiente de programação paralela já não pode ser a mesma usada para se programar em *clusters* de computadores com uma pequena quantidade de recursos. A quantidade de computadores nos sistemas distribuídos atuais pode alcançar centenas e até milhares de unidades, o que dificulta a atividade de programação paralela. O que, há alguns anos, era uma atividade comum a um cientista que lida com aplicações paralelas, hoje em dia, é uma tarefa mais complicada devido a dificuldade de entender como dividir a aplicação dado que existem consideravelmente mais recursos disponíveis porém heterogêneos e dinâmicos.

A computação autônoma aparece como resposta para o problema do gerenciamento de aplicações ou sistemas de larga escala [48, 40, 65]. Tornando a aplicação auto-gerenciável, ela é capaz de reagir às constantes mudanças do ambiente (auto-configuração), de detectar falhas e autorrecuperar-se (autorrecuperação), de proteger seus dados e sua execução (auto-proteção), de melhorar sua execução (auto-otimização), estando ciente do seu estado e do ambiente de execução. Desta maneira, a aplicação consegue adaptar-se ao ambiente sem a necessidade da interferência do usuário, garantindo um melhor desempenho.

1.1 Objetivos

Os objetivos deste trabalho são investigar o projeto de aplicações autônomas e a habilidade da computação autônoma lidar com a complexidade de ambientes de larga escala. Para alcançar tais objetivos, este trabalho propõe uma estratégia de paralelização de algoritmos *branch-and-prune* que possibilite uma maior autonomia à aplicação em um ambiente de larga escala através de um gerenciador de aplicações chamado EasyGrid AMS [52, 61]. O *middleware* EasyGrid AMS é um sistema gerenciador de aplicações que oferece meios da aplicação autogerenciar-se, fazendo com que a aplicação adapte-se ao seu ambiente de execução eficientemente.

Esta mesma estratégia de paralelização pode ser usada em algoritmos *branch-and-bound*, já que tais algoritmos buscam pela solução ótima do problema percorrendo uma estrutura de árvore bastante semelhante à estrutura do algoritmo *branch-and-prune*. Existem poucas diferenças na forma como esses algoritmos de ramificação são implementados.

A avaliação da proposta é realizada através de dois estudos de caso da classe de algoritmos *branch-and-prune*:

- N-rainhas [21] - uma aplicação da área da computação. Seu objetivo é encontrar todas as soluções do problema. Uma solução corresponde a dispor N rainhas em um tabuleiro $N \times N$ de forma que elas não se ataquem conforme as regras de xadrez.
- PDGDM [44] - Problema Discreto de Geometria das Distâncias em Moléculas (aplicação da área de bioquímica). Seu objetivo é encontrar possíveis estruturas tridimensionais de moléculas de proteínas considerando informações disponibilizadas pela Ressonância Magnética Nuclear (RMN).

1.2 Organização do Trabalho

O restante do trabalho está organizado da seguinte maneira: no Capítulo 2 os trabalhos relacionados serão apresentados assim como o middleware EasyGrid AMS, a ferramenta utilizada neste trabalho. A proposta da estratégia de paralelização é explicada no Capítulo 3 detalhando os mecanismos utilizados para se fazer a paralelização dos algoritmos. A avaliação é apresentada no Capítulo 4. O último capítulo corresponde à conclusão e aos trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Muitas aplicações utilizam algoritmos de *branch-and-prune* ou *branch-and-bound* [43, 45]. Como geralmente elas demandam um longo tempo de execução em um único processador, opta-se por utilizar uma quantidade maior de recursos e paralelizar a aplicação entre eles. No entanto, a carga de trabalho atribuída a cada processador dificilmente é distribuída igualmente devido à natureza do algoritmo. Além disso, os ambientes distribuídos nem sempre são homogêneos e dedicados, o que ocasiona a variação do poder computacional de cada processador no tempo.

O objetivo deste trabalho é tornar aplicações *branch-and-prune* capazes usufruir de ambientes de larga escala eficientemente. O que vem sendo feito nesta área em trabalhos recentes encontra-se no uso de estratégias de balanceamento de carga, algumas somadas a mecanismos de tolerância a falhas, para distribuir o trabalho deste tipo de aplicação entre os diversos processadores heterogêneos conectados por diversos tipos de redes.

Neste capítulo, será feita uma descrição do tipo de algoritmo tratado neste trabalho assim como dos dois problemas utilizados na avaliação. Em seguida, o conceito de computação autônoma será abordado além de uma ferramenta importante usada neste trabalho que fornece autonomia às aplicações - o *middleware* EasyGrid AMS - citando o conjunto de trabalhos que descreve as funcionalidades mais importantes e a filosofia deste *framework*. Por fim, alguns trabalhos da literatura que realizam estratégias de paralelização em ambientes de larga escala são apresentados e algumas comparações são feitas afim de classificar e mostrar a contribuição deste trabalho.

2.1 Aplicações *Branch-and-Prune*

Branch-and-prune são algoritmos geralmente muito utilizados para se resolver problemas de satisfação de restrição (*Constraint Satisfaction Problem* - CSP) [43, 60]. Tais problemas necessitam de algum tipo de busca exaustiva para encontrar soluções viáveis, podendo

fazer o uso de um conjunto de restrições para selecionar as soluções corretas. Problemas de satisfação de restrição são empregados por diversas aplicações de diversas áreas como logística, bioquímica e robótica [70].

Aplicações *branch-and-prune* utilizam técnicas de busca exaustiva baseadas em ramificação (*branch*) e poda de ramos inviáveis (*prune*). A topologia de uma árvore é conceitualmente formada durante a busca onde cada nó indica uma solução parcial do problema da aplicação e cada ramo da árvore representa um conjunto de possíveis soluções viáveis a partir de um determinado nó. Soluções parciais podem ser completas - uma solução do problema - ou soluções que sofreram poda e, portanto, não são soluções do problema.

O algoritmo *branch-and-prune* geralmente é construído utilizando-se o tipo de busca em profundidade (*depth-first search*) que tem o objetivo de percorrer sempre o ramo partindo do nó da árvore mais a esquerda ou o mais a direita ainda não percorrido. Caso um ramo da árvore não seja viável devido a alguma restrição do problema, tal ramo é excluído da busca e o algoritmo retrocede (*backtracking*) para o nó pai do nó que definiu a poda do ramo. Este processo se repete até todas as possibilidades serem examinadas.

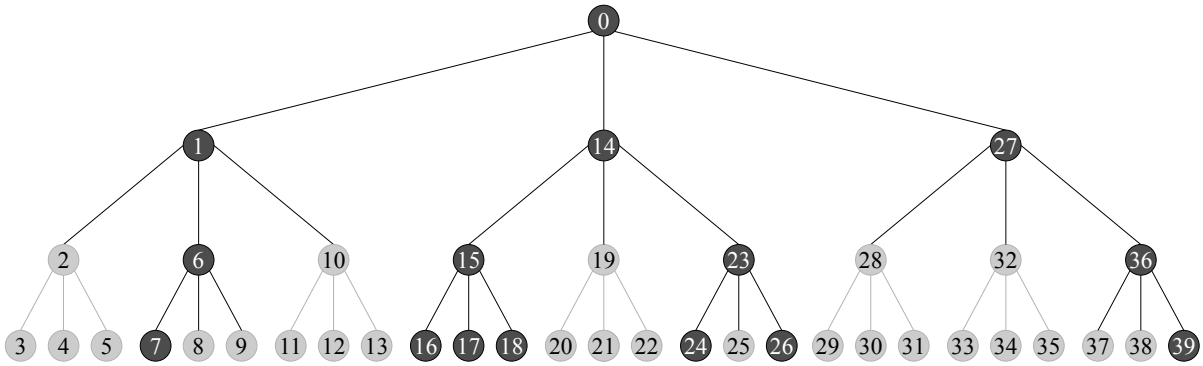


Figura 2.1: Exemplo de uma árvore *branch-and-prune*.

A Figura 2.1 representa uma exemplo de árvore com todo o espaço de busca. A busca inicia pelo nó raiz ou 0 e cada nó significa uma verificação considerando os nós anteriores no caminho na árvore até sua posição (sendo representado pelas circunferências numeradas). A cada nó, verifica-se as restrições do problema e ramifica para os nós possíveis. Na figura, os nós escuros indicam um nó viável, isto é, não infringem nenhuma das restrições, e os nós claros são inviáveis, isto é, infringem alguma restrição. No exemplo, o nó 0 verifica as restrições iniciais e identifica a ocorrência de 3 possibilidades. Como a busca é feita em profundidade e pela esquerda, o nó 1 é escolhido primeiro. O nó 1 verifica as restrições e identifica que o nó 2 é inviável. Neste momento, a busca retrocede para o nó 1 para verificar se há mais possibilidades. O nó 1 percebe que o nó 6 é viável e continua a busca pelo nó 7. Como o nó 7 é folha, indica-se uma solução completa e a busca retrocede para

o nó 6. A busca continua desta forma até que todo o espaço de busca seja percorrido, considerando as eventuais podas.

Existe uma grande semelhança na construção algorítmica da técnica *branch-and-prune* (B&P) e *branch-and-bound* (B&B) [35]. Basicamente, a diferença está no objetivo da busca e na forma como a poda de ramos é feita. Enquanto nos algoritmos B&P geralmente avalia todas as soluções viáveis considerando as restrições do problema, nos algoritmos B&B avalia apenas soluções viáveis que ficam dentro de um dado limite [27]. Na maioria das vezes, este limite é o custo da melhor solução já encontrada e a poda acontece em um nó quando é visto que nenhuma solução derivada desta solução parcial seria melhor que aquela já encontrada.

2.1.1 *N*-Rainhas

O N-Rainhas é um problema clássico que consiste em encontrar configurações de N rainhas em um tabuleiro $N \times N$ respeitando certas restrições. Estas restrições correspondem a não haver nenhuma peça em posição de ataque às demais, de acordo com as regras do jogo de xadrez. Em outras palavras, duas rainhas não devem ocupar a mesma linha, coluna ou diagonal. A Figura 2.2 ilustra soluções para $N = 8$.

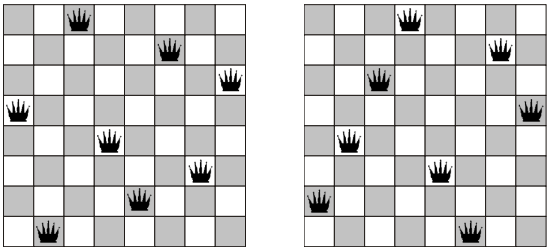


Figura 2.2: Duas das soluções para o 8-*Queens*.

Uma simples representação do problema é dada através de uma matriz $N \times N$, onde cada célula (l, c) (l e c são, respectivamente, linha e coluna) equivale a uma possível posição de rainha no tabuleiro. Assim, uma solução S é dada pelo conjunto de N pares $S=\{p_1, p_2, \ldots, p_n\}$, tal que, para cada par $p_i = (l_i, c_i)$ e $p_j = (l_j, c_j)$:

- $l_i \neq l_j$ (mesma linha),
- $c_i \neq c_j$ (mesma coluna),
- $l_i + c_i \neq l_j + c_j$ (mesma diagonal positiva) e
- $l_i - c_i \neq l_j - c_j$ (mesma diagonal negativa).

Neste contexto, diagonais positivas são todas as diagonais do tabuleiro que se estendem do canto superior direito ao canto inferior esquerdo. Ou seja, cada diagonal é um conjunto de células do tabuleiro onde a soma dos índices da linha e da coluna são valores constantes. Deste modo, o número de diagonais positivas equivale a $2N - 1$. De forma similar, diagonais negativas são aquelas que se estendem do canto superior esquerdo ao canto inferior direito do tabuleiro. Ou seja, cada diagonal negativa é um conjunto de células onde a subtração dos índices da linha e coluna são valores constantes. Do mesmo modo que ocorre com as diagonais positivas, o número de diagonais negativas equivale a $2N - 1$.

O N-Rainhas pode apresentar vários escopos de estudo de acordo com suas diferentes perspectivas. Em [21], são expostas abordagens distintas do problema capazes de classificá-lo em 3 categorias. A primeira categoria consiste em encontrar o número total de soluções, ou seja, uma contagem de todas as disposições de N rainhas no tabuleiro $N \times N$ obedecendo as restrições. A segunda categoria corresponde a encontrar soluções (ou o número delas) fundamentais ou únicas. Ou seja, soluções que são equivalentes a partir de rotações do tabuleiro. A terceira refere-se a encontrar uma determinada quantidade de soluções, e não necessariamente todas. O número de soluções já conhecidas para cada valor de N pode ser encontrado em [7, 8].

Tanto a primeira categoria quanto a segunda, são tratadas neste trabalho. O algoritmo utilizado para contar o número total de soluções de um dado N, calcula, na verdade, todas as soluções únicas [66] e, delas, derivam-se as soluções totais. O Apêndice A apresenta uma descrição do algoritmo sequencial utilizado. Este algoritmo foi escolhido por ser considerado, atualmente, um dois mais eficientes em relação ao tempo de processamento [66].

O uso de rotações e reflexões do tabuleiro é comum neste problema para diminuir o espaço de busca do problema [18]. Rotações de 90^o, 180^o ou 270^o do tabuleiro e reflexões (solução espelho) das soluções obtidas fazem com que o algoritmo se resuma a encontrar soluções únicas e derivar as soluções totais delas.

As soluções únicas ou fundamentais são aquelas que, ao se fazer rotações de 90^o, 180^o ou 270^o do tabuleiro e reflexões, se obtém a mesma solução que a inicial. Na Figura 2.3, a solução do tabuleiro A representa uma solução única. Dela, gera-se a solução B, C e D, onde A=C e B=D. As soluções provenientes das rotações (soluções que não são únicas) podem ser todas iguais, iguais em partes (como na Figura 2.3) ou até todas diferentes entre si. Dessas soluções rotacionadas, obtêm-se as reflexões (no caso da Figura 2.3, as reflexões E, F, G e H).

Se a solução derivada da rotação de 90^o for igual a solução original, tem-se apenas 2 soluções totais (a original mais a sua reflexão). Caso contrário, se a rotação de 180^o

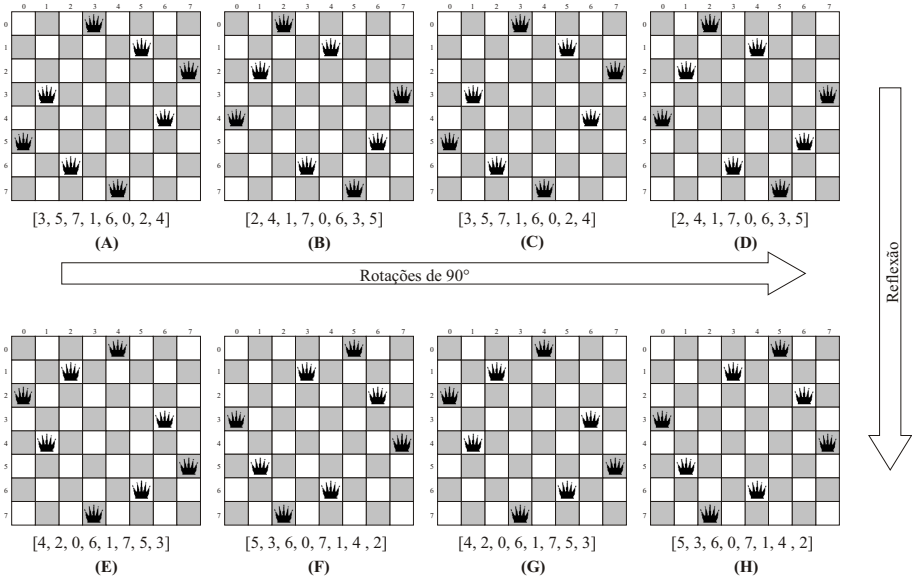


Figura 2.3: Uma solução única para 8-*Queens*, suas rotações e reflexões.

for igual a solução original, tem-se 4 soluções no total (a original mais a rotação de 90^o (diferente da original) mais as respectivas reflexões). Caso contrário, todas as soluções derivadas das rotações são diferentes da original, e então existem 8 soluções totais através do cálculo de apenas uma. Assim, estas características podem ser usadas no algoritmo e diminuir, então, o número de operações.

2.1.2 PDGDM

O objetivo do Problema Discreto de Geometria das Distâncias em Moléculas (PDGDM) é encontrar soluções viáveis da representação estrutural tridimensional de uma molécula de proteína, dado apenas algumas distâncias provenientes da Ressonância Magnética Nuclear (RMN) da molécula. Ele é uma versão discreta do Problema de Geometria das Distâncias em Moléculas (PGDM) [17].

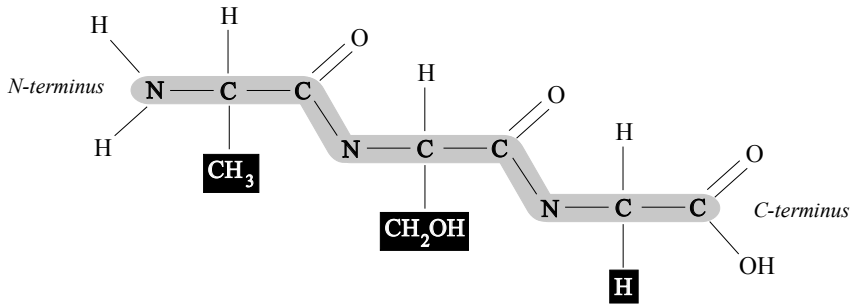


Figura 2.4: Parte de uma cadeia proteica.

Uma molécula de proteína pode ser vista como uma sequência de átomos. A Figura 2.4 apresenta a representação clássica de uma molécula, mostrando a cadeia de seus

elementos químicos. Uma proteína é formada por aminoácidos conectados através de ligações peptídicas [16]. A parte sobre a faixa cinza mostra a sua cadeia principal ou *backbone*. A parte sobre um retângulo preto indica o radical que caracteriza o aminoácido.

O PDGDM, a princípio, tem o objetivo de encontrar a estrutura tridimensional da cadeia principal de uma molécula de proteína. Este problema considera um conjunto de hipóteses que são comumente aplicáveis às estruturas moleculares proteicas [64]. Dados quaisquer 4 átomos consecutivos $a_{i-3}, a_{i-2}, a_{i-1}, a_i$ da cadeia principal:

Hipótese 1 todas as distâncias inter-atômicas entre esses 4 átomos devem ser conhecidas. Em outras palavras, deve-se existir uma clique entre estes 4 átomos onde o tamanho das arestas são exatamente proporcionais às distâncias;

Hipótese 2 o ângulo entre os vetores $\overrightarrow{a_{i-3}, a_{i-2}}$ e $\overrightarrow{a_{i-2}, a_{i-1}}$ não deve ser múltiplo de π.

Para entender o problema e sua complexidade, é interessante saber como podem ser calculadas as coordenadas de um átomo $a_i = (a_{i_x}, a_{i_y}, a_{i_z})$ com n átomos $(0 \leq i < n)$ [64]. A distância entre dois átomos a_i e a_j é representada por $d_{i,j}$ com $0 \leq i < n-1$ e $1 \leq j < n$. Os ângulos de ligação existentes entre os vetores $\overrightarrow{a_{i-3}, a_{i-2}}$ e $\overrightarrow{a_{i-2}, a_{i-1}}$ são iguais a $\theta_{i-2,i}$ para $2 \leq i < n$. Existem ainda os ângulos de torção $\omega_{i-3,i}$ para $4 \leq i < n$ entre 4 átomos consecutivos que indicam o ângulo formado entre os vetores $\overrightarrow{a_{i-3}, a_{i-2}}$ e $\overrightarrow{a_{i-1}a_i}$ no espaço tridimensional. Através das distâncias, dos ângulos de ligação e dos ângulos de torção é possível obter as coordenadas de um átomo a_i através da Equação 2.1.

$$\begin{bmatrix} a_{i_x} \\ a_{i_y} \\ a_{i_z} \\ 1 \end{bmatrix} = B_0 B_1 \dots B_i, 0 \leq i < n \tag{2.1}$$

B_i são as matrizes de transformação apresentadas na Equação 2.2.

$$B_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, B_1 = \begin{bmatrix} -1 & 0 & 0 & -d_{0,1} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$B_2 = \begin{bmatrix} -\cos\theta_{0,2} & -\sin\theta_{0,2} & 0 & -d_{1,2}\cos\theta_{0,2} \\ \sin\theta_{0,2} & -\cos\theta_{0,2} & 0 & d_{1,2}\sin\theta_{0,2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ e}$$

$$B_i = \begin{bmatrix} -\cos\theta_{i-2,i} & -\sin\theta_{i-2,i} & 0 & -d_{i-1,i}\cos\theta_{i-2,i} \\ \sin\theta_{i-2,i}\cos\omega_{i-3,i} & -\cos\theta_{i-2,i}\cos\omega_{i-3,i} & -\sin\omega_{i-3,i} & d_{i-1,i}\sin\theta_{i-2,i}\cos\omega_{i-3,i} \\ \sin\theta_{i-2,i}\sin\omega_{i-3,i} & -\cos\theta_{i-2,i}\sin\omega_{i-3,i} & \cos\omega_{i-3,i} & d_{i-1,i}\sin\theta_{i-2,i}\sin\omega_{i-3,i} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

O cosseno do ângulo de torção $\omega_{i-3,i}$ para $3 < i < n$ pode ser obtido através da expressão da Equação 2.3 e o seno, através da Equação 2.4.

$$\begin{aligned} \cos\omega_{i-3,i} = {} & \frac{d_{i-2,i-1}^2 + d_{i-3,i-2}^2 + d_{i-1,i}^2 - d_{i-3,i}^2}{2d_{i-3,i-2}d_{i-1,i}\sin(\theta_{i-2})\sin(\theta_{i-1})} - \frac{d_{i-2,i-1}\cot(\theta_{i-2})}{d_{i-1,i}\sin(\theta_{i-1})} - \frac{d_{i-2,i-1}\cot(\theta_{i-1})}{d_{i-3,i-2}\sin(\theta_{i-2})} + \\ & + \cot(\theta_{i-2})\cot(\theta_{i-1}) \end{aligned} \tag{2.3}$$

$$\sin\omega_{i-3,i} = \pm\sqrt{1 - (\cos\omega_{i-3,i})^2} \tag{2.4}$$

Assim, tem-se os valores das coordenadas dos 3 primeiros átomos:

$$a_0 = (0,\, 0,\, 0),\ a_1 = (-d_{0,1},\, 0,\, 0) \text{ e } a_2 = (-d_{0,1} + d_{1,2}\cos\theta_{0,2},\, d_{1,2}\sin\theta_{0,2},\, 0).$$

O valor da quarta coordenada de a_3 apresenta duas possibilidades:

$$a_3 = \begin{pmatrix} -d_{0,1} + d_{1,2}\cos\theta_{0,2} - d_{2,3}\cos\theta_{0,2}\cos\theta_{1,3} + d_{2,3}\sin\theta_{0,2}\sin\theta_{1,3}\cos\omega_{0,3}, \\ d_{1,2}\sin\theta_{0,2} - d_{2,3}\sin\theta_{0,2}\cos\theta_{1,3} - d_{2,3}\cos\theta_{0,2}\sin\theta_{1,3}\cos\omega_{0,3}, \\ d_{2,3}\sin\theta_{1,3}\sqrt{1 - \cos^2\omega_{0,3}} \end{pmatrix} \text{ e}$$

$$a_3' = \begin{pmatrix} -d_{0,1} + d_{1,2}\cos\theta_{0,2} - d_{2,3}\cos\theta_{0,2}\cos\theta_{1,3} + d_{2,3}\sin\theta_{0,2}\sin\theta_{1,3}\cos\omega_{0,3}, \\ d_{1,2}\sin\theta_{0,2} - d_{2,3}\sin\theta_{0,2}\cos\theta_{1,3} - d_{2,3}\cos\theta_{0,2}\sin\theta_{1,3}\cos\omega_{0,3}, \\ -d_{2,3}\sin\theta_{1,3}\sqrt{1 - \cos^2\omega_{0,3}} \end{pmatrix}$$

Pode-se reparar que apenas a coordenada z é modificada. Seus valores indicam que existem duas possíveis coordenadas para o átomo a_3: uma oposta a outra em relação ao plano formado pelos três átomos anteriores. Desta forma, seguindo os próximos átomos da molécula, sempre haverá duas possibilidades dada a fixação dos átomos anteriores. Logo, existirão 2^{n-3} estruturas tridimensionais possíveis da molécula [64], o que projeta a visualização de uma árvore binária. No entanto, na maioria das vezes, outras distâncias, que são diferentes das distâncias entre cada 4 átomos consecutivos, podem ser conhecidas através da RMN. Através destas distâncias extras, o algoritmo é capaz de descartar ramos da árvore, reduzindo o espaço de busca. Quando a coordenada de um átomo é encontrada durante a busca, é verificado se essa coordenada está correta de acordo com as distâncias extras envolvendo os átomos anteriores ao átomo em questão.

Por exemplo, em uma molécula hipotética com 6 átomos, as distâncias $d_{0,1}$, $d_{0,2}$, $d_{0,3}$, $d_{1,2}$, $d_{1,3}$, $d_{1,4}$, $d_{2,3}$, $d_{2,4}$, $d_{2,5}$, $d_{3,4}$, $d_{3,5}$ e $d_{4,5}$ são obrigatoriamente conhecidas no PDGDM (entre cada 4 átomos consecutivos). As distâncias $d_{0,4}$ e $d_{1,5}$ são as distâncias extras conhecidas. Ao se calcular as possíveis coordenadas do átomo 4, se a distância calculada entre a coordenada do átomo 0 (x_0, y_0, z_0) e a do átomo 4 (x_4, y_4, z_4) for considerada diferente (com erro ϵ) da distância real $d_{0,4}$, a poda ocorria neste nó e os ramos que seriam gerados a partir dele estariam fora da busca. O mesmo aconteceria para o átomo 5, neste exemplo.

2.2 Computação Autônoma

O conceito de computação autônoma vem de uma analogia ao sistema nervoso humano, um dos mais sofisticados exemplos de comportamento autônomo existente no mundo [65]. O sistema nervoso autônomo é a parte do sistema nervoso que controla as funções vegetativas ou involuntárias do corpo humano, como a circulação sanguínea, respiração, controle térmico, recuperação de ferimentos e resposta a estímulos do ambiente. Por exemplo, se uma pessoa encontra-se em um ambiente frio, seu organismo apresenta uma resposta involuntária à temperatura baixa arrepiando os pelos (devido à contrações musculares) e tremendo-se. Este autocontrole impacta todo o organismo e ele é responsável pela sobrevivência da espécie além de possibilitar o equilíbrio e um bom funcionamento do organismo. A autonomia do sistema humano recentemente inspirou um novo paradigma na computação para o controle de aplicações e/ou sistemas de computadores [48].

Atualmente, princípios de autonomia vêm sendo introduzidos em computação, especialmente na área de computação distribuída [48, 40]. Tais princípios de autogerência são basicamente garantidos através das seguintes propriedades [65, 56]:

- autoconfiguração (*self-configuring*) - capacidade do sistema reconfigurar-se, isto é, reconhecer mudanças no ambiente e na aplicação, entender seus impactos e adaptar seus parâmetros a elas.
- autorrecuperação (*self-healing*) - habilidade do sistema detectar e recuperar-se de falhas sem impedir e prejudicar seu próprio funcionamento.
- auto-otimização (*self-optimising*) - capacidade de detectar um desempenho inferior ao esperado e otimizar-se garantindo uma melhor execução conforme as configurações estabelecidas.
- autoproteção (*self-protecting*) - capacidade do sistema proteger-se de possíveis ataques maliciosos, garantindo a integridade dos dados da aplicação e do sistema (possivelmente através de técnicas eficazes de criptografia e certificação digital).

Os atributos necessários para o sistema tornar-se autogerenciável e possuir tais propriedades são: automonitoramento (*self-monitoring*), autoconhecimento (*self-awareness*), conhecimento do ambiente (*environment-awareness*) e autoajuste (*self-adjusting*) [56]. Seja o ambiente de grades computacionais ou qualquer outro ambiente dinâmico, tais atributos são importantes para que o sistema ou a aplicação sejam capazes de manter sua execução eficientemente sem a interferência do usuário.

Apesar dos benefícios da computação autônoma, existe uma grande dificuldade no desenvolvimento dos programas paralelos e, ao mesmo tempo, autônomos. O programador deverá estar sempre preocupado em garantir as propriedades da autonomia, como a detecção e reparação de falhas, a autoconfiguração do sistema, a coleta de informações para a melhoria do desempenho e a integridade dos dados. Os programas serão ainda difíceis de testar em tempo de desenvolvimento, pois seu comportamento depende da interação que ele irá ter com outras entidades. Isto torna o uso e o desenvolvimento da autonomia complexos e até hoje relativamente pouco evoluídos.

2.2.1 Sistemas Gerenciadores para Computação Distribuída

Para contornar o problema da dificuldade em desenvolver programas paralelos de forma autônoma em ambientes computacionais de larga escala, muitos projetos vêm desenvolvendo *middlewares*, isto é, camadas de *software* de interface entre a aplicação e a infraestrutura [42].

A maioria dos projetos atuais adota uma visão centrada nos recursos disponíveis no ambiente distribuído (como grades computacionais) para garantir uma utilização eficiente dos mesmos. Geralmente existe um ou mais *broker* que recebem tarefas (*jobs*) e as

distribuem estaticamente entre os recursos. Tal gerenciamento é feito tipicamente por um Sistema Gerenciador de Recursos (SGR ou RMS - *Resource Management System*) com base em monitoramento e análise das informações específicas do sistema, numa maneira similar ao gerenciamento de *clusters* [42, 49]. Os RMSs têm o objetivo de maximizar a utilização dos recursos, independentemente dos requisitos e características internas da aplicação.

Condor-G é um exemplo de um sistema gerenciador de recursos especialmente projetado para grades de computadores [25]. Ele integra ferramentas do Globus *toolkit* [28] e do sistema de gerência Condor [46], fazendo com que os usuários tenham acesso a recursos em diferentes domínios, possuindo uma visão unificada da grade computacional. Através do Globus, este gerenciador obtém uma série de serviços como transferência de arquivos, autenticação de usuários e disseminação de informações sobre o estado dos recursos. O usuário define os processos a serem executados e o RMS Condor-G é responsável pela descoberta e aquisição de recursos, inicialização, monitoramento, gerenciamento da execução, notificação de término, detecção e tratamento de falhas. Seu escalonador de processos é centralizado e estático. Ele segue o objetivo de maximizar a utilização dos recursos disponíveis, combinando pedidos de recursos dos usuários com as ofertas de recursos do sistema. Por possuir suporte para segurança do sistema (autoproteção) e tolerância a falhas (autorrecuperação), o sistema Condor-G pode ser considerado um sistema parcialmente autônomo.

Cactus [3, 2] é outro exemplo de um sistema que trata a gerência de recursos. Seu objetivo é esconder atrás de um único ponto as complexidades do ambiente distribuído de larga escala. A descoberta de recursos também é feita através da ferramenta Globus, assim como a autenticação de usuários e a transferência de arquivos. Mecanismos de tolerância a falhas baseados em *checkpoint* são implementados neste *middleware* e o balanceamento de carga é realizado através da decomposição da aplicação em subproblemas, de forma orientada aos recursos e centralizada. Este sistema também pode ser considerado parcialmente autônomo por possuir autoproteção, auto-otimização e autorrecuperação.

Em ambientes computacionais de larga escala, esse tipo de abordagem voltada aos recursos pode não ser suficiente para que toda a variedade de aplicações que necessitam de tais ambientes obtenham um bom aproveitamento dos recursos. É importante considerar as características individuais de cada aplicação para que possam ser feitos ajustes adequados à sua execução [53]. *Middlewares* que se concentram na execução da aplicação são chamados de Sistemas Gerenciadores de Aplicação (SGA ou AMS - *Application Management Systems*). Uma das maiores vantagens de um *middleware* do tipo AMS é a capacidade de transformar a aplicação do usuário em uma versão *system aware*. Aplicações cientes do estado dos recursos do ambiente durante a sua execução podem se

auto-ajustar as suas mudanças, buscando uma maior eficiência na sua execução conforme variações na disponibilidade de recursos.

GrADS [24] é um exemplo de sistema gerenciador de aplicações. Este *middleware* fornece ferramentas e bibliotecas que permitem ao usuário criar aplicações que possam ser encapsuladas como programas objetos configuráveis (*COPs*) que inclui um modelo que estima o desempenho da aplicação em um conjunto de recursos. 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. Apresenta mecanismos de tolerância a falhas (*checkpoints*) e monitoramento dos recursos. GrADS pode ser considerado um AMS parcialmente autônomo através de sua autoconfiguração e auto-otimização. Além disso, apresenta projetos específicos para aplicações, como é o caso do GrADSAT [15] a ser descrito adiante.

O EasyGrid é outro exemplo de um *middleware* AMS e apresenta várias propriedades de autonomia, destacando-se pela sua eficiência.

2.2.2 EasyGrid AMS

O *middleware* EasyGrid AMS [52, 61] é um sistema gerenciador de aplicação que é integrado ao código do programa, tornando transparente a gerência da execução de aplicações MPI [26, 47] em ambientes como grades computacionais. Este *middleware* é independente de qualquer outro sistema de *middleware grid*, necessitando apenas do Globus Toolkit [28] (GSI para autenticação através de certificação digital e GRAM para submeter *grid jobs*) e da instalação padrão LAM/MPI [38] nas máquinas pertencentes ao ambiente de grades computacionais. Existem versões diferentes especificamente afinadas para classes distintas de aplicações, como por exemplo aplicações *bag-of-tasks* (BoT), ou mestre-trabalhador, e aquelas em que as tarefas possuem relação de precedência.

O EasyGrid AMS é composto por três níveis de hierarquia de gerenciamento. O nível mais alto (nível 0) refere-se ao gerenciador global (GG), encarregado de gerenciar todos os *sites* pertencentes à grade computacional e envolvidos na execução da aplicação. Um nível mais abaixo (nível 1) refere-se aos gerenciadores de *sites* (GS), responsáveis por gerenciar os processos da aplicação atribuídos a cada *site* da grade. Por fim, o último nível (nível 2) é composto por gerenciadores locais de máquina (GM), responsáveis pelo escalonamento, criação e execução de processos da aplicação nas máquinas locais. A Figura 2.5 mostra uma representação gráfica dessa hierarquia. Os gerenciadores são implementados por processos que executam junto aos processos da aplicação. No entanto, os gerenciadores foram projetados para apresentar um baixo grau de intrusão [61], pois comportam-se

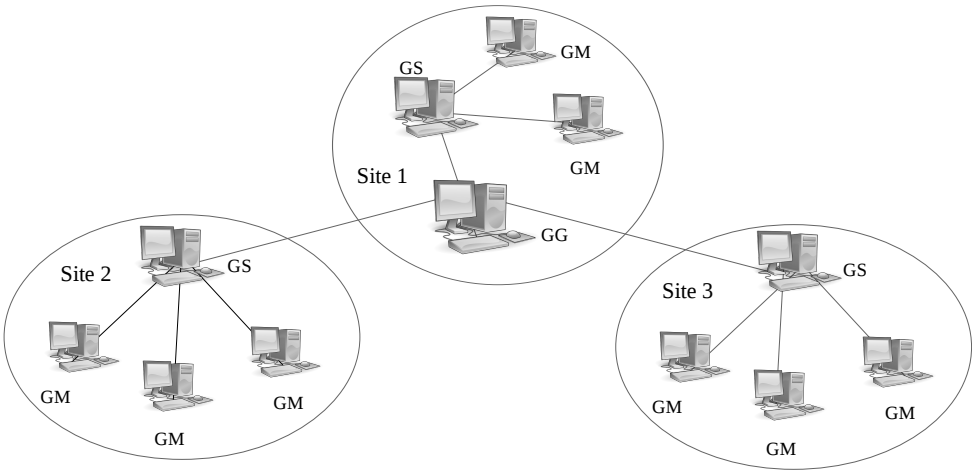


Figura 2.5: Hierarquia de gerenciadores do EasyGrid AMS.

como *daemons*, tratando mensagens de gerenciamento.

Cada processo gerenciador é baseado em uma arquitetura integrada [52] contendo quatro camadas: gerenciamento de processos, monitoramento da aplicação, escalonamento dinâmico [51] e tolerância a falhas [63]. A funcionalidade de cada camada está associada ao nível hierárquico do processo gerenciador. Por exemplo, políticas de escalonamento dinâmico e mecanismos de tolerância a falhas podem diferir de acordo com o nível hierárquico em que se encontram.

A camada de gerenciamento de processos do AMS é responsável pela criação dinâmica de processos tanto da aplicação quanto de gerenciadores. Ela também controla o roteamento de mensagens destes processos. A camada de monitoramento (*self-awareness*) coleta estados do sistema, alimentando as camadas de escalonamento dinâmico e de tolerância a falhas com diversas informações da execução (*self-configuring*) [68]. A camada de escalonamento dinâmico (*self-optimising*) é responsável por redistribuir tarefas, propondo um balanceamento de carga proativo de acordo com informações do monitoramento [13, 51]. A camada de tolerância a falhas (*self-healing*) é implementada principalmente através de recriações de processos e de *logs* de mensagens da aplicação. Assim, se um recurso falhar, os gerenciadores de *site* ou global se encarregarão de recuperar os *logs* de mensagens e re-executar tarefas da aplicação, utilizando outros recursos disponíveis e sem interferir a execução do restante da aplicação. No caso de falha do gerenciador global, a camada de tolerância a falhas recupera o sistema através do mecanismo *de checkpoint*.

Uma segunda característica fundamental para a eficiência do EasyGrid AMS é o modelo de execução de aplicações que é diferente do modelo convencional e ele será brevemente discutido na Subseção 2.2.2.2.

2.2.2.1 Escalonamento Dinâmico Reativo e Proativo

O escalonamento dinâmico é um importante mecanismo para possibilitar a eficiência das execuções de tarefas da aplicação em um sistema dinâmico e heterogêneo. Diante de um desbalanceamento de carga, isto é, um recurso apresenta mais trabalho que outro, um escalonador dinâmico deve avaliar se é necessário mudar a configuração atual de carga do sistema (tirar carga extra de uma máquina sobrecarregada e passar para uma sub-carregada), distribuindo a carga de forma mais igualitária para reduzir o tempo de execução esperado. Realizar um reescalonamento de tarefas durante a execução da aplicação pode ser um procedimento custoso, pois a decisão deve considerar todos ou um grupo de nós para determinar a escolha da distribuição da carga. Ainda, se a redistribuição for permitida, as tarefas devem ser transferidas entre os nós. Todos estes procedimentos exigem processamento, o que pode comprometer a execução da aplicação. Por outro lado, o esforço do reescalonamento pode ser recompensado com uma distribuição de carga mais justa entre os nós, possibilitando um grande aproveitamento dos recursos e uma melhora na eficiência.

Basicamente existem duas abordagens de escalonamento dinâmico de carga: o reativo (tradicional) e o proativo. Na abordagem reativa, o reescalonamento é feito sob demanda. Logo, só é realizado quando necessário, consumindo menos processamento e banda de rede. No entanto, o reescalonamento geralmente atrasa a execução das tarefas da aplicação, subutilizando os recursos. Enquanto o reescalonamento é realizado, o processador (um recurso importantíssimo para aplicações *cpu-bound*) fica ocioso, já que as tarefas só poderão ser executadas após a decisão do escalonador dinâmico. Na abordagem proativa, como o reescalonamento é feito antes do sistema atualmente ficar desbalanceado, os recursos não tornam-se subutilizados pela aplicação. Por outro lado, o reescalonamento deve ser realizado periodicamente. O período não pode ser muito longo nem muito curto. Se for muito longo, a predição é prejudicada. Se for muito curto, o escalonador dinâmico irá consumir um grau significante dos recursos (principalmente o processamento), chegando ao ponto dos ganhos devido ao escalonamento não compensarem o custo da implementação.

Em [61], uma comparação é feita entre o escalonamento dinâmico reativo e proativo (este último utilizando o EasyGrid AMS) através de uma aplicação *bag-of-task cpu-bound* sintética. Três tipos de cenários são considerados para tarefas de granularidade uniforme: um *cluster* dedicado/quase homogêneo (com 15 processadores), uma grade computacional heterogênea/estática e uma grade computacional heterogênea/dinâmica (ambas as grades com 53 processadores no total e 4 *sites*). Nos dois cenários de grades computacionais, a aplicação com cerca de 500 tarefas obteve um ganho entre 9,62% e 11,17% sobre a abordagem reativa. Com granularidade não uniforme, o ganho da aplicação de 1000 tarefas chegou a 18,58%, sendo a execução em um único *site* (com 22 processadores) heterogêneo

e dinâmico. Assim, os resultados mostram que a abordagem proativa se comporta consideravelmente melhor que a abordagem reativa principalmente em ambientes heterogêneos e dinâmicos.

2.2.2.2 O Modelo de Execução 1PTask

Dois modelos de execução de aplicações do tipo *bag-of-task* - onde há um conjunto de tarefas ou unidades de trabalho independentes entre si, como no caso de aplicações *branch-and-prune* e *branch-and-bound* - são considerados neste trabalho, o mais tradicional 1PProc e o alternativo 1PTask.

No modelo 1PProc (um processo por processador) cada processador recebe um processo que executará as unidades de trabalho até o fim da aplicação. As unidades de trabalho podem ser balanceadas entre os processos, mas cada processo durará toda a execução da aplicação. O escalonamento de carga neste modelo torna-se complicado, pois existem dois problemas relacionados às questões: como escalonar a carga entre os processos e como isto deve ser implementado pelo desenvolvedor da aplicação. Caso não seja implementado, o dinamismo e a heterogeneidade do ambiente afetará bastante no desempenho.

O EasyGrid AMS implementa o modelo de execução de aplicações chamado 1PTask (um processo por tarefa) [61]. Este modelo alternativo define uma tarefa como sendo uma unidade de trabalho computacional independente (assim como no modelo 1PProc) e de granularidade mais fina do que a tradicionalmente utilizada. Mais detalhadamente, este modelo indica que cada tarefa é um processo da aplicação que recebe, computa e envia dados. Então, a aplicação é dividida em uma certa quantidade de tarefas que geralmente tende a ser bem maior que o número de processadores disponíveis. Para evitar uma grande concorrência por recursos, o *middleware* não cria todos os processos imediatamente. Na verdade, esta grande quantidade de tarefas pode ser melhor escalonada entre os recursos enquanto ainda aguardam para serem disparadas.

O mecanismo de tolerância a falhas torna-se mais simples sob este modelo. No lugar de técnicas de *checkpoints*, que podem ser custosas, o *middleware* detecta a falha de um nó e recupera a aplicação recriando as tarefas perdidas por meio de *logs* de mensagens sem a necesidade de parar a execução do resto da aplicação [63]. Como as tarefas tendem a terem granularidade fina, a recriação tende a não ser custosa. Passa a ser melhor re-executar a tarefa ao invés de guardar informações de estado de memória delas [61].

2.3 *Branch-and-bound* e *Branch-and-prune* Paralelos

Nos últimos anos, cientistas de computação de alto desempenho vêm questionando o modo como aplicações vêm sendo paralelizadas em ambientes distribuídos de larga escala [34, 10]. Tais ambientes apresentam uma grande quantidade de recursos que são necessários para aplicações de larga escala e dificilmente são explorados de forma eficiente pelas estratégias de paralelização. A quantidade de recursos nos ambientes de computação atuais é bastante superior em relação aos ambientes distribuídos mais usados na década de 90, como os *clusters* de computadores.

2.3.1 Estratégia Mestre-Trabalhador

O modelo convencional de desenvolvimento de algoritmos paralelos se baseia na abordagem mestre-trabalhador, proposta inicialmente para *clusters* de computadores. Tal modelo de programação inicialmente possui uma estratégia de paralelização estática e centralizada no mestre, que considera que as máquinas são homogêneas e todo o trabalho é dividido igualmente entre os recursos. O trabalho computacional da aplicação é dividido pelo mestre entre os trabalhadores estaticamente, como explicado no artigo [27]. No entanto, esta estratégia não é considerada uma boa alternativa para os sistemas distribuídos atuais, já que eles são dinâmicos, compartilhados e heterogêneos. Além disso, o trabalho pode não ser homogêneo, como no caso de aplicações *branch-and-prune* e *branch-and-bound*. No caso, se um trabalhador terminar seu trabalho antes que outro, ele permanecerá ocioso até o término da aplicação.

Ainda em [27] e em outros trabalhos como [29, 6], para resolver tal problema de desperdício de poder computacional, todo o trabalho da aplicação é separado em unidades de trabalho - a menor unidade de computação de um algoritmo do tipo *bag-of-task*. Nesta estratégia mais inteligente, o mestre possui uma fila dessas unidades de trabalho e as distribui sob demanda para os trabalhadores. A esta estratégia dá-se o nome de mestre-trabalhador sob demanda. Um trabalhador inicialmente resolve parte do problema descobrindo as subárvores. Enquanto ele continua analisando sua subárvore, as outras são depositadas na fila do mestre e conforme os trabalhadores vão tornando-se ociosos, os trabalhos vão sendo distribuídos pelo mestre entre eles.

O principal ponto fraco das estratégias mestre-trabalhador é o fato de existir um gargalo e uma concentração de ponto de falha no mestre. Como os ambientes distribuídos atuais são de larga escala, a concentração dos trabalhos no mestre comprometem a eficiência da execução. Além disso, como tais ambientes são suscetíveis a falhas, o ponto centrali-

zado pode interromper ou atrasar a execução de toda a aplicação.

2.3.2 Estratégia *WorkStealing*

Recentemente, a abordagem mestre-trabalhador vem sendo substituída por propostas mais sofisticadas. Na estratégia de *workstealing*, parte do excesso do trabalho é retirado ou "roubado" de um trabalhador que já possui unidades de trabalho alocadas a ele. Deve existir, então, uma estrutura de gerenciamento capaz de verificar se um trabalhador possui mais unidades de trabalho que outro. Quando um trabalhador ficar ocioso, as unidades de trabalho extras do trabalhador com excesso de carga será retirada e alocada para o trabalhador sem carga.

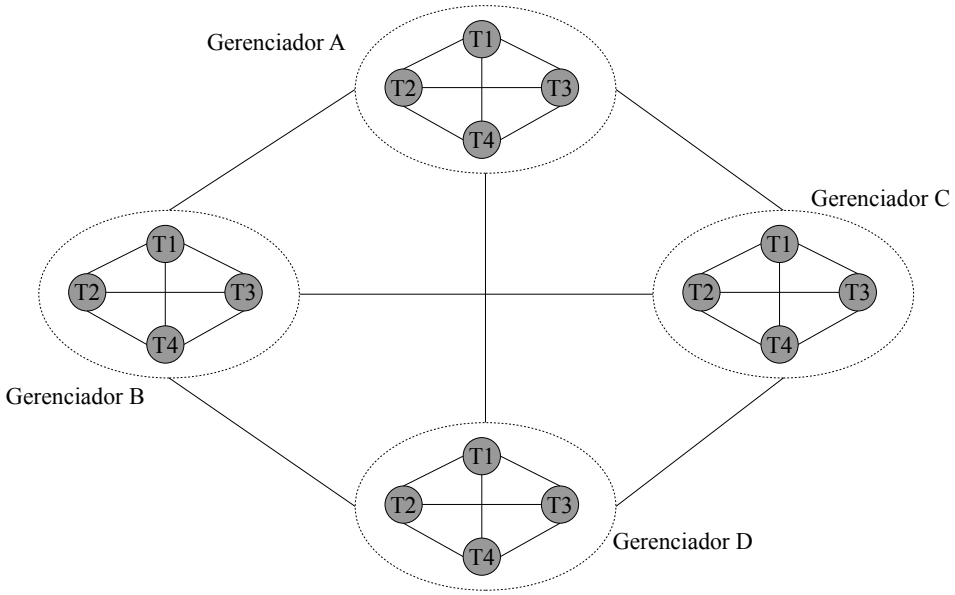


Figura 2.6: Exemplo de gerenciamento distribuído *WorkStealing*.

Dentre os tipos de estratégias *workstealing* mais utilizadas existem dois que se destacam de acordo com a forma como o trabalho é gerenciado entre os trabalhadores. Um deles é através do gerenciamento distribuído, onde um grupo de trabalhadores fica associado a um gerenciador e os gerenciadores trocam informações de seus grupos diretamente entre si. A Figura 2.6 mostra um exemplo de gerenciamento distribuído, onde as arestas representam a comunicação entre as entidades. Neste exemplo, para reduzir o número de troca de mensagens, existe um gerenciador associado a cada grupos de quatro trabalhadores (ao invés de cada trabalhador ter um gerenciador próprio) onde cada um deve comunicar-se diretamente com cada um dos outros e os trabalhadores dentro de cada grupo comunicam-se entre si.

O outro tipo de estratégia é através do gerenciamento hierárquico. Os gerenciadores são geralmente estruturados da seguinte forma: um gerenciador raiz (nível 0) que

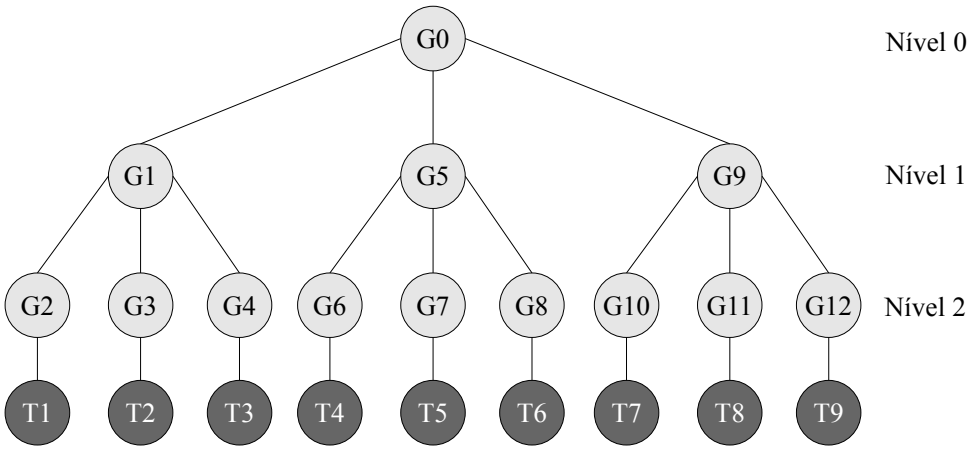


Figura 2.7: Hierarquia de gerenciadores *WorkStealing*.

apresenta uma visão completa de todos os outros gerenciadores, uma certa quantidade de gerenciadores no nível 1 da estrutura hierárquica cada um com visão de gerenciadores abaixo, uma certa quantidade de gerenciadores no nível 2 da estrutura hierárquica e assim sucessivamente até o último nível desejado. A Figura 2.7 mostra um exemplo de estrutura que segue o modelo hierárquico. Existe um gerenciador em cada nível que apenas se comunica com seu pai e seus filhos. A cada gerenciador folha, há um processo trabalhador associado.

2.3.3 Implementações Existentes

Três implementações de algoritmos *branch-and-prune* e *branch-and-bound* em paralelo serão descritas nesta seção. Elas são: GrADSAT [15], *Workstealing* hierárquico [57] e Steiner distribuído [19]. Cada uma é classificada, neste trabalho, como mestre-trabalhador, *workstealing* hierárquico e *workstealing* distribuído.

O trabalho GrADSAT [15] utiliza a abordagem mestre-trabalhador no algoritmo da aplicação, fazendo o uso de um *middleware* que fornece um conjunto de ferramentas para ajudar no gerenciamento de aplicações (caracterizado com um AMS em [50]) - o GrADS (Grid Application Development Software) [24]. Este trabalho implementa uma versão mestre-trabalhador para resolver instâncias do problema de Satisfabilidade (SAT), um problema do tipo *branch-and-prune*. Como a maior parte do gerenciamento está embutido no código da aplicação, o trabalho chama o sistema de GrADSAT, ou seja, uma versão do algoritmo paralelo para grades computacionais para o problema SAT. Através de informações disponibilizadas por ferramentas de monitoramento do GrADS, o mestre é capaz de gerenciar os trabalhadores e distribuir o trabalho ou *jobs* da aplicação entre eles. Inicialmente, os processos trabalhadores são instalados nas máquinas pertencentes à grade e eles ficam esperando *jobs* do mestre. O mestre distribui gradativamente os *jobs*,

conforme a disponibilidade de recursos (pode ser que nem todas as máquinas sejam utilizadas). Toda vez que a decisão de divisão de trabalho é feita, o mestre usa informações de monitoramento de recursos - um monitor centralizado (*server*) - para determinar qual a melhor máquina para transferir o *job*. No entanto, mesmo com as melhorias fornecidas através do *middleware* GrADS (com um certo grau de autonomia), esta abordagem apresenta os mesmos problemas das outras estratégias mestre-trabalhador (ver Seção 2.3.1).

O trabalho apresentado em [57] propõe uma estratégia de balanceamento de carga hierárquica para aplicações MPI-2 [26] baseada em *workstealing*. A proposta é apropriada para aplicações que seguem o modelo de programação de divisão e conquista, como é o caso dos algoritmos *branch-and-prune* e *branch-and-bound*. A estratégia trabalha com uma fila de tarefas que são distribuídas entre gerenciadores. Os gerenciadores estão organizados de forma hierárquica e cada um tem o objetivo de gerenciar as tarefas dos processos em cada processador envolvido. A hierarquia de gerenciadores pode ser vista como uma árvore n-ária. Apenas processos trabalhadores associados aos gerenciadores folhas executam a computação do problema. Cada gerenciador folha possui as tarefas de seu processo trabalhador e os gerenciadores pais têm as tarefas de todos os seus filhos. Quando um processo trabalhador fica ocioso, o gerenciador detecta isto e pede mais unidades de trabalho aos seus gerenciadores acima, que, por sua vez, podem "roubar" trabalho de outros gerenciadores. A terminação deste algoritmo é reconhecida pelos gerenciadores quando não há mais unidades de trabalho. A proposta deste trabalho não apresenta suporte a tolerância a falhas.

Em [19], um algoritmo *workstealing* distribuído *branch-and-bound* para o problema de Steiner em grafos foi desenvolvido para a execução sobre ambientes de grades computacionais. Este algoritmo não se utiliza de um *middleware* para a execução, mas inclui mecanismos de balanceamento de carga e tolerância a falhas integrado ao algoritmo. Os procedimentos adotados para executar o algoritmo de Steiner distribuído são: distribuição inicial, balanceamento de carga, detecção de terminação, comunicação do *primal bound* e tolerância a falhas. Na distribuição inicial, grupos de processos são associados a *clusters* previamente definidos, cada um com seu processo líder, e os trabalhos iniciais são distribuídos, começando pelo processo inicial. Conforme os processos terminam, o trabalho deles pedem mais trabalho para outros processos do mesmo *cluster* através de troca de mensagens. Se não houver trabalho disponível, o pedido é feito a outro *cluster* através de comunicação entre líderes. Então, este procedimento de pedidos de trabalho é um balanceamento de carga reativo. Quando todos os *clusters* terminam, deve haver um consenso para finalizar a execução, já que cada *cluster* não sabe exatamente o que acontece em outro. Isto é feito através do procedimento de detecção de terminação. O procedimento de comunicação do *primal bound* é feito entre os *clusters* através de mensagens de *broadcast* entre líderes. O mecanismo de tolerância a falhas é implementado através de técnicas de

checkpoint com mensagens enviadas periodicamente para o líder.

Os três principais trabalhos relacionados - GrADSAT, *Workstealing* hierárquico e Steiner para grades - assemelham-se à proposta apresentada neste trabalho (ver Capítulo 3) em relação a tentar resolver o problema de distribuição do trabalho ou carga entre processadores. Aplicações que utilizam a técnica *branch-and-prune* e *branch-and-bound* beneficiam-se do escalonamento *on-line* de tarefas, geralmente de tamanhos diferentes, entre os processadores. Se tratando de sistemas distribuídos maiores, como grades computacionais, um mecanismo de balanceamento de carga dinâmico não é o único fator que determina um melhor desempenho de aplicações. É importante também existir um mecanismo de tolerância a falhas eficiente integrado com o sistema de escalonamento.

Tabela 2.1: Resumo da comparação dos trabalhos relacionados com este.

Trabalho	Tipo de Estratégia	Tipo de escalonamento	Uso de *Middleware*	Tolerância a falhas	Modelo de paralelização
Steiner para grades	*Workstealing* Distribuído	reativo	não	sim (checkpoint)	1PProc
WorkStealing Hierárquico	*Workstealing* Hierárquico	reativo	não	não	1PProc
GrADSAT	Mestre-Trabalhador	reativo	sim (GrADS)	não	1PProc
Proposta deste trabalho	*Workstealing* hierárquico	proativo	sim (EasyGrid)	sim (recriação)	1PTask

Existem três diferenças básicas entre a proposta deste trabalho e os três artigos descritos nesta seção. Primeiramente, neste trabalho, a implementação do gerenciamento de processos (escalonamento dinâmico de tarefas, mecanismos de tolerância a falhas e outros) é feita por um *middleware* sem acrescentar complexidade ao código da aplicação. No caso do GrADSAT, é feito o uso de um *middleware*, mas ele não apresenta algumas características de autonomia, como autorrecuperação. Segundo, na proposta deste trabalho, o escalonamento utilizado é proativo, ou seja, o escalonador reescalona as tarefas antes dos processadores ficarem ociosos. A terceira diferença encontra-se na estratégia utilizada para paralelizar as aplicações. A estratégia de paralelização tradicional consiste na existência de um processo por processador que recebe e calcula as tarefas. Na estratégia descrita neste trabalho, baseada no modelo alternativo de execução 1PTask [61], é criada uma quantidade de processos maior que o número de processadores disponíveis para calcular as tarefas, tirando proveito de ambientes dinâmicos e compartilhados. A Tabela 2.1 mostra um resumo da comparação dos três trabalhos citados com a proposta deste trabalho.

2.4 Resumo

Neste capítulo, os algoritmos *branch-and-prune* foram descritos assim como alguns problemas importantes que utilizam tal técnica para serem resolvidos. Em seguida, diversos aspectos da computação autônoma foram abordados, descrevendo alguns trabalhos da literatura que tratam disto. Ainda, uma ferramenta importante usada neste trabalho que fornece autonomia às aplicações é apresentada - o *middleware* EasyGrid AMS - citando as principais diferenças deste *middleware* para os demais. No fim, alguns trabalhos da literatura que realizam estratégias de paralelização em ambientes de larga escala foram descritos e comparados com a proposta deste trabalho.

O próximo capítulo irá descrever a estratégia de paralelização proposta neste trabalho e explicará como é a interação entre a estratégia e o *middleware* gerenciador de aplicações. O algoritmo genérico paralelo para aplicações branch-and-prune será apresentado detalhadamente.

Capítulo 3

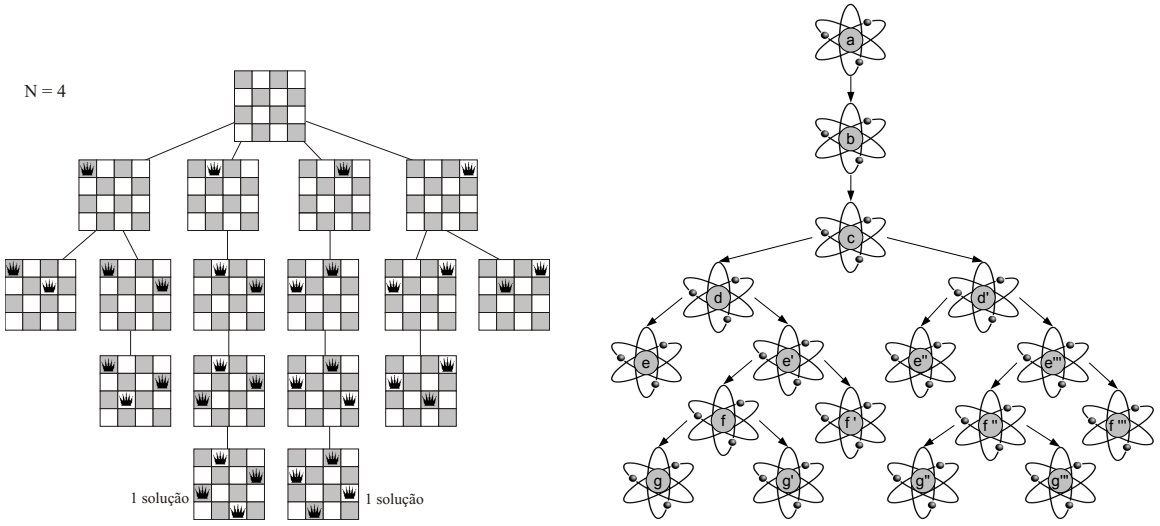
Paralelização de Aplicações Branch-and-Prune

Este capítulo irá descrever uma estratégia de paralelização de aplicações *branch-and-prune*, onde o objetivo é ter um desenvolvimento simples e baseado no algoritmo *branch-and-prune* sequencial. Com isso, o esforço de projetar o algoritmo paralelo seria relativamente pequeno. Além deste objetivo, um elemento fundamental desta estratégia encontra-se na utilização do *middleware* EasyGrid AMS (ver Seção 2.2.2) para gerenciar e atribuir autonomia a aplicação para que ela possa aproveitar os recursos computacionais eficientemente.

Os algoritmos *branch-and-prune* têm a característica de percorrerem o espaço de busca de solução do problema através de uma busca em profundidade em árvore. Esta árvore é construída dinamicamente, conforme as possibilidades de solução vão surgindo. Tais possibilidades geram novos ramos da árvore que são caracterizados como subproblemas (ver Seção 2.1). De acordo com informações extras ou restrições do problema, diversas podas (eliminação de ramos) são efetuadas, diminuindo o espaço de busca original.

Dois exemplos de problema *branch-and-prune* citados e descritos na Seção 2.1 são o N-rainhas e o PDGDM. Ambos geram uma árvore de busca em profundidade considerando informações de poda. A Figura 3.1(a) mostra uma árvore de busca para $N = 4$. Neste problema, cada nível está associado a uma linha do tabuleiro (exceto pela raiz) e a altura da árvore é o valor de N. Cada nó contém uma rainha em uma posição viável do tabuleiro na linha em questão. As folhas no último nível da árvore representam as soluções do problema. Na Figura 3.1(b), cada nó representa um átomo da cadeia na ordem. Os três primeiros nós são valores fixos, já que o ângulo de torção só é calculado após três átomos anteriores já posicionados. As ramificações ocorrem por conta dos dois valores possíveis para o ângulo de torção. No nível igual a altura da árvore, dada pelo número de átomos da cadeia principal da molécula, são encontradas as soluções.

Claramente, cada nó da árvore representa a escolha de uma parte da solução e um



(a) N-Rainhas **(b) PDGDM**

Figura 3.1: Exemplos de árvores de busca em profundidade.

caminho da raiz até a altura máxima desta árvore é uma solução para o problema. Através de informações do problema, ramos da árvore podem ser podados de modo a reduzir o espaço de busca. No caso do N-Rainhas a poda é feita através das restrições do problema, enquanto no PDGDM a poda é realizada através do conhecimento de informações extras (além das distâncias obrigatórias entre cada par de 4 átomos consecutivos, uma instância pode conter distâncias entre outros pares de átomos). Assim, o resultado é uma árvore de busca construída dinamicamente, podendo ter ramos menores e maiores.

A busca em profundidade executada na maioria destes problemas combinatoriais é geralmente realizada através dos algoritmos de *branch-and-prune* e *branch-and-bound*. O algoritmo *branch-and-prune* é usado para encontrar todas as soluções, excluindo ramos que são inviáveis de acordo com as informações do problema. O algoritmo *branch-and-bound* estabelece limites superiores ou inferiores para a solução ótima. De qualquer forma, ambos precisam realizar uma busca que, dependendo dos parâmetros de entrada, pode durar muito tempo, já que os problemas apresentam um grande espaço de busca mesmo após as podas. Estes problemas de larga escala são normalmente NP-difíceis.

Generalizando, tais buscas podem ser representadas através do Algoritmo 1. Este algoritmo recursivo recebe como parâmetros de entrada um nó e um nível da árvore. Inicialmente, o algoritmo calcula os nós filhos do nó atual (linha 1), respeitando eventuais restrições do problema. Se o último nível da árvore for atingido, uma solução foi encontrada (linha 2 e linha 3). Para cada nó filho realiza-se uma nova busca recursivamente (linha 6). A busca para quando não houver mais nós filhos, cancelando a execução do *loop* na linha 5. Para iniciar a execução do algoritmo, os parâmetros atribuídos são o nó raiz da árvore e o nível 0.

Algoritmo 1: Algoritmo sequencial *branch-and-prune* generalizado.

Nome: `buscaSolucaoSeq()`
Entrada:
no - um nó da árvore
nivel - nível da árvore

```
1 calcula filhos de no
2 se nivel é o último então
3 |  retorna solução
4 fim
5 para cada no_filho de no faça
6 |  buscaSolucaoSeq(no_filho, nivel + 1)
7 fim
```

3.1 Técnica de Paralelização

A técnica de paralelização proposta neste trabalho consiste em dividir a aplicação em diversas tarefas, cada uma representando um ramo da árvore de busca a partir de um nível [55]. As tarefas são processos criados dinamicamente pelo seu processo pai, que faz a divisão considerando o nível em que se encontra na árvore. A criação dinâmica de processos em um algoritmo *branch-and-prune* é mostrada no Algoritmo 2. Cada processo criado (inclusive o inicial) executa os passos do algoritmo *buscaSolucaoPar* descrito em 2. Além dos parâmetros de entrada do Algoritmo 1, a versão paralela possui um novo parâmetro chamado *nivel_criacao*. Ele representa o nível da árvore no qual as tarefas devem ser criadas. Cada processo, inclusive filho, executa este algoritmo. O processo pai inicial introduz a construção da árvore de busca até alcançar o *nivel_criacao* (linha 5), quando então efetua a criação de processos filhos que continuarão a busca em cada ramo resultante. Ainda, antes da criação, o valor de *nivel_criacao* é modificado para o valor de nível de corte daquele ramo (linha 6), com o intuito dos novos processos conhecerem o nível onde criar seus respectivos processos filhos.

A Figura 3.2 mostra um exemplo de como a técnica de divisão proposta poderia melhorar o desempenho da aplicação, considerando apenas seu próprio desbalanceamento de carga. Na figura, são representados a árvore de busca da aplicação e o estado dos processadores em cada unidade de tempo (representado pelos retângulos abaixo de cada processador). No exemplo, supõe-se que a computação de cada nó da árvore ocupa uma unidade de tempo. Logo, o processamento de toda a árvore ocupa 16 unidades.

Na Figura 3.2(a), uma divisão simples é realizada no primeiro nível da árvore, resultando em 3 partes ou tarefas: uma de 10, uma de 2 e outra de 3 unidades de tempo. O processo inicial calcula o problema e descobre os seus nós filhos (linha 4 do Algoritmo 2). O nível de criação, neste caso, é 1 e, portanto, os 3 filhos deverão ser criados. Antes de

Algoritmo 2: Realiza a estratégia de criação dinâmica para algoritmo *branch-and-prune*.

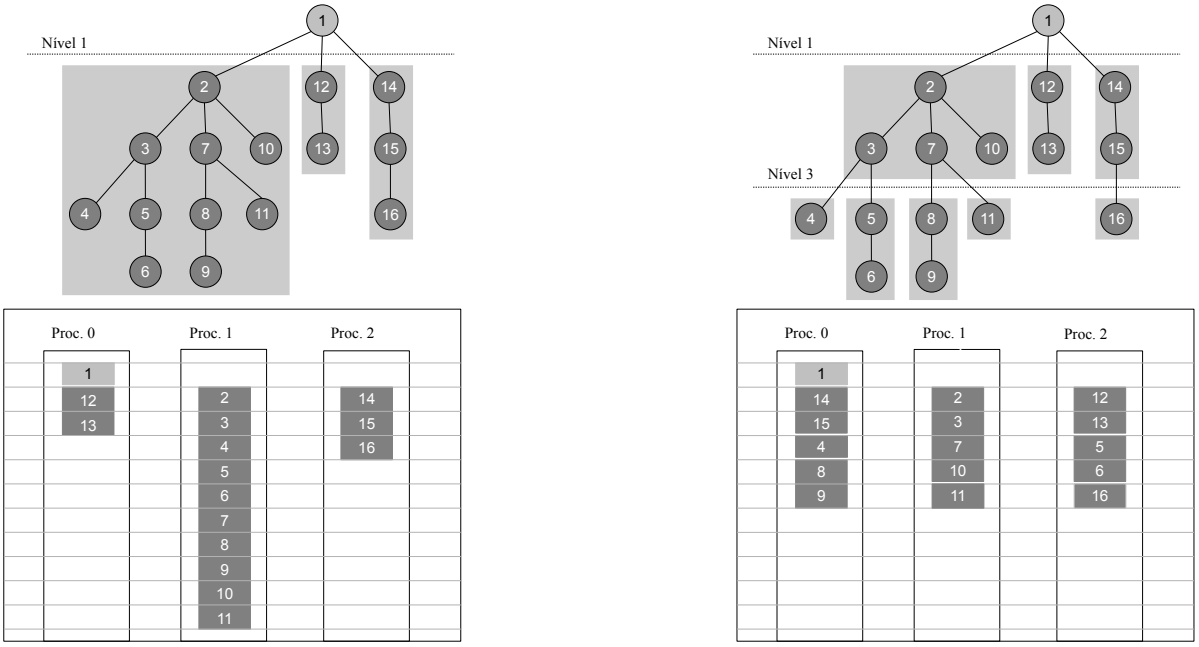
Nome: `buscaSolucaoPar()`
Entrada:
no - um nó da árvore
nivel - nível da árvore
nivel_criacao - nível em que novas tarefas serão criadas

```
1  calcula filhos de no
2  se nivel é o último então
3  |  retorna solução
4  fim
5  se nivel = nivel_criacao então
6  |  modifica nivel_criacao
7  |  para cada no_filho de no faça
8  |  |  criaNovaTarefa(no_filho, nivel + 1, nivel_criacao)
9  |  fim
10 senão
11 |  para cada no_filho de no faça
12 |  |  buscaSolucaoPar(no_filho, nivel + 1, nivel_criacao)
13 |  fim
14 fim
```

criar os novos processos ou tarefas, o nível de criação deve ser modificado para um valor infinito (linha 6 do Algoritmo 2), para que não haja mais cortes. Pode-se reparar que a primeira tarefa é maior que as demais. Isto é bastante comum nos algoritmos *branch-and-prune* e *branch-and-bound*, pois a quantidade de podas realizadas em cada ramo não é uniforme. Como o tempo de execução do melhor escalonamento possível é limitado inferiormente pelo tamanho da maior tarefa, são necessárias, ao menos, 10 unidades de tempo, além da primeira, referente à computação do nó raiz.

Utilizando-se mais um nível de criação, o número de tarefas aumenta enquanto a granularidade diminui. Na Figura 3.2(b), a árvore é separada no primeiro nível, gerando 3 tarefas. Cada uma destas tarefas receberá o nível de criação (no valor igual a 3) e criarão suas respectivas tarefas. Sem alterar o número de processadores, pode-se conseguir um tempo de execução menor. A Figura 3.2(b) mostra, ainda, que é possível obter um escalonamento de 6 unidades de tempo através desta divisão. A primeira unidade de tempo é gasta computando o nó raiz. Assim como na divisão com apenas um nível, nesta unidade, dois processadores ficam ociosos. Em seguida, a computação dos nós restantes continua até o nível 3 e, a partir dele, mais tarefas são criadas. Uma simples divisão em mais um nível gera 8 tarefas de granularidades mais uniformes e resulta em uma redução de 45% do tempo de processamento em relação ao exemplo da Figura 3.2(a).

Como a árvore de busca geralmente cresce exponencialmente, a quantidade de tarefas



(a) Um nível. **(b) Dois níveis.**

Figura 3.2: Exemplos de divisões em uma árvore de busca.

pode ser muito grande, dependendo dos níveis utilizados para a criação. Além disso, quanto maior o primeiro nível de criação, menor a granularidade das tarefas criadas e maior o tamanho da tarefa pai inicial. Em uma análise de alto nível de abstração, a escolha dos valores de nível de criação depende da aplicação e da quantidade apropriada de processos para a quantidade de recursos disponíveis (ver Subseção 3.3).

3.2 Estratégia Paralela e o EasyGrid AMS

A principal meta desta técnica é a simplicidade da implementação do algoritmo, focando na criação de uma grande quantidade de tarefas de granularidade menor e mais uniforme. A aplicação, que possui características dinâmicas devido à não uniformidade dos ramos de sua árvore de busca, consegue, então, executar em ambientes distribuídos dinâmicos de forma autônoma junto ao EasyGrid AMS. Através da criação dinâmica de processos, esta técnica é capaz de tirar proveito do escalonador dinâmico do *middleware* e a criação pode ser feita de qualquer ponto de execução do algoritmo paralelo.

O EasyGrid AMS escalona as tarefas considerando que elas possuem pesos ou cargas iguais, já que tais pesos não são conhecidos a priori. Utilizando informações do sistema, como o poder computacional das máquinas, os escalonadores dinâmicos da hierarquia do *middleware* escolhem proativamente o melhor processador para escalonar as novas tarefas e consegue se adaptar às mudanças do ambientes e à própria irregularidade no tamanho das tarefas da aplicação. Ainda, com um número maior de tarefas (seguindo o modelo

1PTask), os escalonadores têm mais flexibilidade em realocá-las entre os processadores e a distribuição das novas tarefas torna-se mais uniforme. Com isso, o objetivo de melhorar o desempenho de aplicação *branch-and-prune* pode ser alcançado através da autonomia fornecida pelo *middleware*.

3.3 Escolha do Nível de Corte

O nível de corte na árvore determina o número de tarefas a serem criadas e executadas através do *middleware*. Quanto maior o nível, a tendência é que o número de tarefas aumente, pois o crescimento da árvore é exponencial. Junto a isso, as tarefas passam a ser menores (ter menos trabalho) e tendem a ter granularidades semelhantes, apesar de uma grande parte das tarefas terem granularidade bem pequena (as tarefas que podam boa parte de seu trabalho). Além disso, as tarefas podem surgir em momentos diferentes. Se mais de um nível de corte for utilizado, o primeiro nível determinará um conjunto de tarefas e essas tarefas determinaram outras novas tarefas em um momento posterior, assim ocorrendo para outros níveis e tarefas.

A escolha do nível de corte na árvore é uma decisão importante da estratégia desta proposta e deve ser escolhido de acordo com o problema. Se o problema apresenta uma árvore de busca com um fator de ramificação alto (no caso do problema N-rainhas o fator de ramificação é N), o primeiro corte deve ser feito no início, pois o número de possibilidades, mesmo com a poda, irá aumentar em uma escala muito grande. Os outros cortes também não podem ser muito longe da raiz. Se o fator de ramificação for pequeno (no caso do PDGDM o fator é 2), a sugestão é de o primeiro nível de corte ser mais afastado da raiz, porque o grau de crescimento da árvore é menor. Se houver o conhecimento da possibilidade de um grande número de podas, este nível pode ser ainda maior. Os próximos níveis também poderão estar mais distantes da raiz, mas não demasiadamente.

O objetivo dos cortes é fazer com que o número de tarefas seja pequeno o suficiente para esconder a sobrecarga de gerência e grande suficiente para adquirir uma boa paralelização. Não há um fator determinístico que indique o valor do nível de corte. Ele deverá ser escolhido de acordo com as características do problema como a altura da sua árvore de busca, o fator de ramificação e o número de informações conhecidas para realizar-se a poda. No entanto, é uma parte do trabalho que deve ser tratada em trabalhos futuros para automatizar a escolha do nível de corte sem que o usuário tenha que se preocupar com isto.

3.4 Resumo

Neste capítulo, a proposta de paralelização dos algoritmos *branch-and-prune* foi apresentada, abordando os aspectos relacionados como o uso do *middleware* e a escolha de parâmetros de entrada do algoritmo. Ainda, alguns exemplos de execução são mostrados para explicar de forma ilustrativa o comportamento das tarefas em um ambiente distribuído sob esta paralelização.

O próximo capítulo tem o objetivo de apresentar os experimentos efetuados para avaliar a proposta em relação aos *speed-ups* obtidos e aos aspectos autônomos de auto-otimização. A proposta será avaliada considerando o dinamismo da própria aplicação e em um cenário de ambiente compartilhado dinâmico.

Capítulo 4

Avaliação de Desempenho

Uma avaliação foi realizada utilizando-se um *cluster* de computadores homogêneos. Infelizmente não foi possível, ainda, realizar testes em uma grade computacional real como era o objetivo. Mesmo assim, já é possível analisar o comportamento do *middleware* EasyGrid AMS executando aplicações *branch-and-prune* através da análise dos *speed-ups* (já que a própria aplicação apresenta um certo grau de dinamismo devido às diferentes granularidades das tarefas) e de cenários que incluem carga extra no sistema (determinando um certo dinamismo no sistema computacional).

O *cluster* utilizado para os testes é composto por 21 computadores conectados por uma rede local de alta velocidade (Gigabit Ethernet). Essas máquinas utilizam a versão do MPI/LAM 7.1.4 e possuem processadores monoprocessados Pentium IV 2,6 GHz. Os testes executados neste *cluster* foram realizados de modo exclusivo.

A distribuição dos gerenciadores do EasyGrid AMS (ver Seção 2.2.2) é simples: o gerenciador global e o gerenciador do único *site* executa em uma das máquinas, além de 20 gerenciadores de máquina em cada um dos 20 computadores. Ou seja, 20 máquinas são usadas para efetivamente computar as tarefas da aplicação.

As aplicações utilizadas na avaliação foram o N-Rainhas e o PDGDM. A diferença entre as duas aplicações está na construção dinâmica da árvore de busca. Na aplicação N-Rainhas, o fator de ramificação da árvore (número de filhos por nó) é variável, limitado pelo valor de N (número de rainhas). O algoritmo percorre linha a linha do tabuleiro, ramificando a quantidade de candidatos a solução considerando rainhas que já foram posicionadas nas linhas anteriores. Na aplicação do PDGDM, o fator de ramificação é 2, pois só existem duas possíveis soluções a cada ramo (árvore binária). As duas possíveis soluções estão associadas ao ângulo de torção da estrutura [44]. Pode-se destacar então que a árvore do N-Rainhas é maior em largura e a árvore PDGDM é maior em comprimento. Isto é mostrado na Figura 3.1, onde ambas as árvores possuem 17 nós, mas a do N-Rainhas é mais larga, enquanto a do PDGDM é mais alta.

Basicamente, dois tipos de testes podem ser enumerados de acordo com seus objetivos. O objetivo do primeiro teste é avaliar, através de seus resultados, o quanto a estratégia de paralelização proposta contribui para a melhoria do tempo de execução da aplicação em um ambiente distribuído. A segunda avaliação tem o objetivo de analisar o comportamento da execução de algumas instâncias da aplicação em um ambiente compartilhado e dinâmico, mostrando o resultado do escalonador dinâmico do EasyGrid AMS em um ambiente com cargas estáticas e dinâmicas. Todas os experimentos foram executados três vezes no *cluster* e os valores apresentados são as médias que vêm acompanhadas de seu intervalo de confiança, com nível de confiança de 95%.

4.1 Resultados - N-Rainhas

Os resultados obtidos para os experimentos com a aplicação N-Rainhas englobam as seguintes instâncias: 17 a 21 (valores de N). Estes valores foram escolhidos com base no tempo de execução do algoritmo sequencial. Para $N = 16$ o tempo médio de execução sequencial é de 4 segundos e, portanto, não há a necessidade de paralelização. Acima de $N = 21$, os tempos de execução são altos - como por exemplo, para $N = 22$ o tempo de duração da execução sequencial é cerca de 2 semanas - e praticamente inviáveis para esta avaliação. Foram usados dois tipos de níveis de corte em cada instância experimental: um com apenas 1 nível e outro com 2. O primeiro nível, em ambos os tipos, é fixo e seu valor é 0 (o nível da raiz da árvore). O segundo nível de criação, no caso do tipo com 2 níveis de corte, é variado entre as instâncias nos valores 1 e 2. Logo, para cada instância, a avaliação será feita através dos níveis de corte 0, (0,1) e (0,2). Para simplificar, os valores para nível 0, 1 e 2 são usados no lugar de 0, (0,1) e (0,2), respectivamente. Os resultados de cada instância com cada nível de corte serão comparados no restante desta seção.

Os níveis de corte foram escolhidos através do conhecimento da altura da árvore e do número aproximado de nós possíveis a serem criados. A altura da árvore N-Rainhas é N (o número de rainhas) e o fator de ramificação em cada nó no nível i da árvore é, em um pior caso, $N-2$, se $i = 0$ e $N-i-1$, $\forall i$ tal que $0 < i < N$ (a cada nível i, pode-se afirmar que ao menos duas posições serão eliminadas mais $i - 1$ posições, que são referentes a rainhas já posicionadas anteriormente). Logo, é possível saber aproximadamente quantas tarefas serão criadas em um dado nível. Os níveis 0, 1 e 2 são valores razoáveis[1] pela quantidade de máquinas utilizadas no teste. O objetivo é diminuir a granularidade das tarefas aumentando-se o nível de corte da árvore, considerando que existem um *overhead* para a criação da tarefa.

[1]Nos experimentos efetuados, valores razoáveis de número de tarefas foram na ordem de 10, 100 e 1000, respectivamente para os níveis 0, 1 e 2.

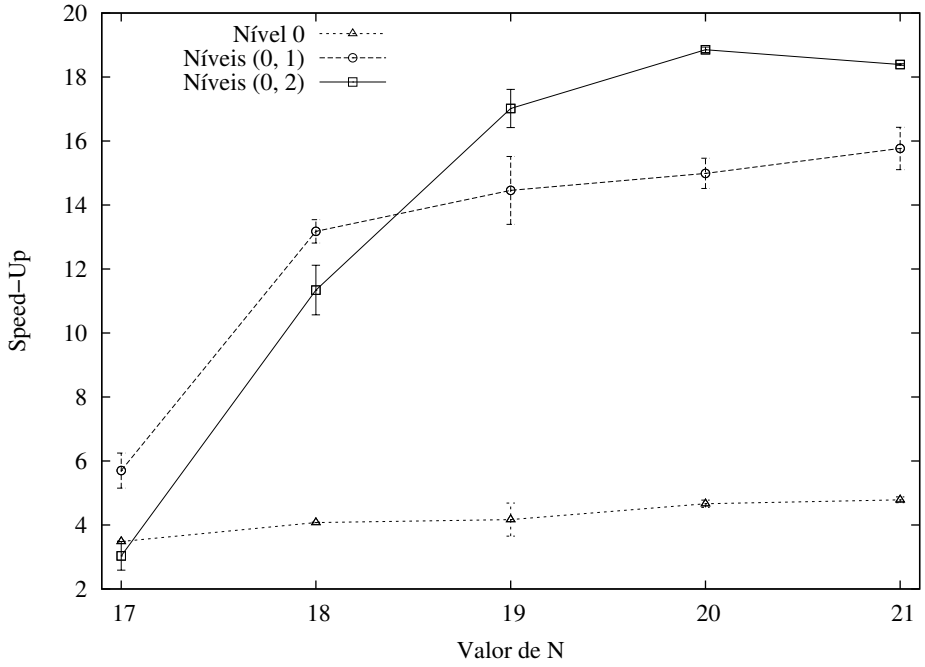


Figura 4.1: *Speed-up* obtido pela execução paralela das N-Rainhas.

A Tabela 4.1 mostra o sumário das execuções, tanto paralela quanto sequencial. A primeira coluna mostra os valores de N usados no experimento. A segunda e a quarta coluna representam a média dos tempos de parede de execução em segundos dos algoritmos sequencial e paralelos para cada valor de N e no caso do algoritmo paralelo, para cada nível de corte indicado pela terceira coluna. Assim como na quarta coluna da tabela, a quinta coluna mostra o número de processos criados para as execuções paralelas para cada tupla <N, Nível>. A sexta coluna representa o número médio de soluções encontradas por processo para os algoritmos paralelos associado, também, à tupla <N, Nível>. Este número de soluções por processo é um indicativo da quantidade média de trabalho por tarefa. Como esperado, o número de processos aumenta consideravelmente, conforme aumenta-se o nível de criação. Conforme o número de processos aumenta, a granularidade deles diminui. Isto pode ser visto na tabela pelo número de soluções encontradas por processo.

O gráfico na Figura 4.1 mostra a curva da média de *speed-ups* obtida pelas três execuções paralelas (níveis 0, (0,1) e (0,2)) com seus respectivos intervalos de confiança. Usando os pares de níveis (0,1) e (0,2), consegue-se obter *speed-ups* muito bons, principalmente para valores altos de N. Isto acontece porque a quantidade de tarefas é grande em relação ao número de máquinas e quanto maior o nível de corte, mais as tarefas tendem a ter granularidade mais finas. Assim, através do escalonador do EasyGrid AMS, as tarefas são escalonadas e aproveitam melhor os recursos. Já com o corte apenas no nível 0, o número de tarefas é bem menor mas cada tarefa apresenta granularidade bastante

Tabela 4.1: Sumário dos resultados obtidos com a aplicação N-Rainhas.

N	**Tempo Sequencial**	**Nível**	**Tempo de Parede**	**Processos Criados**	**Soluções por Processo**
17	38,69	0	11,11	22	4355232,00
		(0,1)	6,81	265	361566,43
		(0,2)	12,83	2304	41586,42
18	279,83	0	68,65	24	27753776,00
		(0,1)	21,24	313	2128085,06
		(0,2)	24,71	3007	221513,34
19	2149,14	0	518,06	25	198722313,92
		(0,1)	148,90	350	14194450,99
		(0,2)	126,35	3683	1348916,06
20	17859,78	0	3830,39	27	1445525514,22
		(0,1)	1191,86	405	96368367,61
		(0,2)	947,39	4640	8411463,12
21	145259,10	0	30348,10	28	11238079382,57
		(0,1)	9217,36	447	703951281,24
		(0,2)	7898,15	5538	56819469,61

diferentes entre si, devido ao desbalanceamento da árvore de busca. No entanto, diferentemente do esperado, para a instância $N = 17$, com os níveis (0,1) e (0,2), os *speed-ups* foram muito baixos. Isto ocorre porque enquanto a quantidade de processos é alta em relação ao número de máquinas, o trabalho realizado por cada um deles é relativamente pequeno (vide Tabela 4.1). O tempo de criação dinâmica de um processo MPI dura cerca de 10 milisegundos. Logo, para processos muito curtos (tempo de execução por volta de 10 milisegundos ou até 100 milisegundos), o paralelismo não compensa o *overhead* associado. Conforme a quantidade de trabalho dos processos aumenta, o *speed-up* melhora. Por outro lado, quanto maior o número de processos para realizar uma dada quantidade de trabalho, mais proveito do paralelismo o escalonador pode tirar. Deste modo, se existem processos suficientes com quantidades razoáveis de trabalho, o *speed-up* chega próximo ao linear. Ambas as curvas de nível (0,1) e (0,2) plotadas no gráfico da Figura 4.1 ilustram este comportamento.

O gráfico da Figura 4.2 explica de um outro ponto de vista o crescimento dos valores de *speed-up*. O eixo das ordenadas representa o trabalho total realizado pelas tarefas nas execuções paralelas. Esta grandeza é definida como a razão entre a soma da média dos tempos de parede de todos os processos executados no algoritmo paralelo e o tempo de parede da execução sequencial. Ambas as versões, sequencial e paralela, executam o mesmo processamento (computam todos os nós da árvore de busca exatamente uma vez). Desta forma, idealmente esta razão deveria ser 1. No entanto, dados os *overheads* de gerenciamento, criação dinâmica de processos e troca de mensagens, o trabalho total executado pela versão paralela tende a ser maior. A medida que a granularidade das

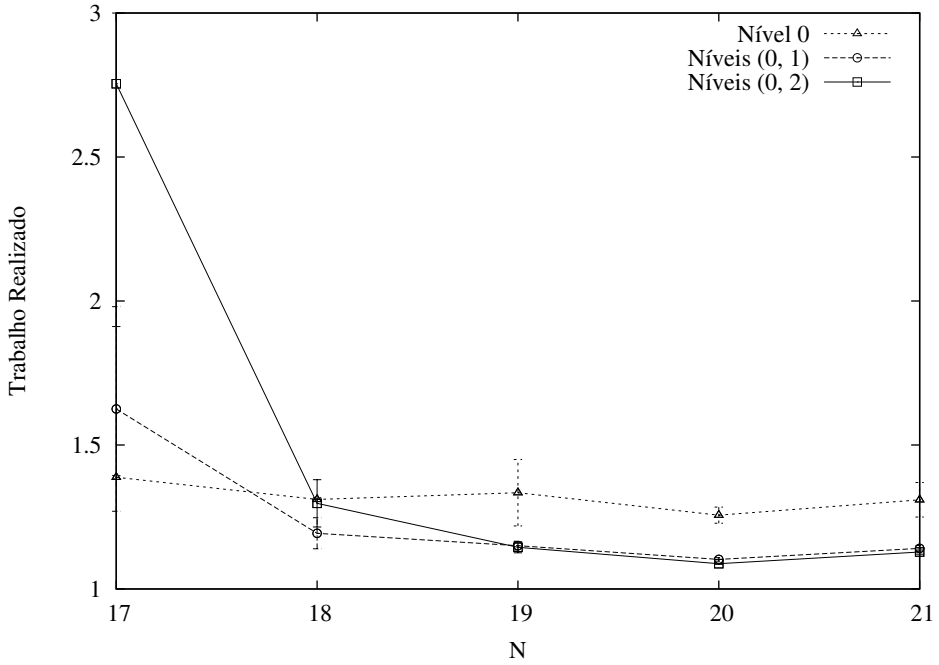


Figura 4.2: Trabalho realizado pela execução paralela das N-Rainhas.

tarefas fica mais grossa, os *overheads* diminuem proporcionalmente. Como o número de tarefas também aumenta em função do aumento de N, garante-se a existência de paralelismo e como consequência, o *speed-up* tende ao ideal.

Na Figura 4.7 cinco gráficos mostram como ficou a distribuição de trabalho pelas 20 máquinas em uma das três rodadas de execução para cada uma das instâncias 17, 18, 19, 20 e 21. O trabalho é a soma dos tempos de execução em segundos de cpu e parede (*wall clock*) em cada máquina. Com a variação dos níveis de corte, pode-se observar que quando o nível é apenas o valor 0 (apenas um valor de corte), a distribuição de tarefas é precária, enquanto para a utilização de níveis duplos (no caso, (0, 1) e (0, 2)), a distribuição de tarefas passa a ser mais uniforme. O melhor caso é para o par de níveis (0, 2), onde a distribuição é bastante semelhante entre todas as máquinas. Estes resultados mostram que a estratégia de paralelização proposta consegue dividir o trabalho de tal aplicação *branch-and-prune* gerando tarefas de granularidade menores e mais similares e, somada ao uso do EasyGrid AMS, consegue obter bom desempenho em ambientes distribuídos.

4.2 Resultados - PDGDM

Para os testes com a aplicação do PDGDM foram utilizadas 5 instâncias da base de dados PDB - *Protein Data Bank* [58], sabendo que elas demoram um certo tempo para serem solucionadas através de um algoritmo sequencial. Nos experimentos desta aplicação foram usados apenas um nível e um par de níveis de corte para a criação das tarefas. Logo,

pode-se dizer que existem os níveis de corte único e de 2 cortes.

No PDGDM, a maioria dos valores dos níveis de corte foram escolhidos de forma similar ao problema N-Rainhas. Sabendo-se, em média, quantos nós existirão em cada nível, pode-se escolher alguns valores de acordo com a quantidade de máquinas disponíveis. Para o primeiro valor de corte, estes procedimentos foram utilizados (exceto para a instância 1AYK). No entanto, diferentemente do problema N-Rainhas, no PDGDM é mais difícil conhecer as regiões de poda, dado que estes valores são diversos dados do problema, enquanto no N-Rainhas as podas estão relacionadas apenas às restrições. Para realizar escolhas melhores, o algoritmo sequencial foi executado várias vezes introduzindo um parâmetro de parada para o algoritmo. Quando ele chegar ao nível desejado (este é o parâmetro), a execução é terminada e, então, é possível saber a quantidade de tarefas que seriam criadas a partir deste nível. Porém, isto foi feito para apenas alguns valores de níveis (aqueles mais próximos da raiz), devido à demora de execução. Deste modo, os níveis de corte posteriores ao primeiro foram escolhidos.

A instância 1AYK é um caso especial. Ela apresenta uma árvore de busca diferente das outras. A árvore começa a crescer em largura exponencialmente até um certo nível e depois, ela diminui bruscamente. Por exemplo, no nível 15, só existem dois ramos e nos níveis 25 e 30, apenas 16 ramos (valores de ramos não suficientemente bons para a paralelização, trazendo baixo desempenho). Para executá-la, foram escolhidos valores de níveis de corte bem maiores (45 e $< 45, 200 >$) e esta melhoria só foi possível através de testes de execução preliminares.

Tabela 4.2: Sumário dos resultados obtidos com a aplicação PDGDM.

Instância	**Núm. de átomos**	**Tempo Sequencial**	**Nível**	**Tempo paralelo**	**Núm. de processos**	**Soluções por processo**
1AJE	582	322,71	15	34,06	64	32
			15 e 30	22,71	1088	1,88
1AJW	435	3223,97	15	212,33	256	1024
			15 e 25	200,82	4352	60,24
1AWJ	231	11696,97	15	1173,24	64	4096
			15 e 30	714,32	4160	63,02
1AWX	201	15435,66	15	1152,13	128	10240
			15 e 25	925,27	1408	930,91
1AYK	507	69694,99	45	10419,32	128	20
			45 e 200	4242,70	2176	1,18

A Tabela 4.2 mostra os resultados obtidos com as execuções sequencial e paralela variando o nível de criação de processos. A duas colunas iniciais contêm o nome da instância (na base PDB) e o número de átomos na cadeia principal da proteína, respectivamente. A terceira coluna mostra a média dos tempos da execução sequencial em segundos que, como

pode ser visto, não está diretamente associado ao número de átomos[2]. A quarta coluna apresenta os níveis (na primeira linha de cada instância está o nível único e na segunda estão os dois níveis de corte) de criação utilizados, que são diferentes entre as instâncias. A quinta coluna contém as médias dos tempos de parede em segundos das execuções paralelas e a última coluna informa, em média, a quantidade de soluções encontradas por cada processo (um indicativo do tamanho médio das tarefas).

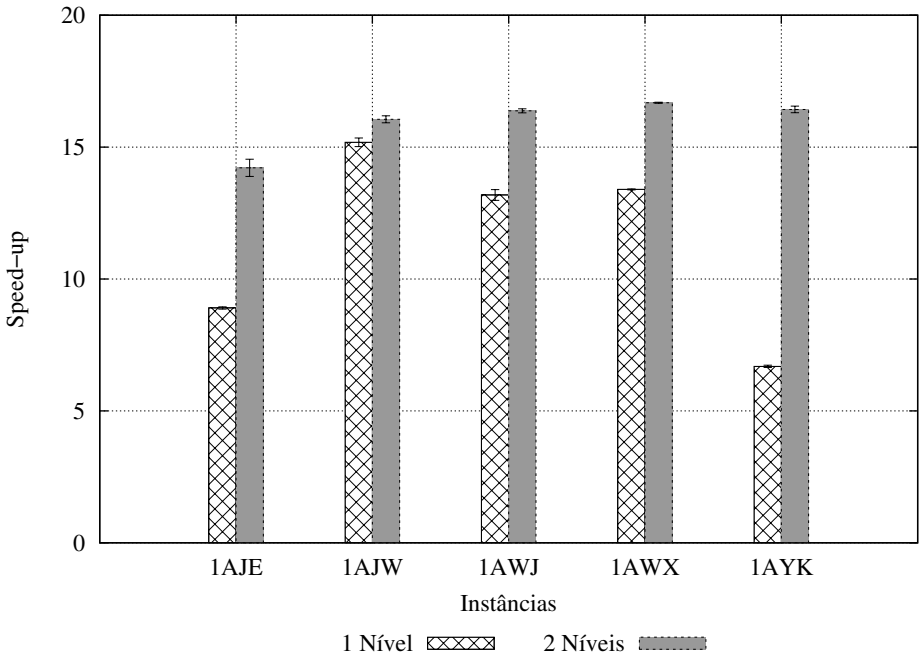


Figura 4.3: *Speed-up* obtido pela execução paralela do PDGDM.

O gráfico da Figura 4.3 mostra a média dos *speed-ups* alcançados pelas execuções paralelas em 20 processadores. Todos os *speed-ups* com os dois níveis foram melhores comparado aos níveis únicos. A razão é a mesma do problema N-rainhas: por apresentarem uma quantidade maior de tarefas com granularidade capaz de compensar o *overhead*. Os *speed-ups* relacionados aos níveis únicos de cada instância foram mais baixos e são explicados pelo *overhead* de gerenciamento geralmente associado ao número de tarefas e, no caso, principalmente a sua quantidade de trabalho. Assim como no problema N-rainhas, os valores de níveis de corte menores fazem com que a execução paralela tenha menos tarefas e elas apresentem granularidade menos uniforme entre si, devido ao desbalanceamento da árvore de busca do problema em questão. No caso da instância 1AJE, os *speed-ups* não foram tão bons (abaixo de 80% de eficiência) quanto os das outras instâncias. Isto ocorreu porque a maioria das tarefas geradas são todas curtas de tal maneira que o tempo gasto para criá-las não é compensado pelo paralelismo. Em todas as instâncias avaliadas

[2]Instâncias com uma quantidade maior de átomos podem ter uma quantidade maior de dados informativos de poda, tornando o espaço de busca menor que uma instância com menos átomos e menos informação útil de poda e portanto, com o tempo menor de execução.

do PDGDM existe este problema. Uma grande parte das tarefas leva cerca de 1 ms a 10 ms para executar e este tempo é 10 vezes menor que o tempo de criação de um processo. Isto faz com que boa parte do *overhead* fique evidente. Tais *overheads* podem ser também deduzidos através dos valores de trabalho realizados mostrados na Figura 4.4.

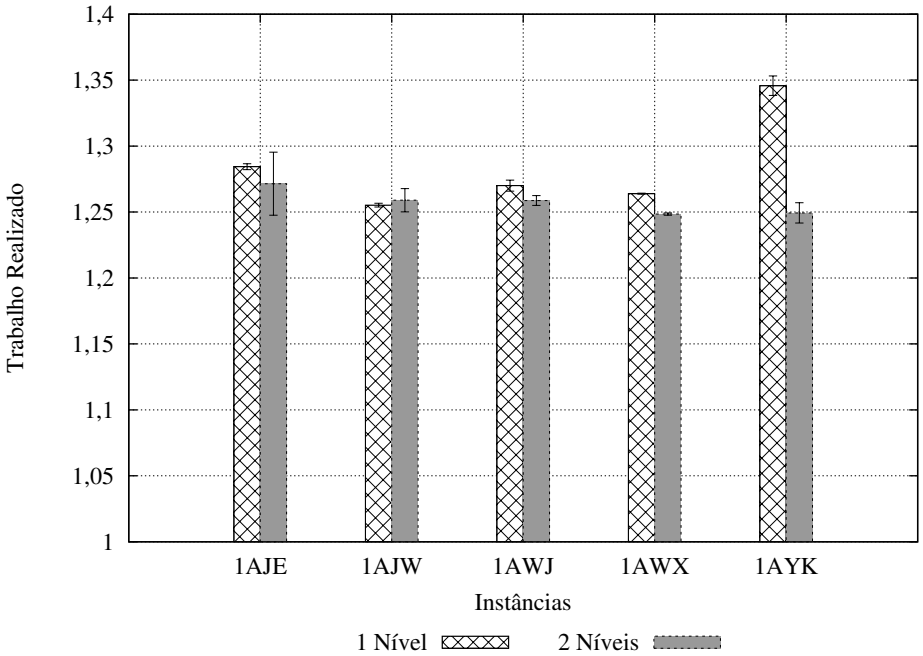


Figura 4.4: Trabalho realizado pela execução paralela do PDGDM.

A Figura 4.8 mostra também os cinco gráficos de distribuição de tarefas nas 20 máquinas utilizadas. Em todas as execuções usando os dois níveis de corte, a distribuição de tarefas é bem mais uniforme comparada às execuções com o nível único. Com isto, pode-se entender que a estratégia de paralelização usando o EasyGrid AMS consegue aproveitar bem melhor os recursos através de um bom balanceamento de carga e escalonamento de tarefas dinâmico.

4.3 Resultados com Carga Externa

Esta seção descreve os resultados obtidos através da execução do N-rainhas com o EasyGrid AMS em um ambiente compartilhado. Este ambiente é o mesmo *cluster* das avaliações anteriores mas com a introdução de cargas extras de processamento e externas à aplicação. A instância N considerada é 20 e os níveis utilizados foram (0,1) e (0,2).

Dois cenários foram explorados neste experimento: com 5 cargas estáticas e com 5 cargas dinâmicas. As 5 cargas estáticas ficam sempre executando em 5 máquinas fixas (25% do *cluster*). Já as cargas dinâmicas, como o próprio nome sugere, ficam "saltando" de máquina em máquina no período de 10 segundos iniciadas sincronamente. O objetivo é

observar o comportamento da execução com cargas dinâmicas (uma forte característica de ambientes de larga escala como o de grades computacionais), comparando com a execução com cargas estáticas.

Tabela 4.3: Comparação entre as execuções compartilhadas estática e dinâmica.

	Carga Estática		**Carga Dinâmica**		**Sem Carga**	
Níveis	Tempo	Int. Confiança	Tempo	Int. Confiança	Tempo	Int. Confiança
(0,1)	1378,12	127,57	1305,69	48,71	1191,86	38,02
(0,2)	1086,92	11,86	1073,46	14,79	947,39	4,30

A Tabela 4.3 apresenta os resultados obtidos das execuções. Através dela, é possível comparar os tempos totais de execução (em segundos) entre o experimento com a carga estática (segunda coluna) e com a carga dinâmica. Como, para cada nível de corte utilizado os tempos representados são médias de 3 execuções, os intervalos de confiança são mostrados ao lado de cada tempo de parede. Os tempos foram bem similares, considerando os intervalos de confiança, tanto na execução com carga estática quanto dinâmica. Isto mostra que o EasyGrid AMS lida bem com cargas dinâmicas.

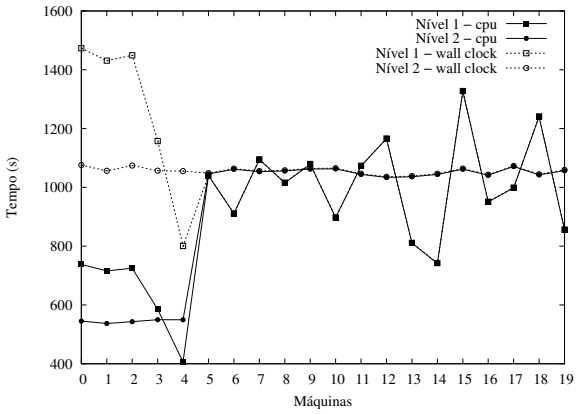


Figura 4.5: Distribuição de tarefas para a execução compartilhada estática 20-Rainhas.

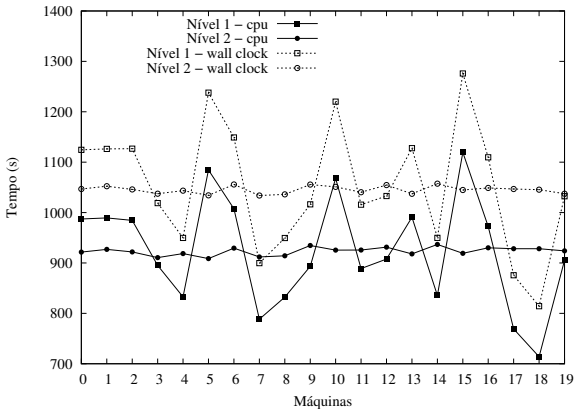


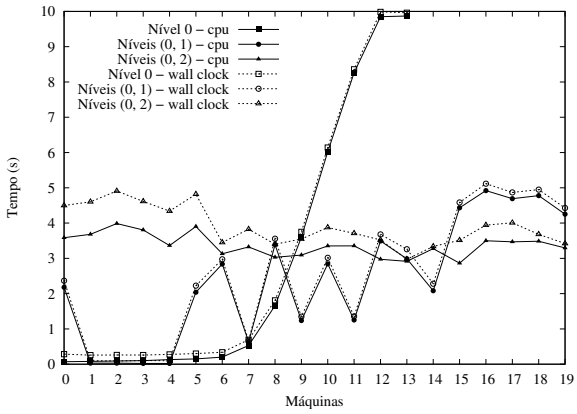
Figura 4.6: Distribuição de tarefas para a execução compartilhada dinâmica 20-Rainhas.

Os gráficos das Figuras 4.5 e 4.6 mostram, respectivamente, a configurações da distribuição das tarefas na execução com campartilhada estática e dinâmica, com ambos os níveis de corte. O eixo Y está associado à soma dos tempos de execução de cpu e parede (*wall clock*) em segundos de todas as tarefas em cada máquina. No gráfico da Figura 4.5, as 5 cargas estão concentradas nas 5 primeiras máquinas e isto explica a diferença entre o tempo de cpu e de parede. Principalmente com o par de níveis (0,2), que possui uma distribuição mais uniforme devido a boa distribuição de carga entre as tarefas, pode-se identificar a perda de processamento causada pela concorrência com as cargas extras em ambos os gráficos.

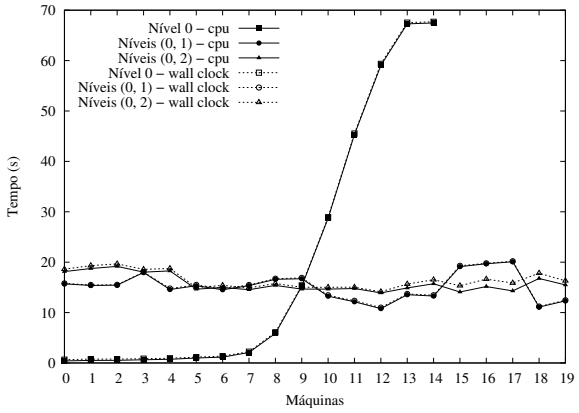
4.4 Resumo

Neste capítulo, vários experimentos foram executados com dois problemas *branch-and-prune*: o N-Rainhas e o PDGDM. Para ambos os algoritmos, os speed-ups foram calculados e as distribuições de trabalho foram exibidas através de gráficos. Com o N-Rainhas, o teste de inclusão de carga externa foi efetuado. Os resultados foram avaliados, descritos e comentados para obter-se uma conclusão da proposta apresentada.

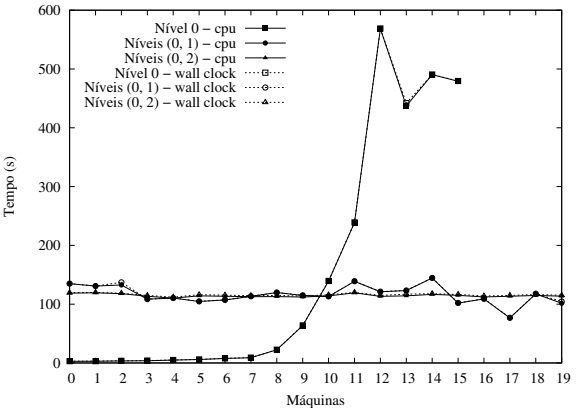
O próximo capítulo apresentará as conclusões obtidas assim como a descrição dos trabalhos futuros.



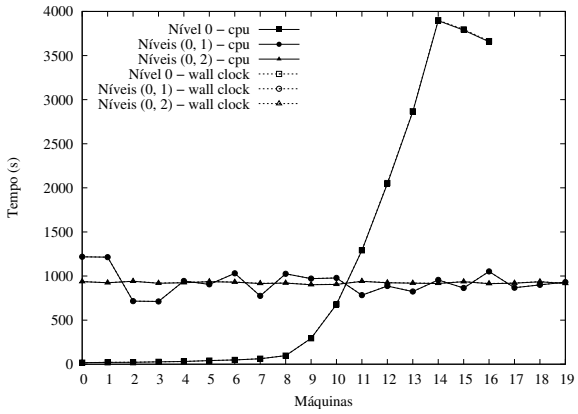
(A) $N = 17$.



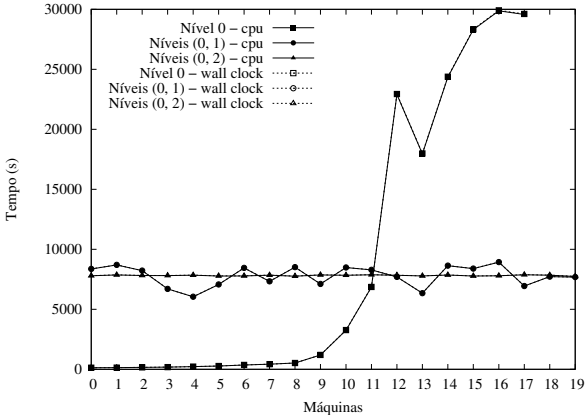
(B) $N = 18$.



(C) $N = 19$.

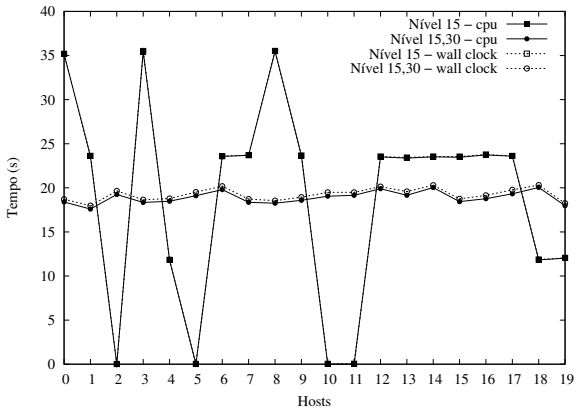


(D) $N = 20$.

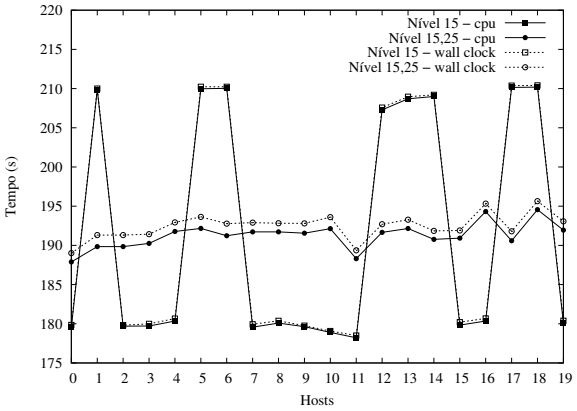


(E) $N = 21$.

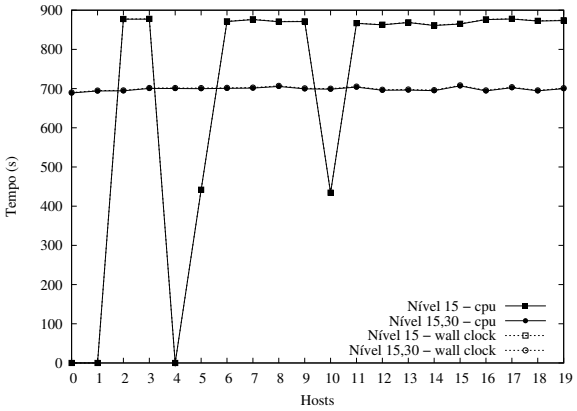
Figura 4.7: Distribuição de carga do problema N-rainhas nas máquinas.



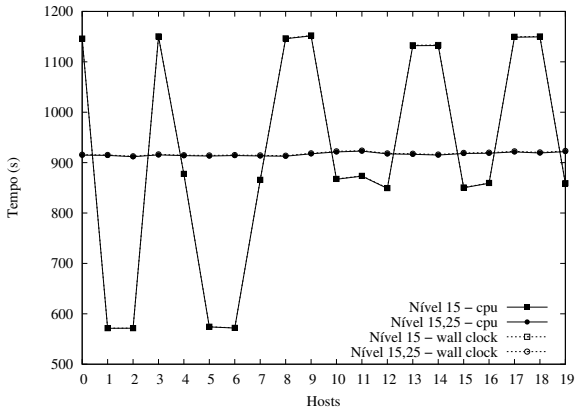
(A) Instância 1AJE.



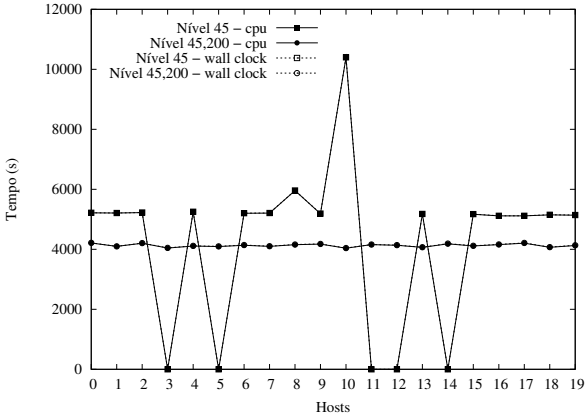
(B) Instância 1AJW.



(C) Instância 1AWJ.



(D) Instância 1AWX.



(E) Instância 1AYK.

Figura 4.8: Distribuição de carga do PDGDM nas máquinas.

Capítulo 5

Conclusão

Diversas aplicações fazem o uso de algoritmos *branch-and-prune* ou *branch-and-bound* para serem solucionadas [43, 45]. Problemas que utilizam estes tipos de algoritmos geralmente realizam buscas exaustivas que demandam muito processamento e, por esta razão, diversas soluções paralelas são propostas.

Este trabalho propôs uma estratégia de paralelização para algoritmos do tipo *branch-and-prune* para integração com o EasyGrid AMS. Esta estratégia visa expor um maior grau de paralelismo de aplicações de maneira simples e autônoma. Apenas inserindo algumas modificações no código do algoritmo sequencial, o algoritmo paralelo é obtido de modo a escalar em grandes sistemas distribuídos. A autonomia proveniente do EasyGrid AMS é uma característica importante neste tipo de estratégia, dado que boa parte dos desenvolvedores de aplicações distribuídas são cientistas que não são aptos para lidar com a complexidade de ambiente de larga escala.

A árvore de busca de algoritmos *branch-and-prune* (assim como algoritmos *branch-and-bound*) apresenta a característica de ser desbalanceada devido à poda de ramos que não satisfazem as restrições do problema. Somado ao fato dos ambientes distribuídos de larga escala atuais serem heterogêneos, compartilhados e dinâmicos, existe um grande desafio em descobrir qual a melhor maneira de se paralelizar tais problemas de forma a garantir bom aproveitamento dos recursos computacionais envolvidos.

Dado tais problemas encontrados ao desenvolver-se versões paralelas de aplicações *branch-and-prune* para ambientes de larga escala, um dos objetivos deste trabalho foi de projetar aplicações autônomas e analisar seu comportamento em ambientes distribuídos. A autonomia é dada pelo *middleware* EasyGrid AMS, um sistema gerenciador de aplicações que oferece características de autoconhecimento (*self-awareness*); autoconfiguração (*self-configuring*); auto-otimização (*self-optimizing*) e autorrecuperação (*self-healing*)[1]. Ou seja, o AMS fornece meios da aplicação se autogerenciar, fazendo com que a aplicação

[1] A parte da autoproteção é um trabalho recente em andamento.

adapte-se ao seu ambiente de execução eficientemente. Espera-se que exista um certo *overhead* de gerenciamento, mas sua interferência na execução da aplicação deve ser reduzida ao ponto de não resultar na queda do desempenho das aplicações.

Os resultados com o intuito de medir os *speed-ups* em relação a algoritmos sequenciais eficientes mostraram que, dependendo dos valores de níveis de corte realizado na árvore, pode-se obter uma boa paralelização. As tarefas são melhores distribuídas quando há um número grande de tarefas e elas apresentam trabalho suficiente para compensar o *overhead* de gerenciamento. Além disso, quando as tarefas são submetidas a um ambiente compartilhado dinâmico (onde as cargas das máquinas variam no tempo), elas podem continuar aproveitando os recursos eficientemente. A característica autônoma de auto-otimização fornecida pelo *middleware* é o grande fator responsável pelo bom uso dos recursos pela a aplicação.

Na avaliação de desempenho realizada verificou-se que com um número suficientemente grande de processos criados dinamicamente, a aplicação N-Rainhas, em conjunto com o EasyGrid AMS, obteve *speed-ups* bastante próximos ao linear (aproximadamente 19), mesmo executando sobre a camada de gerenciamento do *middleware*. Isto condiz com a teoria de que a estratégia de paralelização junto ao *middleware* permite uma boa distribuição de carga e consequentemente um bom uso dos recursos.

Com a aplicação do PDGDM, os *speed-ups* obtidos foram bons, chegando até o valor de 16,7. Usando mais de um nível de corte, a maioria dos *speed-ups* ficaram em torno de 80% do linear. Acredita-se que os *speed-ups* desta aplicação não foram tão altos quanto os do problema N-Rainhas porque o PDGDM apresenta uma grande quantidade de tarefas de granularidade muito fina (cerca de 10 milisegundos cada). Isto prejudica a execução já que o tempo de criação de uma tarefa é 10 vezes maior. Mesmo assim, os resultados mostraram que a boa distribuição de carga entre as outras tarefas afetou satisfatoriamente o resultado final da execução paralela.

Em resumo, os *speed-ups* obtidos mostram como a estratégia se comporta considerando a variação de carga da própria aplicação. Logo, o desenvolvedor da aplicação paralela terá uma facilidade maior para implementar o algoritmo, sabendo que o dinamismo da variação de carga é bem explorado e poderá ter um melhor aproveitamento em ambientes de larga escala devido à possibilidade da grande divisão de trabalho entre tarefas e à autonomia fornecida pelo *middleware*.

Neste trabalho, foi possível concluir ainda que pode-se aumentar o *speed-up* das execuções paralelas simplesmente alterando os valores de níveis de corte no algoritmo sequencial. No entanto, tais valores são escolhidos através de características previamente conhecidas de cada instância, como a altura da árvore, o número de processos a serem criados e, em algumas vezes, parte da execução efetuada (até o nível escolhido). Seria

interessante automatizar esta escolha (parte da auto-configuração e da auto-otimização da aplicação), para que o desenvolvedor não precise preocupar-se em descobrir tais valores.

5.1 Trabalhos Futuros

A próxima fase do trabalho irá focar dois aspectos. Um dos primeiros passos na continuidade deste trabalho é avaliar formas de se automatizar a escolha de bons valores de nível de corte. Tal escolha pode ser feita em tempo de execução de acordo com a quantidade de tarefas na fila de execução e/ou uma estimativa do tamanho médio das tarefas. Além disso, os níveis de corte podem não ser necessariamente valores estáticos. Eles poderiam variar entre ramos conforme a demanda da execução. A próxima fase deste trabalho seria exatamente nesta linha. Dado que teríamos múltiplos níveis de corte, há várias questões para responder como por exemplo: o intervalo entre níveis deve ser fixo ou variável?, os níveis devem ser determinados para a árvore inteira ou para cada ramo independentemente? No entanto, cabe avaliar se tais escolhas realmente trazem benefícios a execução em relação a simples alternativa estática.

Um outro ponto a ser trabalhado futuramente é a comparação desta estratégia com aquelas descritas na Seção 2.3 e também executá-la em um ambiente de grades computacionais real. Caso não seja possível obter as implementações originais dos trabalhos relacionados comentados, tais estratégias serão implementadas e testadas da mesma forma como foi feito com a proposta deste trabalho. Através destas comparações pode-se avaliar o impacto do escalonamento dinâmico proativo do EasyGrid AMS em relação aos escalonamentos reativos.

Em trabalhos futuros a longo prazo, pretende-se estender os objetivos para outros tipos de aplicações e descobrir novas formas de paralelizá-las e torná-las autônomas. Dado este maior conhecimento das aplicações a serem tratadas, pretende-se adotar estratégias de paralelização que permitam que qualquer aplicação de uma dada classe seja capaz de aproveitar eficientemente o ambiente distribuído de larga escala. Espera-se, então, obter um grupo de aplicações de diferentes classes (como a de aplicações *branch-and-prune*) com suas respectivas implementações genéricas de uma estratégia de paralelização. Com estas estratégias prontas, o próximo passo é torná-las autônomas, isto é, capazes de gerenciar sua execução em ambientes dinâmicos, heterogêneos e com recursos compartilhados. Este gerenciamento enfrentará diversos desafios em relação à estabilidade e dinamismo do sistema, ao grande grau de compartilhamento de recursos, à questão de segurança e validação de usuários. O *middleware* EasyGrid AMS atual já apresenta algumas características de autonomia, mas possivelmente muitas alterações serão necessárias devido à peculiaridade dos tipos de aplicações. Por esta razão, deve-se avaliar, projetar e docu-

mentar mais versões do EasyGrid AMS para cada tipo de aplicação, já que adotamos uma preocupação em atender sua qualidade de serviço.

Ao término desse conjunto de tarefas, espera-se obter resultados (após avaliações das aplicações selecionadas junto às versões do EasyGrid AMS) que levem a responder a questão da possibilidade de tornar aplicações inteligentes (*Smart G-Apps*). Além das aplicações serem capazes de se autogerenciar, elas poderiam se ajustar ou mudar de algoritmo ou estratégia durante a execução, de acordo com as mudanças do ambiente e informações do autogerenciamento. Isso faria com que elas pudessem melhorar ainda mais seu desempenho conforme suas necessidades. Para o desenvolvimento de tais aplicações, vários estudos e pesquisas devem ser realizados para se conhecer técnicas que atendam as necessidades desta tarefa. Junto a isso, o *middleware* necessita considerar não apenas uma aplicação executando mas também outras do próprio EasyGrid AMS. A colaboratividade e a coordenação de um número maior de aplicações EasyGrid AMS concorrendo aos recursos precisam ser obtidas para que as aplicações se comportem construtivamente como uma sociedade [59]. Muitos conceitos e técnicas precisam ainda ser estudados e avaliados nesta linha para se alcançar tal objetivo.

APÊNDICE A – Algoritmo Sequencial N-Rainhas

O algoritmo sequencial utilizado neste trabalho é baseado no algoritmo de Takaken [66]. Ele busca soluções tanto únicas (ou fundamentais) quanto totais (ou distintas).

Soluções únicas são aquelas que, através de rotações e reflexões do tabuleiro, são contadas como uma (por serem iguais). A ideia básica do algoritmo sequencial é realizar uma busca por todas as soluções únicas, e delas, através de rotações e reflexões, extrair as soluções totais. Os Algoritmos 3, 4 e 5 descrevem, juntos, como isso é feito.

Algoritmo 3: Algoritmo usado para para calcular o número de soluções totais e únicas.

Entrada:
N - número de rainhas
Variáveis Globais:
N - número de rainhas
$tab[N][N]$ - tabuleiro (matriz de booleanos)
$cont_1$ - contador que será multiplicado por 1
$cont_2$ - contador que será multiplicado por 2
$cont_4$ - contador que será multiplicado por 4
Variáveis Locais:
i - usado para controlar o uso das colunas
Saída:
$Solucao_{Unica}$ - número de soluções únicas
$Solucao_{Total}$ - número de soluções totais

```
1 inicia tab com FALSO
2 para i ← 0 até ⌈N/2⌉ faça
3 |   calculaSolucaoNrainhas (0, i)
4 |   tab[0][i] ← FALSO
5 fim
6 Solucao_Unica ← cont_1 + cont_2 + cont_4
7 Solucao_Total ← 2 * (cont_1 + 2 * cont_2 + 4 * cont_4)
```

Existem várias variáveis globais, que são utilizadas em todos os procedimentos. Uma delas é o N (número de rainhas); outra é o tabuleiro $N \times N$ (que funcionado com uma

pilha de execução), onde cada posição indica se há ou não uma rainha; e ainda existem três contadores que serão usados diferentemente para contabilizar as soluções totais.

O algoritmo inicia marcando todas as posições do tabuleiro com falso, isto é, sem rainhas. A partir da Linha 3 do Algoritmo 3, a busca é iniciada e metade dela é descartada, pois estão sendo consideradas as reflexões do tabuleiro. Por isso, na Linha 3, o número de soluções totais é duplicado.

Em seguida (Linha 3, o procedimento recursivo `calculaSolucaoNrainhas( )` é chamado, iniciando com a linha 0 e a coluna i (com i variando de 0 a $\lceil \frac{N}{2} \rceil$, o teto da metade de N), e depois, sempre após a chamada deste procedimento, esta posição global do tabuleiro é desmarcada (Linha 3).

O procedimento `calculaSolucaoNrainhas( )` é descrito pelo Algoritmo 4. Ele inicia marcando a posição do tabuleiro (*lin*, *col*) na Linha 4. Então, é verificada a altura da árvore na Linha 4: se chegou na última linha ou não.

Algoritmo 4: Procedimento recursivo que calcula o número de soluções totais e únicas.

Nome: `calculaSolucaoNrainhas()`
Entrada:
lin - linha do tabuleiro
col - coluna do tabuleiro
Variáveis Locais:
conj_col - conjunto de colunas possíveis

```
1  tab[lin][col] ← VERDADEIRO
2  se lin = N − 1 então
3  |  se tab é solução única então
4  |  |  verificarRotacao()
5  |  |  guarda solução original e rotacionadas
6  |  fim
7  senão
8  |  conj_col ← colunas possíveis na linha lin + 1
9  |  para cada col em conj_col faça
10 |  |  calculaSolucaoNrainhas(lin + 1, col)
11 |  |  tab[lin + 1][col] ← FALSO
12 |  fim
13 fim
```

Se não chegou ao fim, deve-se descer o nível da árvore. Para isso, procuram-se as posições possíveis na linha $lin + 1$ do tabuleiro. Essas posições são calculadas a partir das restrições do problema: não estar na mesma linha, coluna ou diagonal que outra rainha (no caso, as rainhas já posicionadas anteriormente). Outro critério a ser utilizado é considerando as rotações. Algumas configurações do tabuleiro são descartadas, visto que

ela já foi calculada utilizando uma rotação de uma solução já encontrada. Na Linha 4, esta procura é feita e o resultado é armazenado em um conjunto de colunas, já que a linha é fixa ($lin+1$). Desse conjunto de colunas, para cada coluna na linha $lin+1$ do tabuleiro, é feita uma chamada recursiva (Linha 4) e, em seguida, essa posição é desmarcada (Linha 4). Com isso, a busca em profundidade é realizada e a poda é feita ao se construir o conjunto de colunas possíveis.

No caso da linha *lin* indicar o fim do tabuleiro, algumas verificações devem ser feitas. Primeiro, é verificado se o tabuleiro representa uma solução única (Linha 4 do Algoritmo 4). Parte dessa verificação é feita antes, na escolha das posições possíveis, na Linha 4 do Algoritmo 4. Isto acontece porque sabe-se que, em algumas configurações do tabuleiro, a solução é certamente única [66]. Essas configurações são todas excetuando quando existe uma rainha nas posições $(N-3,\ N-1)$, $(N-1,\ 2)$, ou $(2,\ 0)$. Assim, se não existirem rainhas em algumas dessas posições, não haverá necessidade de checar se a solução é única ou não.

Caso seja uma solução única, as rotações desta solução são checadas pelo procedimento `verificarRotacao( )` na Linha 4 e, deste modo, pode-se saber a quantidade de soluções totais que essa solução única gera. A Linha 4 pode ser usada para guardar as soluções totais, caso necessário.

Algoritmo 5: Procedimento que verifica as rotações do tabuleiro e incrementa o contador de acordo com elas.

Nome: `verificarRotacao()`

```
1  se tab[0][0] = VERDADEIRO então
2  |  cont_4 ← cont_4 + 1
3  senão
4  |  se rotação de 90º for igual a original então
5  |  |  cont_1 ← cont_1 + 1
6  |  senão se rotação de 180º for igual a original então
7  |  |  cont_2 ← cont_2 + 1
8  |  senão
9  |  |  cont_4 ← cont_4 + 1
10 |  fim
11 fim
```

O procedimento `verificarRotacao( )` no Algoritmo 5 verifica as rotações da solução encontrada e incrementa os contadores de acordo com elas. Se existir uma rainha na posição (0, 0) (Linha 5), sabe-se que todas as rotações são diferentes. Assim, através das rotações, tem-se quatro novas soluções. Por esta razão, a variável *cont*4 é incrementada na Linha 5, pois ela será multiplicada por 4 no fim para se obter o número de soluções totais (Linha 3 do Algoritmo 3). Caso contrário, as rotações são testadas nas Linhas 5 e 5 do Algoritmo 5. Se a rotação de 90^o do tabuleiro for igual a original, isto quer dizer

que todas as rotações (90^o, 180^o e 270^o) dessa solução serão iguais a ela. Logo, tem-se apenas uma solução (*cont*1 é incrementado na Linha 5). Se a rotação de 180^o do tabuleiro for igual a original e a de 90^o não for (Linha 5), isto indica que a rotação de 180^o dessa solução será igual a ela e a 270^o dessa solução será igual a rotação de 90^o. Logo, existem 2 novas soluções: a original e a sua rotação de 90^o (*cont*2 é incrementado na Linha 5). Caso a rotação de 90^o e 180^o sejam diferentes da original, a de 270^o será também. Assim, existem 4 novas soluções e *cont*4 será incrementando na Linha 5.

APÊNDICE B – Demonstração do Cálculo do Ângulo de Torção entre Quatro Átomos Consecutivos

Sejam os átomos consecutivos $\{i-3, i-2, i-1, i\}$. Sejam ainda:

- $d_{i-3,i-2}$ o valor da distância entre os átomos $i-3$ e $i-2$,
- $d_{i-2,i-1}$ o valor da distância entre os átomos $i-2$ e $i-1$,
- $d_{i-1,i}$ o valor da distância entre os átomos $i-1$ e i,
- $d_{i-3,i}$ o valor da distância entre os átomos $i-3$ e i,
- θ_{i-2} o ângulo formado entre os vetores $\overrightarrow{(i-3)(i-2)}$ e $\overrightarrow{(i-2)(i-1)}$
- θ_{i-1} o ângulo formado entre os vetores $\overrightarrow{(i-2)(i-1)}$ e $\overrightarrow{(i-1)(i)}$ e
- ω o ângulo de torção.

O ângulo de torção ω pode ser calculado usando-se a lei dos cossenos aplicado sobre o triângulo qualquer representado pelos lados m, n e p na Figura B.1. Assim, tem-se a expressão representada pela Equação B.1.

$$\cos\omega = \frac{m^2 + n^2 - p^2}{2mn} \tag{B.1}$$

Para resolver a Equação B.1, os valores de p, m e n precisam ser conhecidos. Esses valores podem ser calculados através dos três triângulos retângulos representados na Figura B.1. Pelo triângulo retângulo formado pelos lados m, $d_{i-3,i-2}$ e r, sabe-se que $\sin(\pi - \theta_{i-2}) = \frac{m}{d_{i-3,i-2}}$. Sabe-se também que $\sin(\pi - \theta_{i-1}) = \frac{n}{d_{i-1,i}}$, através do triângulo retângulo formado pelos lados n, $d_{i-1,i}$ e q. Logo, tem-se os valores de m e n que são mostrados nas Equações B.2 e B.3 respectivamente.

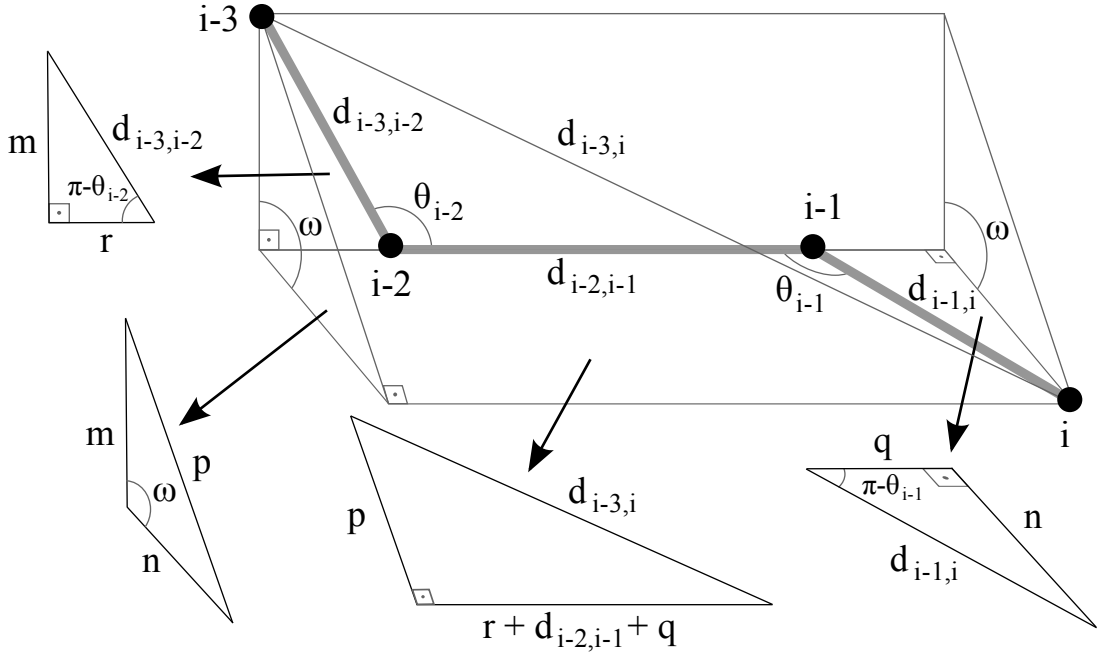


Figura B.1: Triângulos usados para o cáculo do ângulo de torção.

$$m = \sin(\pi - \theta_{i-2}) d_{i-3,i-2} = \sin(\theta_{i-2}) d_{i-3,i-2} \tag{B.2}$$

$$n = \sin(\pi - \theta_{i-1}) d_{i-1,i} = \sin(\theta_{i-1}) d_{i-1,i} \tag{B.3}$$

O valor de p pode ser calculado usando-se o Teorema de Pitágoras: $d_{i-3,i}^2 = (r + d_{i-2,i-1} + q)^2 + p^2$. Semelhante a forma que m e n foram calculados, os valores de r e q podem ser encontrados. As Equações B.4 e B.5, respectivamente, mostram os valores de r e q.

$$r = \cos(\pi - \theta_{i-2}) d_{i-3,i-2} = -\cos(\theta_{i-2}) d_{i-3,i-2} \tag{B.4}$$

$$q = \cos(\pi - \theta_{i-1}) d_{i-1,i} = -\cos(\theta_{i-1}) d_{i-1,i} \tag{B.5}$$

Esses valores são usados, assim, para obter-se o valor de p, que é dado pela Equação B.6.

$$p^2 = d_{i-3,i}^2 - (d_{i-2,i-1} - (\cos(\theta_{i-2}) d_{i-3,i-2} + \cos(\theta_{i-1}) d_{i-1,i}))^2 \tag{B.6}$$

Substituindo as Equações B.2, B.3 e B.6 na Equação B.1, resulta-se na Equação B.7 que, após algumas operações algébricas, torna-se a Equação B.8, que apresenta o valor do

cosseno do ângulo de torção.

$$\cos\omega = \frac{d^2_{i-2,i-1}+d^2_{i-3,i-2}+d^2_{i-1,i}-d^2_{i-3,i}-2d_{i-2,i-1}d_{i-3,i-2}\cos(\theta_{i-2})}{2d_{i-3,i-2}d_{i-1,i}\sin(\theta_{i-2})\sin(\theta_{i-1})} + \frac{2d_{i-3,i-2}d_{i-1,i}\cos(\theta_{i-2})\cos(\theta_{i-1})-2d_{i-2,i-1}d_{i-1,i}\cos(\theta_{i-1})}{2d_{i-3,i-2}d_{i-1,i}\sin(\theta_{i-2})\sin(\theta_{i-1})} \tag{B.7}$$

$$\cos\omega = \frac{d^2_{i-2,i-1}+d^2_{i-3,i-2}+d^2_{i-1,i}-d^2_{i-3,i}}{2d_{i-3,i-2}d_{i-1,i}\sin(\theta_{i-2})\sin(\theta_{i-1})} - \frac{d_{i-2,i-1}\cot(\theta_{i-2})}{d_{i-1,i}\sin(\theta_{i-1})} - \frac{d_{i-2,i-1}\cot(\theta_{i-1})}{d_{i-3,i-2}\sin(\theta_{i-2})} + \\ + \cot(\theta_{i-2})\cot(\theta_{i-1}) \tag{B.8}$$

Referências

[1] S.J. Aarseth. *Gravitational N-body simulations: Tools and algorithms*. Cambridge Univ. Press, 2003.

[2] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel e J. Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *International Journal of High Performance Computing Applications*, 15(4):345, 2001.

[3] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel e J. Shalf. Cactus tools for grid applications. *Cluster Computing*, 4(3):179–188, 2001.

[4] N. Andrade, L. Costa, G. Germoglio e W. Cirne. Peer-to-peer grid computing with the OurGrid Community. Em *Proceedings of the SBRC*, pp. 1–8. Citeseer, 2005.

[5] P. Andreetto, S. Andreozzi, G. Avellino, S. Beco, A. Cavallini, M. Cecchi, V. Ciaschini, A. Dorise, F. Giacomini, A. Gianelle et al. The gLite workload management system. Em *Journal of Physics: Conference Series*, volume 119, pp. 062007. Institute of Physics Publishing, 2008.

[6] K. Anstreicher, N. Brixius, J.P. Goux e J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3):563–588, 2002.

[7] ATT. The on-line encyclopedia of integer sequences `http://www.research.att.com/~njas/sequences/A000170`. Acessado em Novembro de 2009.

[8] ATT. The on-line encyclopedia of integer sequences `http://www.research.att.com/~njas/sequences/A002562`. Acessado em Novembro de 2009.

[9] AWS. Amazon elastic compute cloud (amazon ec2) `http://aws.amazon.com/ec2/`, 2009. Acessado em Novembro de 2009.

[10] Vivek Sarkar Barbara Chapman, Jesús Labarta e Mitsuhisa Sato. Programmability issues. *International Journal of High Performance Computing Applications*, 23:328–331, 2009.

[11] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq e Vijay S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009.

[12] Gordon Bell, Jim Gray e Alex Szalay. Petascale computational systems. *Computer*, 39(1):110–112, 2006.

[13] Cristina Boeres, Aline Nascimento, Vinod Rebello e Alexandre Sena. Efficient hierarchical self-scheduling for MPI applications executing in computational Grids. Em *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pp. 1–6, New York, NY, USA, 2005. ACM Press.

[14] CERN. Organisation européenne pour la recherche nucléaire `http://www.cern.ch`. Acessado em Novembro de 2009.

[15] W. Chrabakh e R. Wolski. GrADSAT: A parallel sat solver for the grid. Em *Proceedings of IEEE SC03*. Citeseer, 2003.

[16] Thomas E. Creighton. *Proteins: Structures and Molecular Properties*. W. H. Freeman, August de 1992.

[17] G. Crippen e T. Havel. *Distance Geometry and Molecular Conformation*. John Wiley & Sons, New York, 1988.

[18] Paul Cull e Rajeev Pandey. Isomorphism and the N-Queens problem. *ACM SIGCSE Bulletin*, 26(3):29–36, 1994.

[19] Lúcia Drummond, Eduardo Uchoa, Alexandre Gonçalves, Juliana Silva, Marcelo Santos e Maria Castro. A grid-enabled distributed branch-and-bound algorithm with application on the steiner problem in graphs. *Parallel Comput.*, 32(9):629–642, 2006.

[20] D. Du, J.M.G. Smith e JH Rubinstein. *Advances in Steiner Trees*. Kluwer Academic Publishers, 2000.

[21] Cengiz Erbas, Seyed Sarkeshik e Murat Tanik. Different perspectives of the N-Queens problem. Em *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pp. 99–108, New York, NY, USA, 1992. ACM Press.

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

[23] I. Foster, Y. Zhao, I. Raicu e S. Lu. Cloud computing and grid computing 360-degree compared. Em *Grid Computing Environments Workshop, 2008. GCE'08*, pp. 1–10, 2008.

[24] Keith Cooper Jack Dongarra Ian Foster Dennis Gannon Lennart Johnsson Ken Kennedy Carl Kesselman John Mellor-Crumme Dan Reed Linda Torczon Francine Berman, Andrew Chien e Rich Wolski. The GrADS Project: Software support for high-level Grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

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

[26] A. Geist, W. Gropp, E. Lusk, S. Huss-Lederman, A. Lumsdaine, W. Saphir, T. Skjellum e M. Snir. MPI-2: Extending the message-passing interface. *Europar96, Lyon (France), 26-29 Aug 1996*, 1996.

[27] B. Gendron e T.G. Crainic. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, pp. 1042–1066, 1994.

[28] Globus Alliance. The globus toolkit `http://www.globus.org/toolkit`. Acessado em Novembro de 2009.

[29] J.P. Goux, S. Kulkarni, M. Yoder e J. Linderoth. Master–worker: an enabling framework for applications on the computational grid. *Cluster Computing*, 4(1):63–70, 2001.

[30] S. L. Graham, M. Snir e C. A. Patterson. *Getting up to speed: The future of supercomputing*. Natl Academy Pr, 2005.

[31] A. Grama, V. Kumar e A. Sameh. Scalable parallel formulations of the Barnes-Hut method for n-body simulations. *Parallel Computing*, 24(5):797–822, 1998.

[32] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[33] Pascal Van Hentenryck. Constraint solving for combinatorial search problems: A tutorial. Em *CP '95: Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pp. 564–587, London, UK, 1995. Springer-Verlag.

[34] T. Hey e A.E. Trefethen. Cyberinfrastructure for e-science. *Science*, 308(5723):817–821, 2005.

[35] Bruce Hillam. *Algorithms : An Object Oriented Introduction*, capítulo Backtracking and Branch-and-Bound. 2003.

[36] E. Huedo, R.S. Montero, I.M. Llorente, D. Thain, M. Livny, R. van Nieuwpoort, J. Maassen, T. Kielmann, H.E. Bal, G. Kola et al. The GridWay framework for adaptive scheduling and execution on grids. *SCPE*, 6(8), 2005.

[37] IBM. Ibm triples performance of world's fastest, most energy-efficient supercomputer `http://www.ibm.com/`, 2007. Acessado em Janeiro de 2010.

[38] Indiana University. LAM/MPI parallel computing `http://www.lam-mpi.org`. Acessado em Novembro de 2009.

[39] Patrick Aerts Frank Cappello Thomas Lippert Satoshi Matsuoka Paul Messina Terry Moore Rick Stevens Anne Trefethen Jack Dongarra, Pete Beckman e Mateo Valero. The international exascale software project: a call to cooperative action by the global high-performance community. *International Journal of High Performance Computing Applications*, 23:309–322, 2009.

[40] Jeffrey O. Kephart. Research challenges of autonomic computing. Em *Proceedings of the 27th international conference on Software engineering*, pp. 22. ACM, 2005.

[41] Bernhard Korte e Jens Vygen. *Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics)*. Springer, 4th edição, novembro de 2007. Hardcover.

[42] Klaus Krauter, Rajkumar Buyya e Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software Practice and Experience*, 32(2):135–164, 2002.

[43] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[44] Carlile Lavor, Leo Liberti, Antonio Mucherino e Nelson Maculan. On a discretizable subclass of instances of the molecular distance geometry problem. Em Sung Y. Shin e Sascha Ossowski, editors, *SAC*, pp. 804–805. ACM, 2009.

[45] EL Lawler e DE Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[46] M. Litzkow, M. Livny e M. Mutka. Condor-a hunter of idle workstations. Em *proceedings of the 8th International Conference of Distributed Computing Systems*, volume 43, 1988.

[47] MPI Forum. Message passing interface forum `http://www.mpi-forum.org`. Acessado em Novembro de 2009.

[48] Richard Murch. *Autonomic Computing*. IBM Press, 2004.

[49] J. Nabrzyski, J.M. Schopf e J. Weglarz. *Grid resource management: state of the art and future trends*. Kluwer Academic Pub, 2004.

[50] Aline Nascimento. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. Tese de Doutorado, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, 2008.

[51] Aline Nascimento, Alexandre Sena, Cristina Boeres e Vinod Rebello. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience*, 19(14):1955–1974, Set de 2007. Published Online: 14 Nov 2006.

[52] Aline Nascimento, Alexandre Sena, Jacques Silva, Daniela Vianna, Cristina Boeres e Vinod Rebello. Managing the execution of large scale MPI applications on computational grids. Em *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)*, pp. 69–76, Rio de Janeiro, Brazil, Out de 2005. IEEE Computer Society Press.

[53] A.P. Nascimento, A.C. Sena, J.A. Silva, D.Q.C. Vianna, C. Boeres e V. Rebello. Autonomic application management for large scale MPI programs. *International Journal of High Performance Computing and Networking*, 5(4):227–240, 2008.

[54] L. Nyland, M. Harris e J. Prins. Fast n-body simulation with CUDA. *GPU gems*, 3:677–695, 2007.

[55] Fernanda Oliveira e Vinod Rebello. Algoritmos branch-and-prune autônomos. Em *SBRC 2010*, maio de 2010. A ser publicado.

[56] Manish Parashar e Salim Hariri. Autonomic computing: An overview. Em *Unconventional Programming Paradigms*, pp. 247–259. Springer Verlag, 2005.

[57] Guilherme Pezzi, Marcia Cera, Elton Mathias, Nicolas Maillard e Philippe Navaux. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. *Computer Architecture and High Performance Computing, 2007 SBAC-PAD 2007 19th International Symposium on*, pp. 247–254, Out de 2007.

[58] RCSB. Research collaboratory for structural bioinformatics `http://www.rcsb.org`, 2003. Acessado em Dezembro de 2009.

[59] Henrique Rodrigues. Grid sa: Um sociedade autônoma. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2009.

[60] Zsófia Ruttkay. Constraint satisfaction - a survey. *CWI Quarterly*, 11:123–161, 1998.

[61] Alexandre Sena, Aline Nascimento, Jacques Silva, Daniela Vianna, Cristina Boeres e Vinod Rebello. On the advantages of an alternative MPI execution model for grids. Em *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pp. 575–582, Rio de Janeiro, Brazil, 2007. IEEE Computer Society.

[62] M.R. Shirts, C.D. Snow, E.J. Sorin e B. Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68:91–109, 2003.

[63] Jacques Silva e Vinod Rebello. Low Cost Self-healing in MPI Applications. Em *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting*, pp. 144–152. Springer, 2007.

[64] Warley Silva. Algoritmos para o cálculo de estruturas de proteínas. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2008.

[65] Roy Sterritt, Manish Parashar, Huaglory Tianfield e Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187, July de 2005.

[66] Takaken. N-Queens problem (number of solutions) `http://www.ic-net.or.jp/home/takaken/e/queen`, julho de 2003. Acessado em Novembro de 2009.

[67] TOP500. Top 500 supercomputing sites `http://www.top500.org`. Acessado em Novembro de 2009.

[68] Daniela Vianna. Um sistema de gerenciamento de aplicações MPI para ambientes Grids. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2005.

[69] Vijay Pande and Stanford University. Folding@home distribuited computing `http://folding.stanford.edu/`, 2000. Acessado em Janeiro de 2010.

[70] Mark Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1):139–168, Set de 1996.

[71] Alison Wright e Richard Webb. The large hadron collider. *Nature Insight*, 7151(448):269–312, 2007.