Algoritmos *Branch-and-Prune* Autônomos

Fernanda G. Oliveira, Vinod E.F. Rebello

Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói, RJ – Brasil

{fgoliveira,vinod}@ic.uff.br

Abstract. *This paper 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 into independent sub-tasks and executed in parallel. However, the distribution of the non-uniform workloads of these sub-tasks is not trivial since they may not be known a priori. Moreover, large scale distributed systems are becoming more complex and dynamic due to their collaborative and shared natures. The proposed strategy addresses these issues by helping turn branch-and-prune applications autonomic - with the EasyGrid AMS middleware - and thus better able to take advantage of these computing environments efficiently without the need for user intervention.*

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 proposta lida com estes problemas tornando aplicações branch-and-prune mais autônomas - com o middleware EasyGrid AMS - e portanto mais capazes de tirar proveito de ambientes computacionais dinâmicos e de larga escala eficientemente sem a necessidade de intervenção do usuário.*

1. Introdução

Diversas aplicações de várias áreas de pesquisa e desenvolvimento necessitam resolver problemas com alta demanda de processamento e memória. Tais aplicações são chamadas de larga escala (especificamente de tera, peta ou exa escala) por necessitarem de recursos computacionais pouco acessíveis para muitos cientistas e pesquisadores.

Muitas dessas aplicações de larga escala usam as técnicas de *branch-and-prune* e *branch-and-bound* [Hillam 2003]. Elas são geralmente consideradas aplicações *cpu-intensive* por analisarem exaustivamente um enorme espaço de busca de soluções. Um

exemplo de aplicação que utiliza o algoritmo *branch-and-prune* é o problema de determinar a estrutura tridimensional de proteínas [Lavor et al. 2009]. Este problema é aplicado à área de bioquímica possibilitando o conhecimento de importantes informações sobre as proteínas. O problema de Steiner em grafos é um exemplo que pode ser resolvido utilizando a técnica de *branch-and-bound*. Este problema NP-difícil consiste em determinar um sub-grafo (de um grafo) conexo e de custo mínimo que contenha todos os vértices de um dado conjunto de terminais. O problema de Steiner tem diversas aplicações na área de redes de computadores, como a distribuição de conteúdo multimídia e teleconferências.

Atualmente, existem diversos tipos de sistemas distribuídos para a execução de aplicações de larga escala, alguns construídos exclusivamente para computação em alto desempenho e outros simplesmente adaptados para isto. Supercomputadores são sistemas de alto desempenho que chegam a alcançar capacidades de 1 PetaFLOPS, como é o caso do *Blue Gene/P* [TOP500]. No entanto, estes sistemas são caros para se adquirir e manter, tornando-se de difícil acesso para a maior parte da comunidade científica. Grades de computadores, por serem aglomerações de recursos geograficamente distribuídos, são tipicamente compostas de recursos heterogêneos interconectados por uma rede compartilhada. Podem compor uma grade PCs, *clusters* e até supercomputadores. Seus princípios de colaboratividade e compartilhamento de recursos entre instituições tornam este sistema bastante acessível e escalável especialmente para aplicações que necessitam de pouca comunicação entre processos paralelos. Ainda, tais ambientes de larga escala, tanto grades quanto supercomputadores, são mais suscetíveis a variações de poder computacional, de banda e latência na rede, a falhas de recursos, por exemplo. Deste modo, os ambientes distribuídos estão se tornando cada vez mais complexos e precisam 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 não são aptos para lidar com a complexidade e peculiaridade associados a esta classe de ambiente.

A computação autônoma aparece como resposta para o problema do gerenciamento de aplicações ou sistemas de larga escala [Murch 2004]. Tornando a aplicação auto-gerenciável, ela é capaz de reagir às constantes mudanças do ambiente, de detectar falhas e autorrecuperar-se, 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.

O objetivo deste trabalho é descrever uma estratégia de paralelização de algoritmos *branch-and-prune* que possibilite uma maior autonomia através de um gerenciador de aplicações chamado EasyGrid AMS. O *middleware* EasyGrid AMS é um sistema gerenciador de aplicações hierárquico que oferece meios da aplicação se auto-gerenciar. Assim, a aplicação poderá se adaptar ao sistema eficientemente e com maior flexibilidade. A avaliação da estratégia é realizada através de dois casos de estudo: o problema N-Rainhas e o problema de determinar a estrutura tridimensional de proteínas.

O restante do artigo está organizado da seguinte maneira: na Seção 2 os trabalhos relacionados serão apresentados. Na Seção 3 será feita uma descrição do sistema de gerenciamento de aplicações EasyGrid AMS. A proposta da estratégia de paralelização é explicada na Seção 4 e a avaliação é apresentada na Seção 5. A última seção resume as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

Branch-and-prune são algoritmos geralmente muito utilizados para se resolver problemas de satisfação de restrição (*Constraint Satisfaction Problem* - CSP) [Kumar 1992, Ruttkay 1998]. Aplicações *branch-and-prune* utilizam técnicas de busca exaustiva baseadas em ramificação (*branch*) e poda de ramos que determinam soluções 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.

Existe uma grande semelhança na construção algorítmica da técnica *branch-and-prune* (B&P) e *branch-and-bound* (B&B) [Hillam 2003]. 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 [Gendron and Crainic 1994]. 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.

Na literatura, existem diversos trabalhos que tratam da paralelização de algoritmos B&P [Anstreicher et al. 2002, Chrabakh and Wolski 2003] e, principalmente B&B. Classificações de algoritmos *branch-and-bound* paralelos são apresentadas em [Gendron and Crainic 1994, Trienekens and De Bruin 1992]. Já as implementações para grades computacionais, vem sendo estudadas em trabalhos mais recentes. As implementações mestre-trabalhador foram inicialmente abordadas. Na composição básica desta estratégia o objetivo é ter uma fila de tarefas centralizadas no mestre e distribuí-las entre os trabalhadores, aqueles que computam a tarefa. Conforme a execução dos trabalhadores, eles vão descobrindo novas tarefas para então enviá-las ao mestre. O mestre envia tarefas para trabalhadores ociosos. Em [Goux et al. 2001, Anstreicher et al. 2002, Chrabakh and Wolski 2003] trabalhos que utilizam a estratégia mestre-trabalhador para grades computacionais são propostos. No entanto, esta gerência centralizada de tarefas traz o problema de gargalo concentrado em um único ponto, o mestre. Recentemente, a abordagem tradicional mestre-trabalhador vem sendo substituída por propostas mais sofisticadas.

Em trabalhos como [Drummond et al. 2006] e [Mezmaz et al. 2007], soluções *workstealing* distribuídas para grades computacionais são desenvolvidas. A estratégia de *workstealing* é aquela onde trabalhadores ociosos retiram tarefas diretamente de outros trabalhadores. A comunicação entre trabalhadores para a troca de tarefas é feita de forma distribuída ou parcialmente distribuída. Em [Drummond et al. 2006], por exemplo, os trabalhadores são separados em grupos onde um deles é elegido líder. Os trabalhadores dentro dos grupos se comunicam entre si e a comunicação entre grupos é feita diretamente entre líderes. No entanto, vários procedimentos de gerência envolvendo um certo custo de comunicação devido a abordagem distribuída devem ser efetuados, como: balanceamento de carga, detecção de terminação e tolerância a falhas. Os grupos de trabalhadores devem trocar informações periodicamente para detectarem desbalanceamento de carga e eventuais falhas durante a execução.

Já em trabalhos como [Aida and Osumi 2005] e [Pezzi et al. 2007] existe uma outra estratégia alternativa para realizar-se o escalonamento dinâmico de tarefas. O

workstealing hierárquico trabalha com uma hierarquia de gerenciadores que realizam a gerência de tarefas entre os trabalhadores. Tal solução possibilita uma maior distribuição das tarefas realizando a gerência em pontos locais, tendo gerenciadores acima na hierarquia com visão mais ampla destes pontos. Por exemplo, o trabalho apresentado em [Pezzi et al. 2007] adota esta estratégia para aplicações MPI-2. A proposta é um *framework* para aplicações que seguem o modelo de programação de divisão e conquista, como é o caso dos algoritmos B&P e B&B. A estratégia trabalha com uma fila de tarefas que são distribuídas entre gerenciadores que estão organizados de forma hierárquica. A hierarquia de gerenciadores pode ser vista como uma árvore onde apenas processos associados aos gerenciadores folhas executam a computação do problema. Um gerenciador que detecta a ociosidade de um trabalhador pode requisitar, através de troca de mensagens, novas tarefas ao seu pai na hierarquia.

Existem três diferenças básicas entre a proposta deste artigo e a maioria dos trabalhos 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. Segundo, nesta proposta o escalonamento utilizado é proativo, ou seja, o escalonador reescalona as tarefas antes dos processadores ficarem ociosos. Nos outros trabalhos citados, o escalonamento é reativo - realizado quando é detectada a ociosidade de um processo trabalhador. 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 artigo, baseada no modelo alternativo de execução 1PTask [Sena et al. 2007], é criada uma quantidade de processos maior que o número de processadores disponíveis, para tirar proveito de ambientes heterogêneos, dinâmicos e compartilhados.

3. EasyGrid AMS

O *middleware* EasyGrid AMS [Nascimento et al. 2005, Sena et al. 2007] é um sistema gerenciador de aplicação que é integrado ao código do programa, tornando transparente a gerência da execução de aplicações MPI [Geist et al. 1996] em ambientes como grades computacionais. Este *middleware* é independente de qualquer outro sistema de *middleware grid*, necessitando apenas do Globus Toolkit [Globus Alliance] (GSI e GRAM) e da instalação padrão LAM/MPI 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 *middleware* é composto por três níveis de hierarquia de gerenciamento, um gerenciador global, gerenciadores de *sites* e de máquinas [Nascimento et al. 2005]. Cada processo gerenciador é baseado em uma arquitetura integrada contendo quatro camadas: gerenciamento de processos, monitoramento da aplicação, escalonamento dinâmico [Nascimento et al. 2007] e tolerância a falhas [Silva and Rebello 2007]. O escopo da funcionalidade de cada camada está associada ao nível hierárquico do processo gerenciador. Por exemplo, políticas de escalonamento dinâmico diferem de acordo com o nível hierárquico em que se encontram: dentro de um recurso, em um *site* ou entre *sites*.

O EasyGrid AMS implementa um modelo de execução de aplicações chamado

1PTask [Sena et al. 2007] que difere do convencional 1PProc - um processo por processador. Este modelo alternativo define tarefa como sendo uma unidade de trabalho computacional independente e de granularidade mais fina. 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 maior que o número de processadores. 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 - escalonamento dinâmico proativo. Além disso, 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 da aplicação previamente registrados [Silva and Rebello 2007].

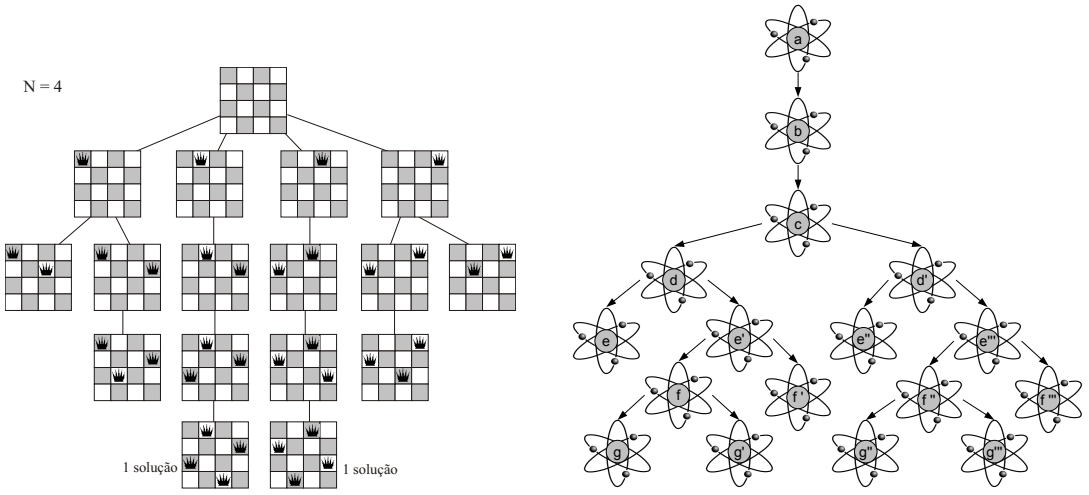
4. Paralelização de Aplicações *Branch-and-Prune*

Muitas aplicações tem a característica de executar algum tipo de busca para encontrar as soluções para o problema a ser resolvido. Geralmente, a execução da busca é a parte computacionalmente mais cara, já que é realizada através de combinações. As combinações podem ser vistas como uma árvore, na qual cada ramo representa uma escolha da combinação. O problema N-Rainhas e o problema de determinação da estrutura tridimensional de uma proteína são dois exemplos de aplicações que realizam busca em profundidade em uma árvore.

O N-Rainhas é um problema da área da computação. 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. A Figura 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.

O problema de encontrar a estrutura tridimensional de moléculas de proteínas é conhecido, em sua versão discreta, como PDGDM (Problema Discreto de Geometria das Distâncias em Moléculas), uma aplicação da área de bioquímica. Seu objetivo é encontrar possíveis estruturas tridimensionais da cadeia principal de moléculas de proteínas, considerando informações sobre os átomos disponibilizadas pela Ressonância Magnética Nuclear (RMN). A versão discreta do problema supõe que, dados quaisquer 4 átomos consecutivos a, b, c, d na cadeia principal de uma proteína, são conhecidas todas as distâncias inter-atômicas e o ângulo entre os vetores $\overrightarrow{a, b}$ e $\overrightarrow{b, c}$ não é múltiplo de π. Na Figura 1(b), cada nó representa um átomo da cadeia na ordem. As ramificações ocorrem por conta dos dois valores possíveis para o ângulo de torção [Lavor et al. 2009].

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



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

Figura 1. Exemplos de árvores de busca em profundidade.

busca construída dinamicamente, podendo ter ramos menores e maiores.

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 (linha 5).

4.1. 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. 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. Além dos parâmetros de entrada do Algoritmo 1, a versão paralela possui um novo parâmetro chamado $nivel\_corte$. Ele representa o nível da árvore no qual as tarefas devem ser criadas. Cada processo,

Algoritmo 1 buscaSolucao( ): Algoritmo *branch-and-prune* generalizado.

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 se**
5: **para cada** no_filho de no **faça**
6: buscaSolucao(no_filho, $nivel + 1$)
7: **fim para**

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

Entrada:
no: um nó da árvore.
$nivel$: nível da árvore.
$nivel\_corte$: nível em que novas tarefas serão criadas.

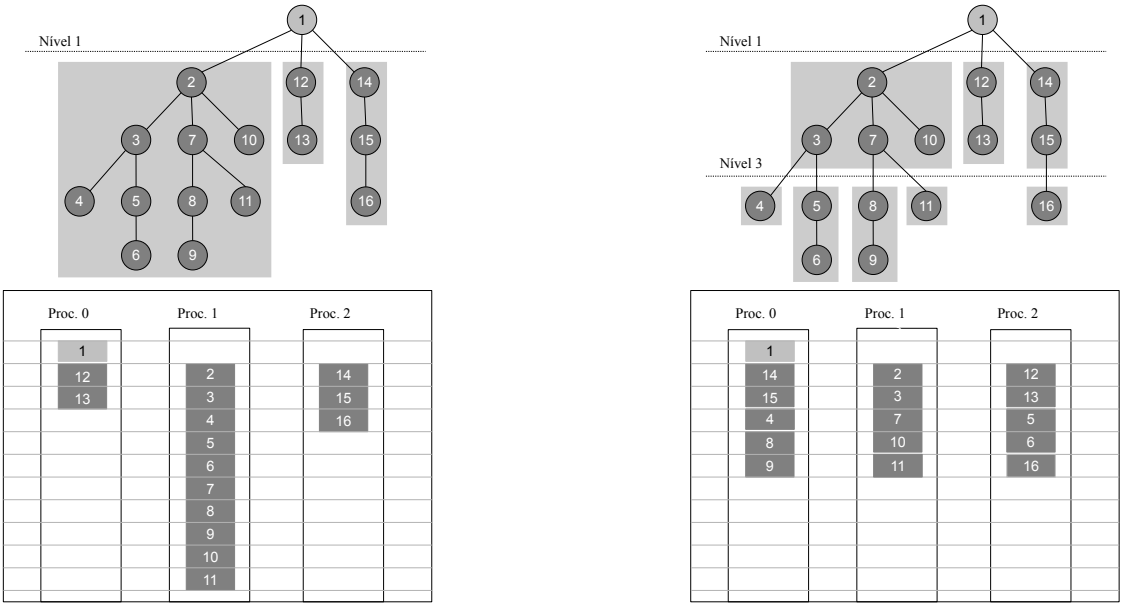
```
1:  calcula filhos de no
2:  se nivel é o último então
3:     retorna solução
4:  fim se
5:  se nivel = nivel_corte então
6:     modifica nivel_corte
7:     para cada no_filho de no faça
8:        criaNovaTarefa(no_filho, nivel + 1, nivel_corte)
9:     fim para
10: senão
11:    para cada no_filho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

inclusive filho, executa este algoritmo. O processo pai inicial introduz a construção da árvore de busca até alcançar o $nivel\_corte$ (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 $nivel\_corte$ é modificado (linha 6), para que os novos processos saibam em que nível criar seus processos filhos.

A Figura 2 mostra um exemplo de como a técnica de divisão proposta poderia melhorar o desempenho da aplicação. 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, considera-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 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. 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 corte, o número de tarefas aumenta enquanto a granularidade diminui. Na Figura 2(b), a árvore é separada no primeiro nível, gerando 3 tarefas. Cada uma destas tarefas receberão o nível de corte (igual a 3) e criarão suas respectivas tarefas. Sem alterar o número de processadores, pode-se conseguir um tempo de execução menor. A Figura 2(b) mostra, ainda, que é possível obter um escalonamento



(a) Um nível. **(b) Dois níveis.**

Figura 2. Exemplos de divisões em uma árvore de busca.

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 2(a).

Como a árvore de busca geralmente cresce exponencialmente, a quantidade de tarefas pode ser muito grande, dependendo dos níveis de corte usados. Além disso, quanto maior o primeiro nível de corte, menor a granularidade das tarefas criadas e maior o tamanho da tarefa pai inicial. A escolha dos valores de nível de corte depende da aplicação e seu objetivo é aumentar o grau de paralelismo com tarefas de granularidades semelhantes.

Através da criação dinâmica de processos, esta técnica é capaz de tirar proveito do escalonador dinâmico do EasyGrid AMS. Utilizando informações do sistema, como o poder computacional das máquinas, os escalonadores da hierarquia do *middleware* escolhem o melhor processador para escalonar as novas tarefas. A aplicação, assim, adquire uma maior autonomia e um melhor desempenho.

5. Avaliação

A 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. Porém, com os resultados preliminares neste *cluster*, já é possível analisar o comportamento do *middleware* EasyGrid AMS executando aplicações *branch-and-prune*.

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

foram realizados de modo exclusivo em 3 rodadas. A distribuição dos gerenciadores do EasyGrid (ver Seção 3) é simples: os gerenciadores global e *site* (apenas 1) em uma das máquinas, além de 20 gerenciadores de máquina em cada um dos 20 computadores restantes. Isto é, 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 e considerando rainhas que já foram posicionadas nas linhas anteriores. A implementação sequencial é baseada no algoritmo avaliado e disponibilizado em [Takaken 2003], atualmente considerado um dos mais eficientes. 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 [Lavor et al. 2009]. 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 1, onde ambas as árvores possuem 17 nós, mas a do N-Rainhas é mais larga, enquanto a do PDGDM é mais alta.

5.1. Resultados Preliminares - N-Rainhas

Os resultados obtidos para os experimentos com a aplicação N-Rainhas englobam as seguintes instâncias: 17 a 21 (valores de N). Para cada instância os valores 0 e os pares (0, 1) e (0, 2) representam os níveis de corte utilizados. O valor 0 é o corte no primeiro nível da árvore, e os pares (0, 1) e (0, 2) são cortes em dois níveis da árvore: o primeiro no nível 0 e o segundo nos níveis 1 e 2 respectivamente.

A Tabela 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. 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$>$. Como esperado, o número de processos aumenta consideravelmente, conforme aumenta-se o nível de corte. A medida que 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 3 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. Primeiramente, para valores altos de N, as execuções conseguem melhores *speed-ups*. A medida que as instâncias vão tendo mais trabalho, a paralelização passa a ter mais impacto na diminuição do tempo de execução em relação ao algoritmo sequencial. Para instâncias como $N = 17$, os *speed-ups* são menores pois as tarefas possuem granularidade muito fina (tempo de trabalho menor que 10 milisegundos, que é o tempo de criação dinâmica de um processo). Segundo, usando os pares de níveis (0,1) e especialmente (0,2), consegue-se obter *speed-ups* muito bons. Através destes cortes, o número de tarefas é maior e com granularidades de tamanho mais uniformes. Com o corte apenas no nível 0, o número de

Tabela 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

tarefas é bem menor e cada tarefa apresentam granularidade bastante diferentes entre si, devido ao desbalanceamento da árvore de busca. Por esta razão, o tempo de execução fica limitado pela maior tarefa e o aproveitamento dos recursos é consideravelmente menor. Com os dois níveis de corte, o escalonador consegue aproveitar mais os recursos.

O gráfico da Figura 4 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

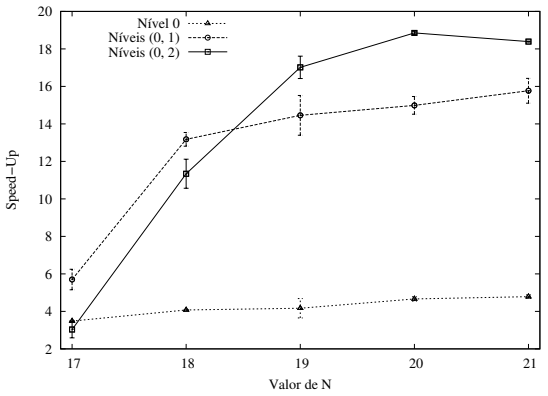


Figura 3. *Speed-up* obtido pela execução paralela das N-Rainhas.

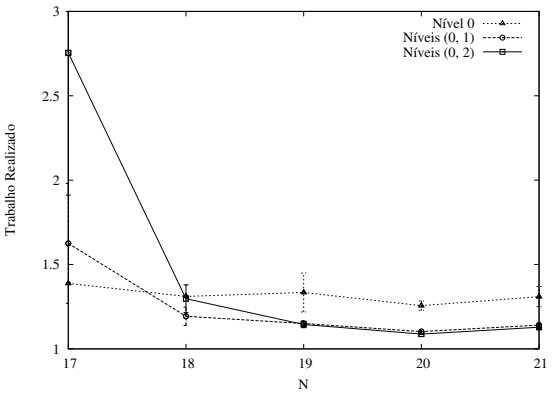


Figura 4. Trabalho realizado pela execução paralela das N-Rainhas.

executado pela versão paralela tende a ser maior. A medida que a granularidade das 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 utilizável e como consequência, o *speed-up* tende ao ideal.

5.2. Resultados Preliminares - PDGDM

Para os testes com a aplicação do PDGDM foram utilizadas 5 instâncias da base de dados PDB - *Protein Data Bank* [RCSB 2003], 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 pares de níveis de corte para a criação das tarefas. Logo, pode-se dizer que existem os níveis de corte único e de 2 cortes.

Tabela 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 2 mostra os resultados obtidos com as execuções sequencial e paralela variando o nível de corte. As 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 o tempo da execução sequencial em segundos que, como pode ser visto, não está diretamente associado ao número de átomos. A quarta coluna apresenta os níveis de corte utilizados, que são diferentes entre as instâncias. Estes valores foram escolhidos de acordo com o número de processos criados, para poder ilustrar melhor a execução paralela, e com a altura da árvore (dada pelo número de átomos). A quinta coluna contém os 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).

O gráfico da Figura 5 mostra a média dos *speed-ups* alcançados pelas execuções paralelas. 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 razoável. Os *speed-ups* relacionados aos níveis únicos de cada instância foram mais baixos e são explicados principalmente pela quantidade de trabalho não uniforme entre as tarefas. 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. Ainda, em todas as instâncias avaliadas do PDGDM existe o problema

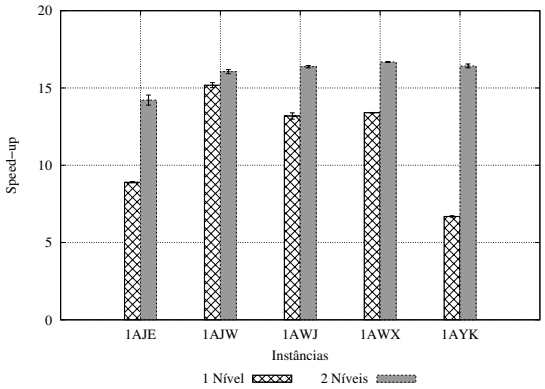


Figura 5. *Speed-up* obtido pela execução paralela do PDGDM.

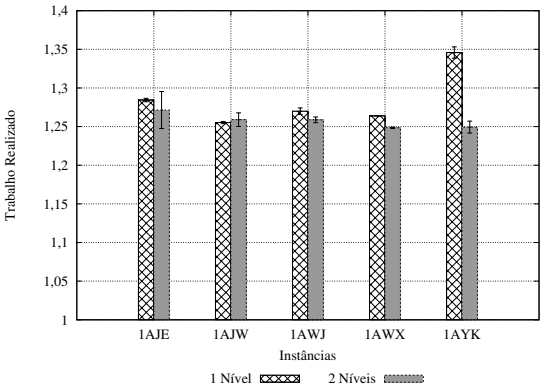


Figura 6. Trabalho realizado pela execução paralela do PDGDM.

de uma grande parte das tarefas terem granularidade muito fina. Elas levam cerca de 1 milisegundo para executarem e este tempo é 10 vezes menor que o tempo de criação de um processo. Isto faz com que boa parte do *overhead* de gerenciamento fique evidente. A Figura 6, que apresenta os valores de trabalho realizados, ajuda a deduzir tais *overheads*.

5.3. Resultados com Carga Externa

Esta subseção 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: um com 5 cargas estáticas e outro com 5 cargas dinâmicas. As 5 cargas estáticas ficam sempre executando em 5 máquinas fixas. Já as cargas dinâmicas ficam "saltando" de máquina em máquina no período de 10 segundos iniciadas sincronamente. O objetivo é observar o comportamento da execução da aplicação com cargas dinâmicas (uma forte característica de ambientes de larga escala) e comparar com a execução usando cargas estáticas.

Tabela 3. Comparação entre as execuções compartilhadas estática e dinâmica.

	Carga Estática		**Carga Dinâmica**	
Níveis	Tempo Parede	Int. Confiança	Tempo Parede	Int. Confiança
(0,1)	1378,12	127,57	1305,69	48,71
(0,2)	1086,92	11,86	1073,46	14,79

A Tabela 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, mostrando a habilidade da aplicação tratar efetivamente variações no poder computacional dos recursos.

6. Conclusão e Trabalhos Futuros

Este trabalho propôs uma estratégia de paralelização para algoritmos do tipo *branch-and-prune*. Esta estratégia visa aumentar o grau de paralelismo de aplicações de maneira simples e autônoma. A autonomia 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.

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 do linear (aproximadamente 19), mesmo executando sobre a camada de gerenciamento do *middleware*. Com a aplicação do PDGDM, os *speed-ups* foram bons, chegando ao valor de 16,7. Além disso, com os resultados em um ambiente distribuído compartilhado, pode-se perceber que o EasyGrid AMS lida bem com cargas dinâmicas.

Como trabalhos futuros, os *overheads* serão melhor investigados no intuito de minimizá-los. Além disso, novos testes serão analisados em um ambiente real de grades computacionais para instâncias maiores tanto do problema das N-Rainhas, quanto para o PDGDM, e serão feitas comparações com outras estratégias de paralelização como *workstealing*. Ainda, serão avaliadas 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. Eles poderiam, também, variar entre ramos conforme a demanda da execução. A próxima fase deste trabalho seria avaliar se o intervalo entre níveis deve ser fixo ou variável, se os níveis devem ser determinados para a árvore inteira ou para cada ramo independentemente. No entanto, cabe avaliar se tais escolhas dinâmicas realmente trazem benefícios a execução em relação a simples alternativa estática.

Referências

Aida, K. and Osumi, T. (2005). A case study in running a parallel branch and bound application on the grid. *Applications and the Internet, IEEE/IPSJ International Symposium on*, 0:164–173.

Anstreicher, K., Brixius, N., Goux, J.-P., and Linderoth, J. (2002). Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3).

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

Drummond, L., Uchoa, E., Gonçalves, A., Silva, J., Santos, M., and Castro, M. (2006). A grid-enabled distributed branch-and-bound algorithm with application on the steiner problem in graphs. *Parallel Comput.*, 32(9):629–642.

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

Gendron, B. and Crainic, T. (1994). Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, 42(6):1042–1066.

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

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

Hillam, B. (2003). *Algorithms : An Object Oriented Introduction*, chapter Backtracking and Branch-and-Bound.

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

Lavor, C., Liberti, L., Mucherino, A., and Maculan, N. (2009). On a discretizable subclass of instances of the molecular distance geometry problem. In Shin, S. Y. and Ossowski, S., editors, *SAC*, pages 804–805. ACM.

Mezmaz, M., Melab, N., and Talbi, E.-G. (2007). A grid-based parallel approach of the multi-objective branch and bound. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:23–30.

Murch, R. (2004). *Autonomic Computing*. IBM Press.

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

Nascimento, A. P., Sena, A. C., da Silva, J., Vianna, D., Boeres, C., and Rebello, V. (2005). Managing the execution of large scale MPI applications on computational grids. In *Computer Architecture and High Performance Computing. SBAC-PAD'05. 17th International Symposium on*, pages 69 – 76.

Pezzi, G., Cera, M., Mathias, E., and Maillard, N. (2007). On-line scheduling of MPI-2 programs with hierarchical work stealing. In *Computer Architecture and High Performance Computing. SBAC-PAD'07. 19th International Symposium on*, pages 247 –254.

RCSB (2003). Research collaboratory for structural bioinformatics `http://www.rcsb.org`. Acessado em Dezembro de 2009.

Ruttkay, Z. (1998). Constraint satisfaction - a survey. *CWI Quarterly*, 11:123–161.

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

Silva, J. and Rebello, V. (2007). Low Cost Self-healing in MPI Applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting*, pages 144–152. Springer.

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

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

Trienekens, H. and De Bruin, A. (1992). Towards a taxonomy of parallel branch and bound algorithms. *Report EUR-CS-92-01. Rotterdam: Erasmus Univ.*