Denis Alessandro Altoé Falqueto

Implementação baseada em Threads de uma heurística GRASP para o problema da Árvore Geradora de Custo Mínimo com Grupamentos

Monografia de Pós-graduação "*Lato Sensu*" apresentada ao Departamento de Ciência da Computação para obtenção do título de Especialista em "Administração em Redes Linux"

Orientador
Profª. Marluce Rodrigues Pereira

Lavras
Minas Gerais - Brasil
2009

Denis Alessandro Altoé Falqueto

Implementação baseada em Threads de uma heurística GRASP para o problema da Árvore Geradora de Custo Mínimo com Grupamentos

Monografia de Pós-graduação "*Lato Sensu*" apresentada ao Departamento de Ciência da Computação para obtenção do título de Especialista em "Administração em Redes Linux"

Aprovada em *21 de novembro de 20096*

Prof. Joaquim Quinteiro Uchôa

Prof. Samuel Pereira Dias

Prof[a]. Marluce Rodrigues Pereira
(Orientador)

Lavras
Minas Gerais - Brasil
2009

Sumário

Lista de Tabelas

Lista de Figuras

Resumo

Este trabalho descreve a implementação de uma heurística GRASP para o cálculo da árvore geradora de custo mínimo, usando *threads* de execução. Utilizou-se a linguagem Java e a *framework* de execução de tarefas para simular dois cenários: threads que executam um número definido de iterações e; uma tarefa para cada iteração da heurística. Isto permite observar as diferenças entre threads de longa duração e tarefas curtas e independentes.

Palavras-Chave: 1. GRASP. 2. Threads. 3. Java. 4. Árvore Geradora de Custo Mínimo com Grupamentos. 5. AGMG

Capítulo 1

Introdução

1.1 Motivação

Na Ciência da Computação, a categoria de problemas NP-Completos é uma constante motivação para o desenvolvimento de técnicas e algoritmos capazes de os resolver. Porém, dada sua característica inerente de crescimento exponencial do espaço de soluções, a busca de algoritmos exatos não é sempre viável. Por este motivo, foram surgindo técnicas aproximadas que geravam resultados bons o suficiente para serem considerados. A esta abordagem dá-se o nome de *Heurística*.

Com o tempo, foi se percebendo que várias heurísticas possuíam características comuns e por isso foram agrupadas em famílias. Uma destas famílias é a *Greedy Random Adaptive Search Procedures*, ou GRASP, que significa Procedimentos Gulosos Aleatórios Adaptativos. Este tipo de heurística consiste em dois passos: (a) a construção de uma solução inicial de forma gulosa, escolhendo a próxima parte baseada apenas no custo que esta acrescenta ao total; (b) a melhoria da solução inicial com a alteração de alguns elementos por seus "vizinhos", na tentativa de melhorar o valor total. Repetem-se estes dois passos inúmeras vezes, de forma a tentar explorar ao máximo o espaço de resultados.

Esta heurística é bastante eficiente, porém ela tem a tendência a produzir ótimos locais, ou seja, soluções boas numa determinada região do espaço, mas não necessariamente boas no geral. Assim, quanto maior o número iterações, maior será a probabilidade da solução ser um ótimo global. A paralelização deste algoritmo teria a vantagem de explorar melhor o espaço de soluções, dado o mesmo

tempo de processamento, ou então explorar o mesmo número de possibilidades em menos tempo.

O passo (a) da heurística indica que a solução inicial será encontrada aleatoriamente. A partir desta solução, uma nova solução será criada. Estes dois passos serão repetidos um determinado número de vezes. Desta forma, cada iteração é independente das demais. Por isto, o potencial para paralelização é muito grande, pois uma iteração pode ser executada simultaneamente a outras.

Uma abordagem para a paralelização desta heurística é o uso de *clusters*. Um cluster é um conjunto de computadores conectados que interagem para a execução de uma operação distribuída entre eles. Normalmente, há um computador principal que coordena os demais, enviando as requisições, recebendo as respostas e montando a solução final.

Uma outra abordagem é o uso de *threads de execução*, a qual permite que uma estrutura similar seja implementada em um único computador. Uma *thread* principal inicia e coordena a execução de outras *threads*, enviando os pedidos, recebendo as respostas e montando a solução final, assim como no cluster. Porém, no caso de *threads*, a comunicação é muito mais eficiente, pois fazem parte do mesmo processo e executam no mesmo computador, tendo apenas que ocorrer a troca de contexto entre elas. Porém, um número excessivo de *threads* pode fazer com que ocorra muitas trocas de contexto, em detrimento do tempo de execução útil das tarefas.

A indústria de processadores para computadores pessoais está sofrendo uma mudança significativa. Até um tempo atrás, a velocidade dos processadores aumentava exponencialmente. De alguns mega Hertz, passou-se às dezenas, depois às centenas, e finalmente aos giga Hertz. Porém, a tecnologia utilizada está começando a chegar a seu limite físico de miniaturização e o aumento constante de processamento parece estar chegando ao fim.

Assim, a indústria começou a popularizar uma tecnologia conhecida como *multicore*. Ela integra mais de um processador no mesmo *chip*, permitindo que mais de uma operação seja executada simultaneamente. Hoje em dia é muito comum a aquisição de computadores para uso pessoal com pelo menos 2 núcleos. Com isto, a tendência do aumento constante de performance pode ser mantida, porém, com o aumento da complexidade de programação. A programação concorrente agora não é somente um capricho, mas sim uma necessidade da qual não se pode mais fugir. No entanto, ela possui alguns problemas específicos que não existem na programação sequencial.

Neste cenário, o presente trabalho busca fornecer uma implementação baseada em *threads* para o problema da Árvore Geradora de Custo Mínimo com Grupamentos, através de uma heurística GRASP. Inicialmente, a implementação se dará de forma a simular um cluster usando *threads*, ou seja, uma *thread* principal coletará os parâmetros, montará uma estrutura global para representar o grafo e disparará um certo número de *threads* que executarão um número predefinido de iterações da heurística GRASP. Ao fim das iterações, cada *thread* retornará uma solução para a *thread* principal, sendo esta responsável por selecionar dentre as recebidas e gerar o arquivo de saída com a melhor solução.

1.2 Organização do texto

O capítulo 2 detalha o problema da Árvore Geradora de Custo Mínimo com Grupamentos e sua solução através de uma heurística GRASP.

O capítulo 3 trata das *threads* de execução e suas consequências para a programação de aplicações, de forma geral. Também aborda o modelo de memória e os recursos para a programação com múltiplas *threads* na linguagem de programação Java, escolhida para a implementação neste trabalho.

O capítulo 4 discute a metodologia adotada para a solução implementada nesta monografia. Nele é mostrada a estrutura geral da solução, com ênfase na separação do papéis e na cooperação das partes entre si.

O capítulo 5 expõe os resultados obtidos com a implementação. O capítulo 6 encerra o trabalho com a opinião do autor e sugestões para extensões e melhorias.

Capítulo 2

O problema da Árvore Geradora de Custo Mínimo com Grupamentos

2.1 Definição do problema

Seja um grafo, não orientado, $G = (V, E)$, onde V é um conjunto de vértices e E é um conjunto de arestas ligando estes vértices, e que estas possuem um custo associado não negativo. Uma árvore geradora de custo mínimo é um subgrafo $A = (V, E_A)$, de forma que $E_A \subseteq E$ e existe um caminho que liga todo par de vértices de V composto por arestas pertencentes a E_A e que a soma dos custos das arestas de E_A é o menor possível (MYUNG; LEE; TCHA, 1995).

Se o conjunto de vértices V for subdividido em partes disjuntas, chamadas grupamentos, $(V_1, V_2, \ldots, V_k)$, e $(V_1 \bigcup V_2 \bigcup \ldots \bigcup V_k) = V$, podemos alterar a definição anterior de árvore geradora para que ela ligue quaisquer dois grupamentos. Ou seja, para dois vértices v_i e v_j quaisquer, $i \neq j$, deve existir pelo menos um caminho que liga v_i a v_j e que todas suas arestas pertençam a E_A, de forma que a soma dos custos das arestas de E_A seja mínimo. A esta árvore dá-se o nome de Árvore Geradora de Custo Mínimo com Grupamentos.

Este problema é NP-Completo, pois à medida que o grafo aumenta de tamanho, seja no número de arestas ou no número de vértices, o conjunto de soluções a serem exploradas numa abordagem exaustiva cresce exponencialmente. Daí, surge

a necessidade de encontrar-se uma heurística que permita encontrar-se uma solução boa o suficiente, sem que todas as soluções possíveis sejam avaliadas.

Uma árvore geradora de custo mínimo com grupamentos possui muitas aplicações, dentre as quais podemos destacar:

- Em telecomunicações, onde redes regionais precisam se conectar a uma árvore que contenha uma conexão para cada sub-rede. Neste caso, um vértice de cada sub-rede deverá funcionar como o *gateway* (MYUNG; LEE; TCHA, 1995).

- Na logística, em que várias regiões precisam estar conectadas umas às outras por meio de rodovias ou links de comunicação. Por exemplo, uma rede de lojas precisa decidir os locais dos centros de distribuição para minimizar os custos com transporte (SHYU *et al.*, 2006).

- A irrigação de áreas rurais ou desérticas pode ser modelada através de uma árvore geradora de custo mínimo com grupamentos, onde cada grupamento representa uma determinada parte do terreno a ser irrigado, os vértices são os pontos de concentração de água e as arestas são os custos de envio da água entre os pontos (ALVARENGA; ROCHA; PEREIRA, 2006). Um exemplo pode ser observado na figura 2.1, em que está modelada uma área irrigável, com uma fonte d'água e uma possível solução de canalização.

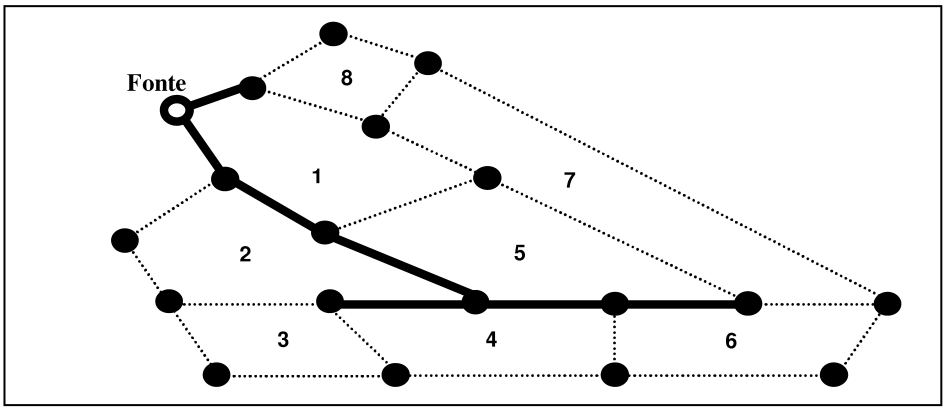

Figura 2.1: Exemplo de grafo, com uma possível AGMG em negrito

2.2 A heurística GRASP

A família de heurísticas GRASP (Greedy Randomized Adaptative Search Procedure - Procedimento guloso e aleatório de busca adaptativa, em tradução livre) é um grupo de procedimentos baseados em dois passos principais: montar uma solução inicial de forma gulosa, ou seja, levando em consideração apenas o custo do próximo elemento a ser incluído na solução; e tentar substituir alguns elementos da solução para tentar encontrar uma solução melhorada (FEO; REZENDE, 1989). O trabalho de (ALVARENGA; ROCHA; PEREIRA, 2006) apresenta a estrutura básica do algoritmo das heurísticas GRASP, aqui mostrado na figura 2.2.

$f(s) \leftarrow \infty$;
para $k \leftarrow 1$ **até** $Maxiter$ **faça**
    $x \leftarrow GulosoAleatorizado()$;
    $x \leftarrow BuscaLocal(x)$;
    se $f(x) < f(s)$ **então**
        $s \leftarrow x$;
    fim se
fim para
retorna s

Figura 2.2: Algoritmo básico das heurísticas GRASP

Na primeira fase, a chamada fase de construção, é montada uma solução inicial iterativamente, elemento a elemento. Inicialmente, monta-se uma lista dos melhores elementos do problema, chamando-se este conjunto de LRC (Lista Restrita de Candidatos). O tamanho desta lista é controlado por um parâmetro chamado α. Este parâmetro varia de 0 a 1, sendo que 0 implicará na lista de tamanho zero e o algoritmo se comportará de forma puramente gulosa; se α for 1, a lista de candidatos será todas as arestas do problema e a execução será guiada apenas pela aleatoriedade.

Com a lista restrita de candidatos montada, o próximo passo é selecionar um dos elementos desta lista como possível próximo elemento da solução. Após isto, ele será validado diante dos elementos atuais para que a solução faça sentido, de acordo com a definição do problema que se esteja tentando resolver. Repete-se estes passos até que a solução satisfaça o problema.

A fase de busca local consiste em pesquisar na vizinhança da solução inicial para tentar descobrir uma outra que seja melhor. Esta busca é feita modificando-se a solução inicial em um de seus elementos para ver se o resultado é melhor. Se for,

esta mudança é incorporada à solução atual. Este processo se repete até que não haja mais nenhuma mudança possível a ser explorada.

Capítulo 3

Threads de Execução

Hardware e Software formam a mais importante dualidade na Informática. Do diálogo entre estes dois campos, se faz o progresso; os avanços em um deles estimulam o outro a estender os limites ainda mais. Por exemplo, a criação dos computadores com programas armazenados na mesma área dos dados foi um passo importante para a criação e evolução dos sistemas operacionais. Com o surgimento destes últimos, o conceito de programa começou a tomar forma como algo que poderia ser gerenciado por um outro programa (o sistema operacional). Desta forma, caminhou-se dos sistemas de processamento em lote para os sistemas de compartilhamento de tempo, que alternavam a execução de vários programas no processador, dando a impressão de que o computador estava á disposição do usuário que o estava operando naquele momento.

O compartilhamento de tempo criou a necessidade do escalonador de processos. Esta parte do sistema operacional é a responsável pela orquestração dos vários programas que competem pela CPU. Esta contingência levou à criação de computadores com a capacidade de executar mais de um programa simultaneamente com a adição de mais de um processador. A disponibilidade de mais recursos de processamento, além do fato de algumas operações serem muito dependentes de entrada ou saída de dados, deixando a CPU ociosa, conduziu a Ciência da Computação a desenvolver os conceitos de multiprogramação.

3.1 Multiprogramação

A programação paralela, ou multiprogramação, é a modalidade de construção e execução de um programa em que mais de uma instrução pode ocorrer simultaneamente ou atuar sobre múltiplas fontes de dados ou um misto dessas duas formas (FLYNN, 1972). A multiprogramação pode ser distribuída ou local, sendo que esta última divide-se em multiprogramação por processos ou por *threads*. A multiprogramação por processos é aquela em que cada parte da implementação não tem acesso a informações particulares das outras partes, como por exemplo, memória e descritores de arquivos. Toda comunicação é feita através de mecanismos oferecidos pelo sistema operacional subjacente.

A multiprogramação por *threads* é aquela em que um programa é dividido em partes que tem acesso total a todos os recursos do processo; uma *thread* pode, inclusive, acessar dados de outras *threads*, pois elas compartilham todos os recursos que o sistema operacional disponibiliza para o processo como um todo. Esta característica é a principal vantagem e, ao mesmo tempo, a maior armadilha das *threads*.

Vejamos um exemplo clássico: atualizar o valor de uma variável com seu valor atual mais um. Digamos que duas *threads A* e *B* compartilhem uma variável *n* e que, executem a seguinte instrução:

```
n = n + 1
```

Esta linha convertida para linguagem de máquina será algo do tipo:

- Busque o valor de *n* e coloque no registrador *r*
- Adicione 1 ao valor do registrador *r*
- Envie o valor de *r* para o endereço de *a*

Como estas instruções serão executadas em duas *threads* diferentes, é possível que o mesmo valor de *n* seja buscado duas vezes, um por cada *thread*, e o valor final será n + 1 ao invés de *n + 2*.

Este problema é chamado *condição de corrida* e a solução convencional é o uso de sincronização para garantir que os dados compartilhados sejam sempre atualizados de forma coerente. A sincronização pode ocorrer por vários mecanismos e normalmente depende dos recursos do sistema operacional e da linguagem

de programação utilizada. O mecanismo mais comum para implementar a sincronização entre *threads* é o bloqueio ou trava (*lock*). Ela garante que apenas uma *thread* poderá executar um trecho do código simultaneamente. Este trecho deve ser explicitamente demarcado pelo programador e é denominado *seção crítica*.

3.2 Vantagens da multiprogramação

Certos problemas são muito mais bem modelados quando feitos através de múltiplas *threads*. Quando uma parte da solução é independente das demais ou então quando a solução é composta pela repetição de um algoritmo e cada iteração é independente das anteriores, o ganho com a paralelização é visível. Mesmo quando um problema não se encaixa perfeitamente neste tipo, ainda é possível utilizar-se de paralelização para isolar seus componentes e tornar a solução mais escalável.

Outra situação que ganha muito com o uso de *threads* é o tratamento de processamento que depende mais de entrada e saída do que do processador propriamente dito. Por exemplo, um servidor *web* é claramente paralelizável, pois cada requisição HTTP é sempre isolada das demais, mesmo que ela pertença a uma sessão, e é extremamente orientada a entrada e saída, seja através da simples disponibilização de arquivos estáticos ou pela montagem dinâmica através de algum tipo de banco de dados. Neste caso, enquanto uma requisição está em espera pela leitura do sistema de arquivos ou do processamento de uma consulta no banco de dados, outras requisições podem ser atendidas pelo processador.

De forma muito similar, as aplicações que utilizam uma interface gráfica com o usuário também ganham com o uso de *threads*. Na sua imensa maioria, estas aplicações são estruturadas em torno de um processador de eventos de usuário ou do sistema operacional. Este processador de eventos reage aos pedidos recebidos e deve responder da forma mais rápida possível, sem que ele mesmo perca tempo demais em uma requisição. Se uma requisição possuir a menor chance de ser demorada, ela deve ser colocada numa *thread* independente para que as demais requisições não sejam postergadas.

Além de tudo isto, com a disponibilidade cada vez maior de processadores multicore ou multiprocessados, o uso de algoritmos paralelos pode trazer muitas vantagens para todos estes cenários, permitindo que a solução seja mais escalável.

3.3 Desvantagens da multiprogramação

As seções críticas devem ser usadas com cautela, pois elas introduzem uma contingência que pode afetar a performance em caso de haver muitas *threads* tentando entrar na mesma seção. Por este motivo, elas devem envolver o menor trecho possível que garanta a corretude da implementação.

Um outro risco do uso de sincronização por bloqueio é o que se denomina de *deadlock*, ou seja, uma *thread A* aguarda num bloqueio que está atualmente em posse da *thread B* e, por sua vez, a *thread B* aguarda num bloqueio que está em posse da *thread A*. A partir deste ponto, o processo como um todo fica estagnado.

Outro risco do uso indevido de sincronização é a *inanição*, que ocorre quando o escalonamento de *threads* a serem executadas leva a alguma delas a sempre ser deferida, fazendo com que esta *thread* nunca termine.

Além disso, o uso excessivo de *threads* pode acarretar numa sobrecarga para o escalonador de processos, que passará a tomar muito tempo do processador com a tarefa de decidir qual a próxima *thread* a ser executada. Isto pode até mesmo deixar o sistema em questão totalmente paralisado, com o processador totalmente ocupado e sem nenhum trabalho prático sendo realmente executado.

3.4 A Lei de Amdhal

Como mencionado anteriormente, a indústria está cada vez mais voltando-se para a criação de processadores com mais de um núcleo de processamento. Esta tendência sugere que os programas deverão ser escritos de forma a tirar proveito da capacidade extra de processamento. A princípio, pode-se imaginar que o ganho obtido com a exploração do paralelismo será simplesmente proporcional ao número de tarefas que estão sendo executadas simultaneamente. Por exemplo, se um processador possui 4 núcleos e uma tarefa for dividida em 4 *threads*, então espera-se que o tempo total de processamento será um quarto do tempo gasto por um processador simples. Na prática, porém, isto dificilmente ocorre.

A questão do ganho de tempo é mais complexa e envolve a noção de que, provavelmente, uma certa parte do processo é inerentemente sequencial. Seja S o ganho de tempo em relação a um processamento puramente sequencial. Ele será, então, a razão entre a quantidade de tempo que o processo puramente sequencial executa, sobre a quantidade de tempo que o processamento ocorre em paralelo. A

Lei de Amdhal (HERLIHY; SHAVIT, 2008, pág. 13) descreve o ganho máximo que pode ser obtido por n processos colaborando numa aplicação, sendo que p é a fração do tempo em que o processamento é executado em paralelo. Assume-se que o tempo total é igual a 1, para que o resultado seja normalizado. Portanto, a parte paralelizada é igual a p/n e a parte sequencial é igual a $1-p$. Desta forma, o tempo gasto pelos n processos será:

$$1-p+\frac{p}{n}$$

A Lei de Amdhal, que representa a razão entre o tempo gasto num processamento puramente sequencial sobre o processamento paralelo, é definida por:

$$S=\frac{1}{1-p+\frac{p}{n}} \tag{3.1}$$

Para exemplificar a aplicação da fórmula 3.1, pode-se supor uma situação em que um determinado processo possa ser dividido em 10 partes, porém, uma (e apenas uma) destas partes dura o dobro das demais. Portanto, n será igual a 10, p será igual a $10/11$ e $1-p$ será igual a $1/11$. Assim, a aplicação da fórmula nos leva a

$$S=\frac{1}{1-10/11+\frac{\frac{10}{11}}{10}}=\frac{1}{\frac{1}{11}+\frac{1}{11}}=\frac{1}{\frac{2}{11}}=\frac{11}{2}=5,5$$

Ou seja, o ganho máximo que se pode esperar é de uma execução 5,5 vezes mais rápida do que uma puramente sequencial, ao invés de 10 vezes, como alguém poderia supor. Isto porque apenas uma das tarefas era o dobro das demais. Neste caso, uma parte do processamento foi executada sequencialmente por um de dois motivos: esta parte é inerentemente sequencial e não tem como ser paralelizada; ou o programador não levou em consideração a possibilidade de um dos processos perdurar por mais tempo do que as demais. Este último caso é o que se deve evitar.

3.5 *Threads* em Java

A linguagem e o ambiente de execução Java tiveram, desde seu surgimento, um suporte muito bom ao uso de *threads*. Inicialmente, apenas existiam as classes *Thread* e o bloqueio chamado *Monitor*. Porém, na versão 5, Java recebeu recursos muito avançados para a multiprogramação.

3.5.1 O modelo de memória

A arquitetura moderna dos processadores baseia-se em vários pontos de paralelismo e cache para fornecer o máximo de desempenho, mesmo em processadores com apenas um núcleo. Instruções executadas numa ordem diferente da especificada no programa, agrupamento de escritas em memória, *caches* locais por núcleo do processador, variáveis em registradores ao invés de memória são otimizações importantes implementadas por vários fabricantes há muitos anos. Estas táticas, porém, são um novo ponto de falha quando se está utilizando mais de uma *thread* de execução. Por exemplo, se uma variável é atualizada, mas o valor novo permanece apenas no cache de um dos núcleos do processador, uma outra *thread* executando em outro núcleo não teria acesso a este novo valor. Para garantir que todos os valores publicados por uma *thread* sejam acessíveis às outras, utiliza-se a sincronização. É interessante observar o seguinte comentário de (GOETZ *et al.*, 2006, pág. 338), aqui traduzido livremente:

> Um modelo mental conveniente para a execução de um programa é imaginar que exista uma ordem única na qual as operações ocorrem, independente do processador em que serão executadas e que cada leitura de uma variável verá a última escrita nesta variável em qualquer processador. Este modelo feliz, mesmo que pouco realista, é chamado de *consistência sequencial*. Os programadores frequentemente assumem esta consistência mas nenhum processador moderno a oferece, assim como a máquina virtual Java. O modelo computacional sequencial clássico, chamado de Modelo de von Neuman, é apenas uma vaga aproximação de como os multiprocessadores modernos se comportam.

Esta ideia é bastante importante para que se perceba que o uso de sincronização não é apenas necessário por causa da ordem de intercalação de duas ou mais *threads*. O modelo de memória do Java fornece as condições nas quais é garantido que o conteúdo da memória estará disponível da forma que o programador supôs numa execução sequencial daquele algoritmo, assim como deixa livre para que o processador e a máquina virtual otimizem todos os demais casos da melhor forma possível. É o melhor equilíbrio entre segurança e eficiência. Para isto, o modelo de memória se baseia no conceito da relação *ocorre-antes*. Toda vez que uma operação garante esta relação, pode-se ter certeza que as alterações na memória estarão visíveis para as demais *threads* e diz-se que uma *thread publicou* um valor. Al-

gumas operações que garantem a publicação dos dados são (GOETZ *et al.*, 2006, pág. 341):

- Instruções dentro da mesma *thread* sempre *ocorrem-antes* das operações que as seguem, dentro da mesma *thread*.
- O desbloqueio de uma trava sempre *ocorre-antes* de um novo bloqueio na mesma trava.
- A escrita numa variável do tipo *volatile* sempre *ocorre-antes* das leituras subsequentes a esta mesma variável
- O método Thread.start() sempre *ocorre-antes* de qualquer instrução desta *thread*.
- A relação *ocorre-antes* é transitiva. Se *A ocorre-antes* de *B* e *B ocorre-antes* de *C*, então *A ocorre-antes* de *C*. Porém, a relação *ocorre-antes* não existe para todo e qualquer par de instruções do programa.

Existem outras condições em que ocorre a relação *ocorre-antes*, porém, estas são as mais importantes e comuns. Ao utilizá-las é possível garantir que um programa em Java não acarretará em resultados inesperados causados por reordenação de código ou *cacheamento* de memória. Elas não garantem, porém, que não haverão problemas de *deadlocks*, *livelocks* ou inanição de alguma *thread*.

3.5.2 Recursos básicos

Para se criar uma *thread* em Java, basta se definir uma classe que estenda a classe *Thread* ou que entenda a interface *Runnable* e, posteriormente, executá-la através de um novo objeto da classe *Thread*. Esta classe é bastante antiga, datando do kit de desenvolvimento Java 1.0 e mostra os sinais de sua longa trajetória de alterações através da quantidade de métodos depreciados, principalmente os relativos à parada e suspensão forçada da execução de uma *thread*.

A forma de iniciar a execução de uma *thread* é chamar seu método *start()*. A partir deste ponto, sua execução será independente da *thread* que a criou. O método *run()* (também na interface *Runnable*) é usado para implementar a funcionalidade da *thread*. O método *join()* pode ser usado para que uma *thread* aguarde o término da execução de outra, de forma que a *thread* que originou a chamada ficará bloqueada até que a outra *thread* conclua sua execução.

O uso de sincronização é implementado através da cláusula *synchronized*, que pode ser associada a todo um procedimento ou apenas a um trecho de código. Em ambos, é garantido que apenas uma *thread* poderá executar o trecho de código protegido, através do uso de travas (*locks*). Todos as instâncias de objetos em Java possuem uma trava implícita, criada e gerenciada pela própria máquina virtual Java. Quando o método todo está sincronizado, a execução deste método exigirá que a trava do próprio objeto a que ele pertence seja adquirida pela *thread* que está executando no momento. Se esta trava já estiver de posse de outra *thread*, então a execução será bloqueada até que esta libere a trava. A cláusula *synchronized* também pode ser usada no interior de um método para proteger apenas um trecho de código. Neste caso é possível que outro objeto seja usado como gerenciador da trava, sendo o padrão que o próprio objeto seja usado na sincronização.

3.5.3 Recursos avançados

A tendência natural para o uso de *threads* é a execução de tarefas. A princípio, uma tarefa é uma entidade que possui um início, um meio e um fim e não tem uma grande dependência de outras tarefas ou partes do sistema. Porém, a vinculação direta entre *threads* e tarefas pode levar a algumas inconveniências, segundo (GOETZ *et al.*, 2006, pág. 116):

- a criação e o gerenciamento de *threads* tem um custo, mesmo que pequeno, que pode representar uma parcela considerável do tempo de processamento das requisições;
- *threads* consomem memória, mesmo as ociosas. Por isto, a criação excessiva de *threads* pode comprometer o funcionamento e a escalabilidade do sistema.
- há um número máximo de threds que podem ser criadas. Isto depende do sistema operacional, porém, se a criação de *threads* for indiscriminada, ela pode levar a uma exaustão deste limite.

Por estas razões, é interessante separar o conceito de tarefas e *threads*. Uma tarefa é uma entidade que representa uma atividade que precisa ser executada; uma *thread*, por outro lado, é uma das muitas formas de se estruturar a execução de tarefas.

Para simplificar a modelagem de problemas de multiprogramação, a biblioteca padrão da linguagem Java, a partir de sua versão 5, inclui um conjunto de

classes que facilitam a manipulação e a estruturação de tarefas. Este conjunto de classes são conhecidas como *framework Executor*. O ponto central desta *framework* é a interface *Executor*, que possui apenas um método *execute(Runnable task)*, que recebe uma implementação da interface *Runnable*. Também acompanham algumas implementações de executadores que permitem desacoplar a criação de tarefas da forma de execução destas. Por exemplo, há um executador que possui apenas uma *thread* e executa as tarefas em sequência. Há outro que possui um conjunto de *threads* e uma lista de tarefas; sempre que uma das *threads* do grupo tornar-se ociosa, a próxima tarefa será retirada da fila e passada à *thread* livre para ser executada.

A principal vantagem desta *framework* é a separação entre a criação das tarefas e a execução destas, permitindo que a aplicação seja facilmente escalável sem que o programa em si sofra uma grande modificação. Com o uso de grupos de *threads*, é possível que um sistema suporte uma alta carga de requisições sem que o sistema operacional fique sobrecarregado. Além disto, existem alguns executadores avançados que permitem que as tarefas retornem resultados e que a sincronização entre a parte que gera o resultado e a que o consome seja feita corretamente.

Capítulo 4

Metodologia

A meta-heurística GRASP, como foi vista na seção 2.2, é bastante apropriada para a paralelização, pois cada iteração é completamente independente das demais. Assim, (ALVARENGA; ROCHA; PEREIRA, 2006) tratou sobre a implementação desta meta-heurística num ambiente clusterizado, onde uma máquina trabalha como o mestre e as demais recebem as requisições. Quando estas são concluídas, o resultado é enviado de volta á máquina mestre para que, ao término de todos os nós, a solução final seja encontrada. Cada nó receberá a incumbência de executar um número fixo de vezes para minimizar a comunicação entre o mestre e os demais componentes.

Segundo a lei de Amdhal, vista na seção 3.4, é desejável que o processamento seja distribuído de forma mais homogênea possível, para que nenhum dos nós fique ocioso antes do término dos demais. Porém, este objetivo fica comprometido na solução em cluster citada, pois não há comunicação entre os nós, exceto no início e no término da execução de cada um. Assim, se um dos nós terminar, ele não terá mais nada a fazer até que uma nova sessão de execução seja iniciada pelo mestre.

Entretanto, a heurística GRASP aplicada ao problema da árvore geradora de custo mínimo leva naturalmente à criação deste tipo de situação, já que a cada iteração, uma solução inicial é obtida aleatoriamente. A aleatoriedade na montagem leva a algumas interações gerarem soluções inviáveis, portanto, encurtando o tempo total de execução. Assim, é bastante possível que um nó não encontre nenhuma ou quase nenhuma solução inicial e outro encontre várias (possivelmente idênticas), fazendo com que o primeiro seja muito breve e o segundo mais demorado.

Por esta razão, este trabalho fará dois experimentos: o primeiro é recriar a mesma estrutura de cluster, porém, utilizando um número fixo de *threads* que executará um número fixo de iterações em um único computador (figura 4.1); o segundo é criar *n* tarefas, com *n* igual ao número total de iterações e utilizar a *framework* de execução de tarefas da biblioteca padrão do Java para que um conjunto de *threads* estejam sempre ocupadas (figura 4.2).

$f(solucao) \leftarrow \infty$;
$nIterPorThread \leftarrow nIter/nThreads$;
para $thread \leftarrow 1$ **até** $nThreads$ **faça em paralelo**
    executaGRASP($nIterPorThread$);
fim para
enquanto *houver respostas na fila* **faça**
    $solucao' \leftarrow$ obtemSolucao();
    se $f(solucao') < f(solucao)$ **então**
        $solucao \leftarrow solucao'$;
    fim se
fim enqto
retorna $solucao$

Figura 4.1: Algoritmo paralelo para *threads* com um número fixo de iterações

$f(solucao) \leftarrow \infty$;
para $tarefa \leftarrow 1$ **até** $nIter$ **faça em paralelo**
    executaGRASP();
fim para
enquanto *houver respostas na fila* **faça**
    $solucao' \leftarrow$ obtemSolucao();
    se $f(solucao') < f(solucao)$ **então**
        $solucao \leftarrow solucao'$;
    fim se
fim enqto
retorna $solucao$

Figura 4.2: Algoritmo paralelo para *n* tarefas e um grupo de *threads*

A linguagem de programação Java foi escolhida por seu excelente suporte à multiprogramação. O programa desenvolvido por (ALVARENGA; ROCHA; PEREIRA, 2006) para o cálculo da árvore geradora de custo mínimo com grupamentos, originalmente escrito na linguagem C, foi adaptado para o ambiente Java e paralelizado usando a *framework* de execução de tarefas.

4.1 Estrutura básica do programa

O programa está estruturado em algumas classes, agrupadas em dois pacotes: um pacote principal que contém as classes envolvidas na execução e outro para as classes que representam o modelo de dados.

As classes que compõem o pacote principal são:

Main é a classe principal que recebe os parâmetros, cria um GerenciadorThreads e monta o arquivo de saída ao término da execução. Os parâmetros mais importantes são: α, que indica o tamanho da lista de candidatos para a solução inicial (seção 2.2); número de iterações; número de *threads*, em ambos os métodos será usado para limitar o número de processamentos paralelos.

GerenciadorThreads monta o grafo e coordena a execução das *threads* ou tarefas, retornando uma solução para a classe Main. Ela será mais detalhada nas seções 4.2 e 4.3, pois ela depende da abordagem usada. É interessante notar que o grafo utilizado será compartilhado por todas as *threads* ou tarefas e será somente para leitura. Isto é possível, pois de acordo com a seção 3.5.1, a montagem do grafo está na *thread* principal e a submissão das *threads* ou tarefas para execução gera um relacionamento *ocorre-antes*. Por este motivo, é seguro acessar o grafo nas *threads* ou tarefas, já que o modelo de memória do Java garante a publicação correta. A sincronização através de travas é dispensada neste caso.

TarefaCalculo Classe que executa o algoritmo da heurística GRASP para a árvore geradora de custo mínimo com grupamentos. Ela terá um comportamento ligeiramente diferente, dependendo da abordagem usada e isto será detalhado nas seções 4.2 e 4.3. Porém, o algoritmo principal foi derivado de (ALVARENGA; ROCHA; PEREIRA, 2006).

As classes que compõem o modelo de dados são as seguintes:

Grafo representa o grafo que está sendo usado como base para o cálculo da árvore geradora de custo mínimo. Ele também possui uma lista de Grupamentos. Ele é construído a partir de um arquivo, passado ao seu construtor.

Grupamento representa uma lista de vértices que será usada como um grupamento.

Aresta representa uma aresta no Grafo. Possui um nó de origem, de destino e um custo associado.

Solucao representa uma solução do problema, ou seja, uma árvore geradora de custo mínimo com grupamentos. Possui um custo e um conjunto de arestas que formam a árvore.

MatAgm encapsula a lista de arestas que representam uma árvore geradora, utilizada durante o algoritmo e para representar a solução.

Status agrupa variáveis usadas durante o processamento do algoritmo.

É interessante perceber que ambas as abordagens utilizam a *framework* de execução de tarefas do Java. Esta *fraemwork* permitiu uma rapidez muito grande na adaptação da primeira abordagem para a segunda, ou seja, a transformação de *threads* com um conjunto fixo de iterações para um grupo de tarefas em que cada um executa apenas uma iteração da heurística. Além disto, ela garante que a consistência sequencial será observada, já que a submissão de tarefas a um executador gera um relacionamento *ocorre-antes* entre o código que emite a submissão e a tarefa submetida (ver seção 3.5.1), assim como o envio da solução das tarefas para o GerenciadorThreads.

4.2 *threads* com um número fixo de interações

Nesta abordagem, a classe TarefaCalculo recebe um número de iterações que deve executar. Ela implementa a interface Callable, que se assemelha muito à interface Runnable. Porém, aquela é usada para representar execuções que retornam algum resultado, ao contrário da segunda interface que apenas representa blocos de código que não retornam nada.

A classe GerenciadorThreads é bastante simples. O método principal está reproduzido na figura 4.3. As linhas 8 a 12 criam o número definido de objetos TarefaCalculo, passando o número de iterações que cada um deverá executar. O número de objetos será igual ao número de *threads* passado por parâmetro ao programa. A variável complService é um objeto do tipo ExecutorCompletionService que representa o grupo de *threads*, além de fornecer um mecanismo seguro de retornar os resultados destes *threads*. A obtenção dos resultados está nas linhas 15 a 26, que iteram maxThreads vezes e recolhem os resultados de todas as tarefas. O método complService.take() na linha 17 retorna uma instância da interface

Future que representa um resultado que pode ainda não estar disponível. Este método bloqueará até que exista uma tarefa terminada. Após isto, a solução é obtida através do método resultTarefa .get() na linha 18, sendo comparado com o resultado atual. Se for melhor, a nova solução substituirá o resultado atual. Senão, será descartada.

```
1  public Solucao executa() throws IOException {
2     Solucao result = null;
3     try {
4        // Criar o grafo a partir do arquivo de entrada
5        grafo = new Grafo(entrada);
6
7        // Disparar as tarefas para serem executadas simultaneamente
8        for (int i = 0; i < maxThreads; i++) {
9           TarefaCalculo tarefa = new TarefaCalculo(grafo,
10             alpha, numIter / maxThreads);
11          complService.submit(tarefa);
12       }
13
14       // Coletar os resultados
15       for (int i = 0; i < maxThreads; i++) {
16          try {
17             Future<Solucao> resultTarefa = complService.take();
18             Solucao solucaoAtual = resultTarefa.get();
19             if (result == null || (soluca != null && result.compareTo(solucaoAtual) > 0)) {
20                result = solucaoAtual;
21             }
22          } catch (ExecutionException ex) {
23             Logger.getLogger(GerenciadorThreads.class.getName()).
24                log(Level.SEVERE, null, ex);
25          }
26       }
27    } catch (InterruptedException ex) {
28       Logger.getLogger(GerenciadorThreads.class.getName()).
29          log(Level.SEVERE, null, ex);
30    }
31    return result;
32 }
```

Figura 4.3: GerenciadorThreads com várias iterações por *threads*

A classe TarefaCalculo itera o número de vezes que foi instruído antes de retornar uma solução. O GerenciadorThreads só considerará as que não forem nulas, mesmo que uma solução inválida seja retornada.

4.3 Tarefas enfileiradas para um grupo de *threads*

Aqui, a classe TarefaCalculo também implementa a interface Callable, para que ela possa retornar um resultado ao objeto GerenciadorThreads. Porém, desta vez, ela executará apenas uma iteração da heurística e poderá não retornar corretamente, caso esta iteração não gere uma solução válida. Neste caso, será levantada uma exceção, que o objeto GerenciadorThreads tratará, registrando-a num arquivo para posterior consulta. Após isto, ele continuará aguardando o próximo resultado.

A classe GerenciadorThreads é ligeiramente diferente da versão anterior (figura 4.4). A grande diferença está no número de iterações utilizado para disparar as tarefas (linha 8), ou seja, uma tarefa para cada iteração. A coleta dos resultados também obedece este mesmo critério (linha 14). Para cada solução retornada, ela será comparada com a solução atual. Se for melhor, esta se tornará a solução atual. Senão, será descartada. Ao término, haverá uma solução que será a melhor entre todas as retornadas e esta, por sua vez, será retornada à classe Main para que o arquivo de saída seja gerado.

```
1  public Solucao executa() throws IOException {
2     Solucao result = null;
3     try {
4        // Criar o grafo a partir do arquivo de entrada
5        grafo = new Grafo(entrada);
6
7        // Disparar as tarefas para serem executadas simultaneamente
8        for (int i = 0; i < numIter; i++) {
9           TarefaCalculo tarefa = new TarefaCalculo(grafo, alpha);
10          complService.submit(tarefa);
11       }
12
13       // Coletar os resultados
14       for (int i = 0; i < numIter; i++) {
15          try {
16             Future<Solucao> resultTarefa = complService.take();
17             Solucao solucaoAtual = resultTarefa.get();
18             if (result == null || (soluca != null && result.compareTo(solucaoAtual) > 0)) {
19                result = solucaoAtual;
20             }
21          } catch (ExecutionException ex) {
22             Logger.getLogger(GerenciadorThreads.class.getName()).
23                log(Level.SEVERE, null, ex);
24          }
25       }
26    } catch (InterruptedException ex) {
27       Logger.getLogger(GerenciadorThreads.class.getName()).
28          log(Level.SEVERE, null, ex);
29    }
30    return result;
31 }
```

Figura 4.4: GerenciadorThreads com uma tarefa para cada iteração

Capítulo 5

Discussão e Resultados

5.1 Ambiente utilizado

Os testes foram realizados sobre um conjunto de instâncias do problema da AGMG disponibilizados em (DROR; HAOUARI; CHAOUACHI, 2000). Para cada instância, o algoritmo foi executado com o número de iterações igual a 5000. Além disto, este processo foi repetido 50 vezes para cada instância, para que fosse possível obtermos uma massa de dados razoável para a análise. A primeira execução de cada instância, dentro das 50, foi descartada, pois a máquina virtual Java possui um otimizador em tempo de execução que demora algum tempo para produzir o código binário mais adequado. Por isto, a primeira execução sempre gerou um tempo desproporcional às demais. As características de cada instância de testes estão presentes na tabela 5.1.

Foram utilizados dois computadores para a execução dos programas, sendo as configurações detalhadas abaixo:

- O ambiente A é um computador com processador Intel Core 2 Duo T7500, com a frequência de clock igual a 2.2 GHz. Ele possui 2 gigabytes de memória RAM e um winchester de 120 gigabytes. O sistema operacional utilizado foi a distribuição Arch Linux, sendo a versão do kernel igual a 2.6.30, empacotada pela distribuição. O ambiente Java foi o Sun Java Runtime Environment versão 1.6.0.14, sendo este o mais recente no momento da montagem deste trabalho. Para que os testes tivessem o mínimo de interferência do resto do sistema, eles ocorreram em um nível de execução sem interface

Instâncias	Nº de vértices	Nº de arestas	Nº de grupamentos
Gmst1	25	50	4
Gmst2	25	100	8
Gmst3	25	150	10
Gmst4	50	150	5
Gmst5	50	300	10
Gmst6	75	200	8
Gmst7	75	300	10
Gmst8	75	400	15
Gmst9	100	300	7
Gmst10	100	500	10
Gmst11	150	300	8
Gmst12	150	500	12
Gmst13	200	500	10
Gmst15	250	500	10
Gmst16	250	1000	25
Gmst17	300	1000	20
Gmst18	300	2000	30
Gmst19	300	3000	40
Gmst20	300	5000	50

Tabela 5.1: Características das instâncias de teste

gráfica e sem nenhum serviço adicional rodando simultaneamente. Isto se justifica pela natureza estritamente orientada a CPU do algoritmo testado.

- O ambiente B é um computador com processador Intel Core 2 Quad Q8200, com a frequência de clock igual a 2.33 GHz. Ele possui 4 gigabytes de memória RAM e um winchester de 320 gigabytes. O sistema operacional utilizado foi o Windows Vista Home Premium 64 bits. O ambiente Java foi o Sun Java Runtime Environment versão 1.6.14. A execução ocorreu em paralelo com o restante do sistema gráfico e antivírus, porém, o usuário não estava executando nenhuma aplicação pessoal além dos testes.

5.2 Avaliação das duas abordagens

5.2.1 Execução no ambiente de testes A

Conforme mencionado no capítulo 4, este trabalho explorou duas formas de paralelizar o algoritmo GRASP: (a) *threads* com um número definido de iterações para cada uma; (b) *threads* que consomem uma fila de tarefas, sendo que cada tarefa

corresponde a uma iteração da heurística. As tabelas 5.2 e 5.3 mostram os dados coletados após a execução de cada abordagem. As colunas chamadas Tempo são as médias de tempo obtidas durante as execuções de cada instância e o desvio padrão está na coluna Desvio. A tabela 5.2 sugere algumas observações:

	Máximo de *threads*				
	1		2		
Instância	Tempo	Desvio	Tempo	Desvio	*Speed up*
gmst01	192,33	13,85	175,63	48,07	1,10
gmst02	263,24	1,88	200,00	36,07	1,32
gmst03	311,71	4,42	218,82	36,24	1,42
gmst04	580,98	3,24	383,73	48,78	1,51
gmst05	954,10	5,41	580,80	49,00	1,64
gmst06	1.145,14	7,33	699,16	54,54	1,64
gmst07	1.587,08	6,73	942,08	67,25	1,68
gmst08	2.275,65	13,14	1.360,65	80,23	1,67
gmst09	2.006,90	12,85	1.198,43	66,16	1,67
gmst10	3.232,13	336,20	1.943,67	76,64	1,66
gmst11	2.190,45	14,18	1.401,76	96,32	1,56
gmst12	5.397,02	34,63	3.168,59	95,82	1,70
gmst13	5.674,80	40,77	3.449,47	133,99	1,65
gmst15	5.638,49	37,62	3.621,61	116,70	1,56
gmst16	25.181,80	247,15	14.200,39	192,72	1,77
gmst17	24.705,06	238,39	14.231,90	222,96	1,74
gmst18	63.056,86	498,33	35.283,24	458,35	1,79
gmst19	86.537,55	727,64	48.026,12	554,32	1,80
gmst20	275.019,02	10.949,65	155.570,96	9.507,74	1,77
				Média do *speed up*	1,63
				Desvio padrão	0,21

Tabela 5.2: Resumo para *threads* com *n* iterações no ambiente A (tempo em milissegundos)

- O tempo médio para o caso de 2 *threads* é sempre menor do que o caso sequencial, porém, para as instâncias menores, o *speed up* é menor do que para as instâncias maiores.
- O desvio padrão é consideravelmente maior para o caso de 2 *threads*. Apenas na instância gmst10 o desvio padrão é maior para o caso sequencial. Nas instâncias menores, o desvio indica que em certos casos, o tempo gasto pode ser até maior que o caso sequencial.
- A média do *speed up* foi relativamente boa, tendo sido bastante influenciada pelas instâncias menores. O desvio padrão desta média também foi bom. Os números indicam que o uso de 2 *threads* acelerou o processo em 1.63 vezes.

	Máximo de *threads*				
	1		2		
Instância	Tempo	Desvio	Tempo	Desvio	*Speed up*
gmst01	257,22	31,07	176,57	43,06	1,46
gmst02	319,65	10,66	188,04	30,27	1,70
gmst03	370,31	13,91	222,20	35,05	1,67
gmst04	679,98	30,96	411,84	58,66	1,65
gmst05	1.044,49	49,80	593,88	47,07	1,76
gmst06	1.255,92	43,59	714,04	49,55	1,76
gmst07	1.705,73	44,77	962,20	62,81	1,77
gmst08	2.427,37	52,64	1.365,71	65,25	1,78
gmst09	2.135,51	49,43	1.224,35	74,80	1,74
gmst10	3.346,80	42,39	1.910,14	85,23	1,75
gmst11	2.352,53	52,52	1.484,65	92,10	1,58
gmst12	5.516,76	53,58	3.232,41	136,87	1,71
gmst13	5.764,65	73,04	3.493,73	123,86	1,65
gmst15	5.656,12	58,14	3.657,92	114,98	1,55
gmst16	24.981,53	210,90	14.076,22	217,17	1,77
gmst17	24.146,41	237,15	13.920,08	194,66	1,73
gmst18	62.371,57	463,19	34.717,10	423,24	1,80
gmst19	86.699,12	712,35	47.870,53	409,33	1,81
gmst20	276.700,82	5.706,13	157.807,63	12.153,80	1,75
				Média do *speed up*	1,72
				Desvio padrão	0,17

Tabela 5.3: Resumo para tarefas enfileiradas num grupo de *threads* no ambiente A (tempo em milissegundos)

A tabela 5.3 sugere as seguintes observações:

- O *speed up* foi mais consistente entre as instâncias. Mesmo as instâncias menores tiverem um ganho bastante interessante. Para as instâncias maiores, não houve um grande ganho de desempenho em relação à abordagem anterior.

- O desvio padrão para o caso de 2 *threads* foi comparativamente semelhante ao caso sequencial. Isto sugere que o algoritmo paralelo aproveitou o processador de forma melhor do que a abordagem anterior.

- A média geral do *speed up* foi melhor do que na abordagem anterior, porém, o desvio padrão foi semelhante.

5.2.2 Execução no ambiente de testes B

As tabelas 5.4 e 5.5 mostram o resumo da execução no ambiente B para a abordagem com *threads* com um número definido de iterações para cada uma. Algumas observações emergem dos dados:

- Os casos com 2 e 3 *threads* tiveram um *speed up* de praticamente 2 e 3 vezes, respectivamente. Isto indica que o sistema operacional e o escalonamento de *threads* não interferiram em nada na execução. Já o caso de 4 threads sofreu mais interferência do meio externo, já que o sistema operacional e o escalonador precisaram interromper as threads, causando um ganho sensivelmente menor do que 4 vezes (3.69 em média).

- As instâncias menores tiveram ganhos menores. A causa provável é que algumas threads terminaram a execução antes de outras, conforme foi explicado na seção 4.3.

- O ganho geral de desempenho foi muito bom, ficando em média 3.69 vezes mais rápido do que o processamento sequencial puro, para 4 *threads*.

	Máximo de *threads*							
	1		2		3		4	
Instância	Tempo	Desvio	Tempo	Desvio	Tempo	Desvio	Tempo	Desvio
gmst01	185,18	2,83	95,82	1,93	66,51	4,38	54,33	8,65
gmst02	259,41	1,57	133,65	2,59	90,43	1,61	70,29	2,35
gmst03	306,73	2,33	156,49	1,93	106,63	2,34	81,84	3,57
gmst04	577,65	3,40	295,57	4,05	198,53	3,32	154,22	6,08
gmst05	969,78	8,89	491,92	4,24	332,24	3,35	252,29	3,61
gmst06	1.159,37	9,01	585,02	5,47	397,69	4,24	306,18	6,18
gmst07	1.614,86	14,96	813,90	6,55	550,63	5,61	420,71	7,67
gmst08	2.379,41	32,71	1.195,22	15,65	809,20	11,34	625,45	14,79
gmst09	2.033,27	14,55	1.032,12	8,50	699,20	8,06	566,82	19,44
gmst10	3.270,90	19,83	1.654,67	12,31	1.117,51	9,69	868,10	28,33
gmst11	2.170,65	20,29	1.113,45	8,81	769,55	8,74	624,45	26,16
gmst12	5.418,53	51,19	2.752,82	20,74	1.871,49	16,79	1.482,20	43,45
gmst13	5.556,90	52,16	2.854,49	26,65	1.968,59	30,64	1.574,61	40,63
gmst15	5.335,27	38,93	2.787,12	21,33	1.956,47	20,00	1.611,39	62,13
gmst16	24.921,27	312,89	12.617,29	152,44	8.537,00	112,66	6.608,12	102,83
gmst17	23.997,67	289,72	12.168,84	139,10	8.312,06	94,89	6.529,49	123,74
gmst18	63.114,59	953,72	31.553,37	360,11	21.250,41	300,65	16.594,90	324,36
gmst19	89.574,14	2.220,88	44.458,20	709,49	29.864,39	505,81	23.064,37	556,03
gmst20	287.598,78	7.407,87	140.106,96	1.579,77	95.350,14	3.041,13	74.383,53	1.832,63

Tabela 5.4: Resumo para *threads* com *n* iterações no ambiente B (tempo em milissegundos)

Instância	2 *threads*	3 *threads*	4 *threads*
gmst01	1,93	2,78	3,41
gmst02	1,94	2,87	3,69
gmst03	1,96	2,88	3,75
gmst04	1,95	2,91	3,75
gmst05	1,97	2,92	3,84
gmst06	1,98	2,92	3,79
gmst07	1,98	2,93	3,84
gmst08	1,99	2,94	3,80
gmst09	1,97	2,91	3,59
gmst10	1,98	2,93	3,77
gmst11	1,95	2,82	3,48
gmst12	1,97	2,90	3,66
gmst13	1,95	2,82	3,53
gmst15	1,91	2,73	3,31
gmst16	1,98	2,92	3,77
gmst17	1,97	2,89	3,68
gmst18	2,00	2,97	3,80
gmst19	2,01	3,00	3,88
gmst20	2,05	3,02	3,87
Média	1,97	2,9	3,69
Desvio padrão	0,03	0,07	0,16

Tabela 5.5: *Speed up* para *threads* com *n* iterações no ambiente B

As tabelas 5.6 e 5.7 mostram o resumo dos resultados para uma lista de tarefas executadas por um grupo de *threads*, variando de 1 a 4. Elas nos sugerem as seguintes observações:

- Os casos com 2 e 3 *threads* também foram bem melhores do que os casos com 4 *threads*, porém, nesta abordagem de uma lista de tarefas, o ganho foi muito mais consistente.
- O *speed up* para o caso de 4 *threads* foi melhor do que o equivalente para a abordagem de *threads* com um número fixo de iterações. O grupo de threads buscando as iterações de uma fila permitiu que os processadores disponíveis fossem explorados de forma contínua, minimizando a situação que ocorreu na abordagem anterior em que algumas threads terminam antes de outras, fazendo que alguns processadores fiquem ociosos.

	Máximo de *threads*							
	1		2		3		4	
Instância	Tempo	Desvio	Tempo	Desvio	Tempo	Desvio	Tempo	Desvio
gmst01	235,80	19,04	116,14	8,68	74,10	6,21	66,06	11,29
gmst02	312,55	12,26	152,35	9,99	94,31	2,83	77,96	4,18
gmst03	352,88	18,49	174,96	9,01	108,63	2,50	86,82	4,90
gmst04	624,27	15,92	310,69	10,04	202,27	4,86	159,98	6,34
gmst05	978,57	15,90	485,51	8,26	324,22	4,53	247,92	4,58
gmst06	1.175,33	14,43	590,29	8,77	394,55	4,86	308,20	7,76
gmst07	1.613,78	17,11	804,08	9,13	537,53	5,35	416,76	7,42
gmst08	2.263,00	23,59	1.132,51	11,06	764,61	18,51	589,31	14,46
gmst09	2.041,24	20,48	1.023,27	11,04	692,94	10,34	553,45	12,81
gmst10	3.359,61	41,37	1.686,53	11,25	1.135,20	9,50	879,27	15,94
gmst11	2.277,22	36,00	1.180,35	50,34	806,92	25,92	650,90	27,19
gmst12	5.403,37	39,48	2.745,41	45,31	1.859,94	48,38	1.458,08	54,30
gmst13	5.780,37	63,55	2.959,45	54,73	2.038,96	50,32	1.619,90	47,93
gmst15	5.717,49	50,42	3.006,80	59,19	2.099,22	55,15	1.692,82	46,81
gmst16	24.562,43	281,94	12.425,73	144,25	8.388,12	126,32	6.488,51	106,39
gmst17	24.757,27	248,02	12.558,69	139,29	8.528,53	108,37	6.651,49	90,08
gmst18	62.207,55	1.046,27	31.212,71	463,96	21.048,88	463,39	16.507,63	761,71
gmst19	84.395,29	1.895,09	41.940,53	777,80	28.320,20	620,84	21.718,84	675,33
gmst20	263.046,67	7.877,72	128.094,55	5.494,23	86.904,98	2.563,13	67.145,94	1.790,53

Tabela 5.6: Resumo para tarefas enfileiradas num grupo de *threads* no ambiente B (tempo em milissegundos)

Instância	2 *threads*	3 *threads*	4 *threads*
gmst01	2,03	3,18	3,57
gmst02	2,05	3,31	4,01
gmst03	2,02	3,25	4,06
gmst04	2,01	3,09	3,90
gmst05	2,02	3,02	3,95
gmst06	1,99	2,98	3,81
gmst07	2,01	3,00	3,87
gmst08	2,00	2,96	3,84
gmst09	1,99	2,95	3,69
gmst10	1,99	2,96	3,82
gmst11	1,93	2,82	3,50
gmst12	1,97	2,91	3,71
gmst13	1,95	2,83	3,57
gmst15	1,90	2,72	3,38
gmst16	1,98	2,93	3,79
gmst17	1,97	2,90	3,72
gmst18	1,99	2,96	3,77
gmst19	2,01	2,98	3,89
gmst20	2,05	3,03	3,92
Média	1,99	2,99	3,78
Desvio padrão	0,04	0,14	0,18

Tabela 5.7: *Speed up* para tarefas enfileiradas num grupo de *threads* no ambiente B

Capítulo 6

Conclusão

A tendência da indústria é a disponibilização de processadores com cada vez mais núcleos a preços comparáveis aos processadores simples anteriormente fabricados. A causa principal para isto é a impossibilidade de fornecer um aumento contínuo da frequência de processamento, devido a impossibilidades físicas dos materiais utilizados. Por isto, a responsabilidade pelo aumento da eficiência de execução recai sobre os programadores, que devem cada vez mais compreender e dominar as técnicas para a criação de programas que explorem o paralelismo inerente às novas arquiteturas de hardware.

O paralelismo pode ser explorado por meio da distribuição de processos em várias máquinas, utilizando-se *clusters* ou pela utilização de *threads* dentro de um mesmo programa. Várias linguagens fornecem mecanismos variados para o desenvolvedor alcançar um nível de paralelismo adequado ao hardware moderno.

O presente trabalho utilizou-se da linguagem Java e sua biblioteca padrão, que fornece um conjunto de classes muito interessante para a separação de processamento em tarefas, chamada *framework* de execução de tarefas. Ela é ao mesmo tempo simples e poderosa. O problema da Árvore Geradora de Custo Mínimo com Grupamentos foi o pano de fundo para a montagem de uma solução baseada na heurística GRASP. Foram exploradas duas formas de paralelização com *threads*: a primeira simulou um *cluster* local, em que um número de *threads* recebia a incumbência de executar n iterações da heurística; a segunda foi estruturada de forma a criar uma lista de tarefas, em que cada uma representava uma das iterações da heurística. Um grupo de *threads* foi o responsável por capturar a próxima tarefa disponível e executá-la, sendo que cada *thread* do grupo ficou constantemente ocupada. Esta segunda abordagem mostrou-se mais eficiente, embora a diferença

tenha sido pequena. Porém, a estrutura da solução sugere que ela possui uma escalabilidade melhor do que a primeira, ganhando mais desempenho à medida em que o número de processadores aumente.

Algumas sugestões para trabalhos futuros são: utilização de uma estrutura mista de *clusters* e *threads*, em que cada nó do *cluster* possua um processador com mais de um núcleo. Outra possibilidade é a utilização de técnicas diferentes para a solução da AGMG, tal como o melhoramento da heurística GRASP chamado *Path Relinking*. Este melhoramento tende a encontrar soluções melhores do que as normalmente encontradas pela heurística GRASP, pois mantém um conjunto das melhores soluções e tenta mesclar a solução atual com algum elemento desta lista, fazendo com que se fuja de ótimos locais,

Referências Bibliográficas

ALVARENGA, F. V. d.; ROCHA, M. L.; PEREIRA, M. R. Implementação Paralela de uma Metaheurística grasp com Path-Relinking para o Problema da Árvore Geradora de Custo Mínimo com Grupamentos. 2006.

DROR, M.; HAOUARI, M.; CHAOUACHI, J. *Generalized Spanning Trees*. 2000. 583-592 p.

FEO, T. A.; REZENDE, M. G. C. A Probabilistic Heuristic for a Computacionally Dificult Set Covering Problem. *Operations Research Letters*, n. 8, p. 67–81, 1989.

FLYNN, M. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21, p. 948, 1972.

GOETZ, B.; PEIERLS, T.; BLOCH, J.; BOWBEER, J.; HOLMES, D.; LEA, D. *Java Concurrency in Practice*. Stoughton, MA, USA: Addison-Wesley, 2006. ISBN 978-0-321-340960-6.

HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming*. Burlington, MA, USA: Morgan Kauffman Publishers, 2008. ISBN 978-0-12-370591-4.

MYUNG, Y. S.; LEE, C. H.; TCHA, D. W. On the Generalized Minimun Spanning Tree Problem. *Networks*, n. 26, p. 231–341, 1995.

SHYU, S. J.; YIN, P. Y.; LIN, B. M. T.; HAOURARI, M. Upper and Lower Bounding Strategies for the Generalized Minimun Spanning Tree Problem. *European Journal of Operational Research*, n. 171, p. 632,647, 2006.