Paralelização de Metaheurísticas para Execução Autonômica em Grades Computacionais

Aletéia P. F. Araújo
Departamento de Computação, Universidade Católica de Brasília
aleteia@ucb.br

Celso Ribeiro, Cristina Boeres, Vinod Rebello
Instituto de Computação, Universidade Federal Fluminense
{celso, boeres, vinod}@ic.uff.br

Resumo

Na busca por melhores serviços ou maiores lucros, a utilização de metaheurísticas tem sido um importante aliado da indústria para resolver questões operacionais complexas em tempos computacionais aceitáveis. O desenvolvimento de metaheurísticas paralelas eficientes é difícil e, para executar instâncias reais, os algoritmos necessitam de muito poder computacional. Enquanto a computação em grades pode oferecer tal poder computacional, suas características específicas criam uma complexidade adicional para desenvolver aplicações eficientes. Este trabalho propõe uma estratégia simples de paralelização para executar metaheurísticas seqüenciais em grades computacionais. O objetivo é eliminar a necessidade do desenvolvedor encarar a tarefa de paralelizar uma metaheurística, e mostrar que executando múltiplas instâncias de uma metaheurística seqüencial de forma coordenada em paralelo é possível reduzir o tempo para alcançar boas soluções. A paralelização proposta é composta de duas camadas: um middleware de gerenciamento da execução na grade e a estratégia de coordenação das metaheurísticas seqüenciais. Para validar essa proposta foram desenvolvidas duas novas metaheurísticas paralelas, uma para o problema do torneio com viagens espelhado e a outra para o problema da árvore geradora de custo mínimo com restrição de diâmetro. Ambas as paralelizações foram capazes de melhorar, para várias instâncias, os melhores resultados conhecidos na literatura.

1. Introdução

Com o advento da grade computacional [8], é possível oferecer uma plataforma suficientemente robusta capaz de executar aplicações que necessitem de um alto poder de processamento ou de armazenamento. Entretanto, a exploração eficiente desse tipo de ambiente ainda é um desafio. Dentre as principais dificuldades, destacam-se a escalabilidade, a heterogeneidade de recursos, o dinamismo, a volatilidade relacionada a falhas na rede, e a distribuição de recursos em múltiplos domínios administrativos. Dessa forma, para garantir que as aplicações possam se beneficiar do ambiente sem o conhecimento das complexidades inerentes à grade, é fundamental desenvolver tecnologia que possa tornar transparente a gerência dessa plataforma.

Problemas de otimização combinatória surgem, na prática, como um amplo universo de aplicações a serem tratadas pelo paralelismo em ambientes de grades, pois freqüentemente são NP-difíceis e, conseqüentemente, consomem muito tempo de CPU. Metaheurísticas são procedimentos de alto nível que coordenam heurísticas simples, objetivando explorar efetivamente o espaço de busca para encontrar boas soluções aproximadas (freqüentemente ótimas) para problemas de otimização combinatória.

Cung et al. [5] mencionam que a paralelização de metaheurísticas não visa somente diminuir os tempos de processamento, tornando possível a resolução de problemas maiores e mais realistas, mas também possibilita encontrar soluções melhores do que as obtidas por algoritmos seqüenciais. Entretanto, ainda há poucas implementações de metaheurísticas paralelas para grades computacionais e, dentre as implementações existentes, a grande maioria adota estratégias inadequadas de paralelização, resultando em algoritmos que não conseguem aproveitar toda a capacidade de processamento fornecida pela grade, por não lidarem com as características específicas desse novo ambiente.

O objetivo deste trabalho é facilitar o desenvolvimento de metaheurísticas paralelas que também podem se beneficiar eficientemente da execução em ambientes de grade computacional. Para isso, uma estratégia de paralelização em duas camadas é proposta. A camada de *middleware* é

responsável por garantir a execução autonômica da metaheurística paralela no ambiente da grade, embutindo os mecanismos de tolerância a falhas, escalonamento de tarefas e monitoramento do ambiente na própria aplicação. A camada de aplicação adota uma hierarquia distribuída de *pools* de soluções de elite, objetivando proporcionar a cooperação entre instâncias de metaheurísticas seqüenciais durante a execução sem acarretar perda de desempenho.

Os problemas do torneio com viagens espelhado (mTTP) [14] e da árvore geradora de custo mínimo com restrição de diâmetro (AGMD) [15] foram escolhidos como casos de teste para validar a estratégia proposta para grades computacionais. Essa escolha foi devido a existência de algoritmos seqüenciais de alta qualidade para ambos os problemas, os quais serão usados em estudo comparativo.

2 O Problema do Torneio com Viagens Espelhado

O problema do torneio com viagens (TTP, do inglês *Traveling Tournament Problem*) é um problema de geração de tabelas proposto inicialmente por Easton et al. [6]. O TTP consiste em gerar uma programação de jogos na qual cada equipe deve jogar duas vezes com cada uma das outras equipes, de forma que: nenhuma das equipes jogue mais de três jogos consecutivos em casa ou fora de casa; não aconteçam jogos consecutivos entre as mesmas duas equipes; e a distância total percorrida pelas equipes durante o torneio seja minimizada. Assume-se que cada equipe tem um estádio na sua cidade e as distâncias entre cada par de estádios são conhecidas e simétricas.

O problema do torneio com viagens espelhado (mTTP, do inglês *mirrored Traveling Tournament Problem*) representa um caso particular do TTP. O mTTP pode ser visto como um torneio simples em suas $n-1$ primeiras rodadas, chamado de primeiro turno, seguido pelo mesmo torneio com os mandos de campo invertidos em relação às $n-1$ primeiras rodadas, chamado de returno. Isso significa que a ordem dos jogos realizados no returno é exatamente a mesma dos jogos do turno, apenas com os estádios invertidos, quem jogou em casa jogará fora e vice-versa.

A grande variedade de combinações possíveis, acrescentada das diversas restrições impostas pelo mTTP, faz com que esse seja um problema de otimização NP-difícil [6]. Como conseqüência, a maior instância teste para o qual a solução ótima é conhecida é composta de apenas oito equipes. Essa solução foi obtida por um algoritmo exato paralelo após quatro dias de processamento usando 20 máquinas Pentium II com 512 Mbytes de memória RAM [7].

2.1 Heurística Híbrida para o mTTP

Ribeiro e Urrutia [14] propuseram a heurística híbrida `GRILS-mTTP` para o mTTP. Essa heurística é baseada na hibridação das metaheurísticas GRASP [13] e ILS [12]. Soluções gulosas randomizadas são construídas na fase GRASP e passadas para a fase de busca local ILS. A fase ILS é repetida até que um critério de reinicialização seja satisfeito, nesse caso uma solução inicial é gerada novamente na fase GRASP.

A fase ILS inicia aplicando uma perturbação randômica na solução corrente e uma busca local na solução resultante. Se o novo ótimo local satisfizer o critério de aceitação, ele tornar-se a nova solução corrente. A heurística contínua executando sucessivamente as fases de construção GRASP e busca local ILS, até que um critério de parada seja atingido.

Apesar da sua eficácia, a heurística `GRILS-mTTP` é uma implementação que demanda muito tempo de processamento. Para acelerar a busca, quatro implementações paralelas foram desenvolvidas. Todas as implementações basearam-se no paradigma de programação mestre-trabalhador, mas usaram diferentes graus de cooperação [1]. A melhor implementação, PAR-MP, descrita abaixo, foi usada nos testes realizados neste artigo.

2.2 Algoritmo Paralelo PAR-MP

O algoritmo paralelo PAR-MP [1] adotou o paradigma de paralelização mestre-trabalhador, tradicional na área de otimização, a fim de explorar as vantagens de um cluster de computadores ou grade. Embora essa aplicação em MPI tenha sido basicamente constituída por um único processo mestre responsável por coordenar vários processos trabalhadores (cada um executando uma versão ligeiramente modificada da heurística `GRILS-mTTP`), foi também desenvolvida uma técnica de cooperação entre os trabalhadores.

Essa cooperação entre os trabalhadores foi implementada através de um *pool* de soluções de elite mantido pelo mestre. Os trabalhadores cooperam indiretamente entre si através da troca de soluções de elite encontrada ao longo das suas trajetórias de busca. Cada trabalhador ao terminar uma iteração completa da heurística `GRILS-mTTP`, envia a melhor solução encontrada para o mestre. O mestre gerencia esse *pool* central, coletando e distribuindo soluções de elite de acordo com a demanda. O objetivo com a cooperação é alcançar uma performance com a busca global paralela melhor do que a performance com a concatenação dos resultados dos processos individuais.

O algoritmo PAR-MP obteve bom desempenho quando comparado com a solução seqüencial para o problema, alcançando uma aceleração razoável em *cluster* de processadores, e com a implementação paralela sem cooperação (i.e. execução em paralelo de instâncias independentes da

GRILS-mTTP). Também foi verificada a sua habilidade de aproveitamento de recursos de uma grade através de melhoria de desempenho de execução e qualidade de solução. No entanto, vários fatores necessitam ser tratados eficientemente para que todo o potencial do ambiente em questão possa ser aproveitado, e isso não pode ser totalmente viabilizado através do paradigma mestre-trabalhador. Nesse paradigma, o mestre é um potencial gargalo e, questões como descoberta de recursos, seleção, alocação de tarefas, escalonamento e tolerância a falhas devem ser tratados para que a execução eficiente em uma grade seja alcançada.

3 O Problema da Árvore Geradora de Custo Mínimo com Restrição de Diâmetro

Para uma definição formal desse problema, considera-se um grafo conexo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, com um custo $c_{ij} \geq 0$ associado a cada aresta $(i, j) \in E$. Um subgrafo $T = (V, E')$ conexo e sem ciclos de G é denominado de árvore gerado de G. Uma árvore geradora com $|V|$ vértices possui $|E| = |V| - 1$ arestas. Uma árvore T de um grafo é denominada Árvore Geradora Mínima (AGM) de G se, e somente se, seu custo for mínimo dentre todas as árvores geradoras possíveis.

O diâmetro de uma árvore é a quantidade de arestas em seu maior caminho. Assim, o problema da Árvore Geradora de Custo Mínimo com Restrição de Diâmetro (AGMD) consiste em encontrar uma árvore geradora de G com custo mínimo, tal que qualquer caminho nesta árvore contenha no máximo D arestas, onde $D \geq 2$. Esse problema é NP-difícil [9] para diâmetros $4 \leq D < |V| - 1$.

Na literatura há vários trabalhos que exploram o problema da AGMD. Os métodos usados variam desde algoritmos exatos [10] até heurísticos [11]. Dentre os métodos heurísticos, uma das propostas mais eficiente foi a metaheurística híbrida proposta por Santos e Ribeiro em [15].

3.1 Heurística Híbrida para o Problema da AGMD

A heurística híbrida proposta em [15] para o problema da AGMD, também é baseado na hibridação das metaheurísticas GRASP e ILS. Na fase GRASP são construídas soluções iniciais por meio de uma heurística gulosa chamada OTT-M2. O algoritmo OTT-M2 inicia com uma árvore consistindo de um único e arbitrário nó, escolhido randomicamente, e repetidamente novos nós na árvore são adicionados, escolhidos randomicamente a partir de uma lista restrita de candidatos. Isso é repetido até que todos os nós sejam conectados na árvore.

A heurística OTT-M2 não garante soluções viáveis para grafos esparsos. Para resolver esse problema, falsas arestas são inseridas com alto custo para completar o grafo. A fim de garantir que a fase de busca local ILS seja aplicada apenas em boas soluções, o número de falsas arestas permitido é controlado por um filtro. Dessa maneira, soluções iniciais são construídas até que uma solução viável seja encontrada.

Em seguida, a fase de busca local ILS é aplicada à solução inicial. Essa fase inicia aplicando uma perturbação randômica na solução corrente. Após a perturbação, um segundo filtro é usado para selecionar dentre as soluções perturbadas, em qual a busca local será aplicada. O segundo filtro é usado para garantir que a busca local é aplicada somente em boas soluções. Caso a solução encontrada satisfaça um dado critério de aceitação, esse ótimo local tornar-se a nova solução corrente. Todos esses passos são repetidos até que o critério de parada seja atingido.

4 Uma Nova Estratégia de Paralelização para Metaheurísticas

As estratégias de paralelização mais típicas enfatizam apenas o problema a ser paralelizado, sem preocupação alguma com as características do ambiente usado na execução. Para tratar esse problema, neste trabalho é proposta uma estratégia de desenvolvimento de metaheurísticas paralelas, na qual, além da especificação da estrutura das tarefas paralelas, há também uma preocupação com a gerência da execução eficiente da aplicação. Com este controle embutido na aplicação, ela tornar-se autônoma para escolher quais decisões devem ser tomadas diante de eventuais mudanças no ambiente.

Para tornar a estratégia de paralelização para grades transparente ao desenvolvedor de metaheuristicas seqüenciais, adotou-se um projeto no qual a metaheurística paralela criada é composto de duas camadas independentes. As funções relacionadas à metaheuristica são desenvolvidas na camada de aplicação, e as funções relacionadas à gerência da aplicação, na camada de *middleware*.

4.1 Camada de Aplicação

Nesta camada são implementadas as funções relacionadas à aplicação paralela. A estratégia proposta, considerando as características da solução heurística e o poder computacional que pode ser oferecido pelo ambiente, é hierárquica distribuída. O objetivo é estruturar a aplicação de tal maneira que ela possa facilmente se adaptar ao ambiente de grade, e garantir um bom nível de cooperação entre os processos da aplicação, independentemente da escala do ambiente, da heterogeneidade dos recursos e do número de processos necessários para concluir a execução.

Com essa estratégia, a aplicação a ser paralelizada é subdividida hierarquicamente em três níveis, como apresentado na Figura 1. No topo da hierarquia há um único processo,

responsável por manter um conjunto das melhores (elite) soluções em um *pool* de cooperação (PC). Esse processo tem a função de permitir a troca de soluções entre processes que estão em diferentes sítios do ambiente de grade.

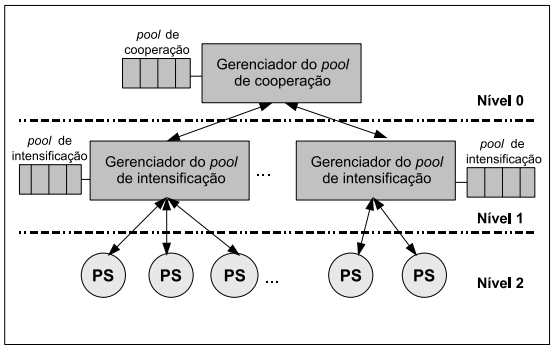

Figura 1. A Camada de Aplicação.

No nível seguinte, há um processo gerenciador de um *pool* local em cada sítio do ambiente participante da execução, chamado *pool* de intensificação (PI). O gerenciador desse *pool* é responsável por armazenar as melhores soluções encontradas pelos processos que estão executando em máquinas locais.

O último nível é formado pelos processos provedores de solução (PS) que executam efetivamente as heurísticas. Eles executam independentemente dos demais processos, aproveitando a estrutura hierárquica dos *pools* criados para garantir a cooperação com todos os processos provedores de solução, sem que isso acarrete uma sobrecarga ao sistema. A função específica de cada provedor dependerá do problema a ser resolvido pelos mesmos. As próximas seções descrevem as principais funções desempenhadas em cada nível dessa estratégia.

4.1.1 Provedor de Solução:

Neste nível, são implementadas as heurísticas que vão gerar as soluções do problema, de maneira que as mesmas possam aproveitar eficientemente da estrutura hierárquica de *pools* proposta para a troca de informação.

Os provedores de solução são independentes e assíncronos, mas compartilham informações através dos *pools* de soluções de elite para acelerar a convergência para boas soluções [5]. Tais provedores podem ser homogêneos ou heterogêneos entre si: em uma mesma execução pode haver provedores de solução executando heurísticas iguais ou diferentes. Nesse sentido, é possível expandir essa heterogeneidade para que se tenha também diferentes métodos, como, por exemplo, métodos exatos e heurísticos executando paralelamente na resolução do mesmo problema.

É fato que, a maioria das metaheurísticas consiste em uma fase de construção de uma solução inicial, seguida por uma fase de melhoria dessa solução por meio de uma busca local. Para que uma instância da heurística seqüencial possa se beneficiar do trabalho de uma outra instância, definiu-se que cada processo provedor de solução pode optar por: iniciar sua execução com soluções armazenadas no *pool* de intensificação, associado ao sítio no qual ele está executando; ou iniciar construindo uma nova solução. No primeiro caso, é necessário que o provedor mande uma mensagem para o processo que controla o *pool* de intensificação solicitando uma ou mais soluções de elite nele armazenado. Após receber a solução, o processo provedor inicia sua execução a partir de uma heurística de busca, objetivando melhorá-la. No segundo caso, o processo provedor constrói uma nova solução a partir de uma heurística construtiva, conforme algoritmo seqüencial.

Em ambos os casos, o provedor envia a melhor solução encontrada para ser armazenada no *pool* de intensificação do sítio. Ou seja, uma das funções dos processos provedores é alimentar esse *pool* com as suas soluções.

4.1.2 Gerenciador do *Pool* de Intensificação (PI):

O processo gerenciador de um *pool* de intensificação é responsável por três operações: inserir novas soluções no *pool* (em posições apropriadas), escolher uma solução do *pool* (de acordo com a demanda dos processos provedores de soluções), e abastecer o *pool* de cooperação com as melhores soluções existentes no seu *pool*.

A primeira operação é ativada todas as vezes que o gerenciador do *pool* de intensificação recebe uma solução a partir de um processo provedor. Se a solução recebida for melhor do que a solução de elite a partir do qual ela foi gerada, a nova solução obrigatoriamente substituirá a anterior. Por outro lado, se a nova solução não tiver sido gerada a partir de uma solução deste *pool*, ela poderá ser inserida em qualquer posição disponível. Todavia, se o *pool* estiver completamente preenchido, a nova solução S_n substituirá a pior solução S_p somente se S_n for melhor que S_p. Note que, todas as soluções no *pool* devem ser diferentes.

A segunda operação ocorre todas as vezes que um processo provedor solicita uma solução de elite. Caso o *pool* de intensificação ainda esteja vazio, o processo gerenciador do *pool* informa ao processo solicitante que não há solução para ser enviada. Nesse caso, o processo solicitante é obrigado a iniciar sua execução construindo uma nova solução. Quando o *pool* não está vazio, o gerenciador do *pool* escolhe aleatoriamente uma solução de elite e a envia ao processo solicitante.

A terceira operação ocorre todas as vezes em que um *pool* de intensificação é preenchido. Imediatamente após o preenchimento, as melhores soluções do mesmo (chamada

de *grupo seleto*) são enviadas para serem armazenadas no *pool* de cooperação. Após enviar o grupo seleto de soluções, o processo gerenciador do *pool* de intensificação envia todas as atualizações que ocorrerem com as soluções do grupo seleto para o gerenciador do *pool* de cooperação. Assim, o *pool* de cooperação sempre manterá as melhores soluções encontradas em todos os sítios envolvidos na execução.

As metaheurísticas tendem a um estado de estabilização, onde melhorias das soluções existentes ocorrem com pouca freqüência. O processo gerenciador do *pool* de intensificação é dito ter estabilizado quando um dado número max de mensagens consecutivas tiver sido recebido de seus provedores com soluções que não tenham sido aproveitadas para inclusão no *pool*. Para escapar desse ótimo local, o *pool* de intensificação deverá executar um procedimento de renovação. Isso significa que o gerenciador do *pool* de intensificação receberá um subconjunto de soluções selecionadas aleatoriamente a partir do *pool* de cooperação, e esvaziará todas as demais posições.

4.1.3 Gerenciador do *Pool* de Cooperação (PC):

No topo da hierarquia, há o processo gerenciador do *pool* de cooperação, cuja principal função é garantir a troca de informações entre os processos provedores de solução que estão em diferentes sítios. Essa cooperação entre eles é fundamental para melhorar o desempenho da aplicação.

O processo gerenciador desse *pool* recebe as melhores soluções de cada *pool* de intensificação e, quando algum gerenciador de intensificação estabiliza, ele escolhe uma fração das suas soluções e envia imediatamente para o processo estabilizado, a fim de que o mesmo possa executar o procedimento de renovação, escapando do ótimo local.

Neste *pool*, novas soluções são inseridas em uma posição vazia. Quando o *pool* está preenchido, as novas soluções atualizam as piores. Dessa forma, o *pool* de cooperação é mantido exclusivamente pelas soluções enviadas por cada sítio, tornando-se um depósito das melhores soluções encontradas durante a execução da aplicação.

A estrutura distribuída e hierárquica de *pools* faz com que a maior parte da comunicação aconteça localmente, entre os processos que executam no mesmo sítio. Em ambiente de grade, isso significa que a maioria das mensagens são trocada em redes locais, ou seja, dentro de um mesmo sítio envolvendo os processos provedores e o seu gerenciador local. Com isso, a sobrecarga de comunicação não causa grande interferência no desempenho da aplicação.

Contudo, nota-se que questões relacionadas diretamente com a gerência do ambiente devem ser tratadas para que a aplicação torne-se autonômica e capaz de executar eficientemente, independente das mudanças no ambiente. Nesse contexto, este trabalho propõe a implementação de uma camada adicional para tratar as características que independem da aplicação.

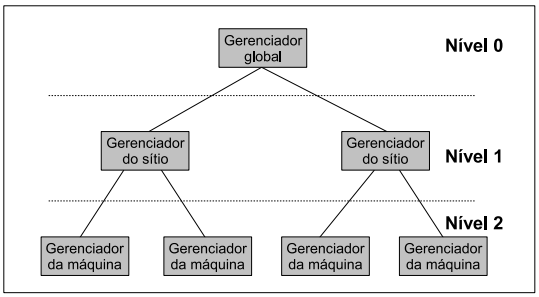

Figura 2. Camada de Middleware.

4.2 Camada de Middleware

Nesta camada estão todas as funções de gerenciamento e monitoramento do ambiente. Dentre as suas principais responsabilidades estão: o balanceamento de carga, a tolerância a falhas, a criação de tarefas e o monitoramento do ambiente. Para garantir a viabilidade dessa camada e a total transparência da implementação dessas funções, essa camada foi implementada através de um *middleware*.

O *middleware* utilizado para realizar a gerência da aplicação foi EasyGrid [4] adaptado para aplicações metaheurísticas, chamado assim de metaEasyGrid [2]. Esse *middleware* é automaticamente embutido nas aplicações paralelas, transformando-as em implementações autonômicas. Aplicações desse tipo são adaptativas, robustas à falhas de recursos e capazes de reagirem às mudanças do sistema que podem ocorrer em ambientes dinâmicos.

O metaEasyGrid é baseado em implementações MPI que suportam criação dinâmica de processos (tais como MPI/LAM). O metaEasyGrid adota uma arquitetura hierárquica distribuída, dividida em três níveis de gerência, conforme indicado na Figura 2. No topo da hierarquia (nível 0), surge o Gerenciador Global (GG), encarregado de gerenciar todos os sítios envolvidos na execução da aplicação.

O nível 1 está relacionado aos processos Gerenciadores dos Sítios (GS), que respondem pelo gerenciamento dos processos da camada da aplicação atribuídos a cada sítio. Para finalizar, o nível 2 é composto pelos Gerenciadores Locais das Máquinas (GM), responsáveis pelo escalonamento, criação e execução de processos da camada da aplicação atribuídos à máquina local. Essa arquitetura foi adotada com o intuito de adequar os processos gerenciadores à organização dos recursos pela grade, minimizando o custo embutido pelo gerenciamento da aplicação.

Uma característica chave para que a aplicação gerenciada pelo EasyGrid alcance melhor desempenho em grades computacionais é o seu modelo de execução. As aplicações devem ser escritas de tal maneira que o paralelismo

disponível seja maximizado, independentemente do número de processadores disponíveis. Embora o modelo tradicional de "um processo por processador" seja eficiente para ambientes dedicados e homogêneos (como os *clusters*) [4], uma implementação que usa um maior número de processos menores, pode obter melhor desempenho na grade, apesar das sobrecargas causadas pela criação de processos [4].

Para maior escalabilidade da aplicação em MPI, melhor balanceamento de carga entre os processadores e menor sobrecarga associada a tolerância a falhas, o metaEasyGrid não cria todos os processos provedores de solução de uma só vez. A criação desses processos é realizada dinamicamente e de maneira orquestrada de acordo com a política de escalonamento distribuída.

Dessa maneira, cada GS deve monitorar o estado do seu sítio, mantendo a carga equilibrada, de acordo com o poder computacional oferecido pelos recursos. Quando o GS detecta que o trabalho restante a ser executado atingiu um limite inferior, um pedido para o GG criar um novo bloco de tarefas é realizado. Ao receber as novas tarefas, o GS ativa o escalonador de tarefas para que as mesmas sejam distribuídas entre as máquinas pertencentes aquele sítio. Assim, os sítios sempre têm carga suficiente a ser executada, não permitindo que recursos fiquem ociosos.

5 Resultados Experimentais

Esta seção sumariza alguns dos experimentos realizados para avaliar a estratégia hierárquica distribuída, usada para implementar metaheurísticas paralelas para o mTTP e para o problema da AGMD, denotadas por AUDImTTP e AUDI-AGMD, respectivamente. A implementação AUDImTTP foi comparada com a versão paralela PAR-MP, e a AUDI-AGMD foi comparada com a heurística híbrida seqüencial proposta por Santos e Ribeiro em [15]. Ambas as metaheurísticas paralelas foram implementadas usando C++ e a versão 7.0.6 do MPI/LAM.

Os experimentos usaram até 60 máquinas de três sítios do Grid Sinergia, localizados em três cidades diferentes dentro do estado do Rio de Janeiro: (a) um cluster, no Rio de Janeiro, com 22 máquinas Pentium II 400 MHz, (b) um em Niterói com 28 máquinas Pentium IV 2.6 GHz e 3 máquinas Pentium IV 3.2 GHz, a partir de 40 km do cluster localizado no Rio de Janeiro, e um outro cluster (c), na cidade de Petrópolis, com sete máquinas Pentium IV 3.2 GHz, distante a 100 quilômetros do cluster (a). Essa grade adota a versão 2.4 do Globus Toolkit entre os sítios participantes.

5.1 Experimentos com o AUDImTTP

Quando desenvolvida, a metaheurística seqüencial `GRILS-mTTP`, melhorou para todas as 22 instâncias oficiais do mTTP [7] as melhores soluções da literatura. A implementação AUDImTTP conseguiu melhorar 16 das melhores soluções alcançadas pela versão seqüencial, e dentre essas 16, nove ainda são os melhores resultados conhecidos na literatura atualmente [7].

Para o experimento realizado com o AUDImTTP foram considerados cinco instâncias oficiais do mTTP [7], para as quais a versão paralela PAR-MP melhorou as soluções encontradas pela versão seqüencial [1]: circ10, circ16, circ18, nl16, e br24 (instância gerada a partir do Campeonato Brasileiro de Futebol de 2003). O critério de parada usado neste experimento foi, para cada instância, o valor alvo mostrado na segunda coluna da Tabela 1. Todos os tempos são relatados em segundos e representam a média aritmética sobre cinco execuções, para cada instância.

Tabela 1. PAR-MP e AUDImTTP em ambientes LAN e WAN de 10 processadores.

		PAR-MP (segundos)			AUDImTTP (segundos)		
Instância	Alvo	LAN	WAN	deg. %	LAN	WAN	deg.%
circ10	274	629,43	747,52	18,76	252,26	257,95	2,26
circ16	984	3.744,72	4.537,74	21,18	515,86	521,49	1,09
circ18	1310	6.703,00	7.532,03	12,37	3.482,06	3.516,47	0,99
nl16	280117	3.894,20	4.500,76	15,58	2.685,08	2.723,68	1,44
br24	503158	4.905,63	5.106,51	4,09	1.129,62	1.190,35	5,38
			Média	14,04			2,01

A Tabela 1 destaca as melhorias obtidas com o AUDImTTP em 10 processadores, em relação ao PAR-MP com o mesmo número de processadores. Neste experimento, os resultados LAN referem-se à execução das implementações paralelas em um *cluster* local, e os resultados WAN à execução em dois *clusters* distribuídos geograficamente. Na Tabela 1, para cada algoritmo, é mostrado também a porcentagem de degradação causado pela execução do processo mestre (no caso de PAR-MP) e o gerenciador do *pool* de cooperação (no caso de AUDImTTP) em sítios diferente dos processos PS. AUDImTTP é menos afetado pela latência entre os sítios. Comparando os resultados LAN de AUDImTTP e PAR-MP, as vantagens da estratégia proposta são claras até mesmo para ambientes que favoreçam o paradigma mestre-trabalhador, onde AUDImTTP fica 3,5 vezes mais rápida na média.

Como o AUDImTTP melhora o desempenho por distribuir processos provedores de solução entre os recursos de múltiplos sítios, é importante avaliar a sua escalabilidade. A Tabela 2 apresenta os tempos de execução médio em segundos dos algoritmos PAR-MP e AUDImTTP em 60 processadores, para as mesmas instâncias da Tabela 1. O algoritmo AUDImTTP foi executado sob três diferentes configurações, com três, cinco e sete *pools* de intensificação, respectivamente.

Tabela 2. Resultados de PAR-MP e AUDImTTP em 60 processadores.

	AUDImTTP			PAR-MP / AUDImTTP		
PAR-MP	3 *pools*	5 *pools*	7 *pools*	3 *pools*	5 *pools*	7 *pools*
72,11	36,84	31,02	32,57	1,96	2,32	2,21
980,66	345,03	319,22	321,54	2,84	3,07	3,05
5.647,42	1.822,45	1.783,69	1.801,76	3,10	3,17	3,13
2.759,23	1.975,99	1.678,51	1.708,97	1,40	1,64	1,61
1.244,28	493,82	374,01	390,68	2,52	3,33	3,18
			Média	2,36	2,71	2,64

Como pode ser observado na Tabela 2, todas as execuções da implementação AUDImTTP tiveram um tempo de processamento menor do que a execução de PAR-MP nas 60 máquinas. Dentre as três configurações de *pools* testadas com a implementação AUDImTTP, a execução com cinco *pools* de intensificação teve o melhor desempenho, alcançando uma redução média no tempo de processamento equivalente a 2,71 em relação a versão PAR-MP. O tempo computacional médio da execução com sete *pools* foi levemente pior do que com cinco, apresentando uma redução média do tempo igual a 2,64. Os resultados mostraram que há um ponto de saturação no número de *pools* de intensificação que deve haver em cada execução. Esse número é dependente da aplicação e do número de máquinas disponível na plataforma. Para um número fixo de recursos, mais gerenciadores de *pools* de intensificação implicam em menos recursos para provedoras de solução.

5.2 Experimentos com o AUDI-AGMD

Para os testes realizados com o AUDI-AGMD foram usadas algumas instâncias de grafos completos, provenientes da OR-Library [3]. Para os experimentos mostrados na Tabela 3, foram usados as cinco primeiras instâncias dos grupos de 70, 100 e 250 nós. O diâmetro máximo foi definido em 7, 10 e 15 arestas, respectivamente.

Na Tabela 3, para cada instância, é mostrado na primeira coluna o número de nós, na segunda o número de arcos. Em seguida, são relatados o diâmetro máximo especificado (D) para cada instância, e o custo (S^*) da melhor solução conhecida na literatura. Nas duas próximas colunas são destacados os custos das melhores soluções obtidas pela versão paralela AUDI-AGMD em 10 processadores e o tempo de execução (em segundos) necessário para atingir esses valores. As duas últimas colunas mostram o custo das soluções alcançadas pela heurística seqüencial e o tempo máximo dado para que ela pudesse obter as mesmas soluções da versão paralela.

Comparando os resultados da versão AUDI-AGMD com os resultados da heurística seqüencial, as vantagens da estratégia proposta são claras. Para todos os casos, a aplicação paralela melhorou os resultados da versão seqüencial, com exceção das duas primeiras instâncias de 70 nós, cujos resultados foram os mesmos. Além disso, a versão AUDI-AGMD foi capaz de melhorar as melhores soluções da literatura para cinco instâncias, as quais são destacadas em negrito na Tabela 3.

Tabela 3. Resultados de AUDI-AGMD em 10 processadores e da heurística seqüencial.

				AUDI-AGMD		Seqüencial	
$\|V\|$	$\|E\|$	D	S^*	melhor	tempo (s)	melhor	tempo(s)
70	2415	7	7,228	7,228	21,93	7,228	220,00
			7,080	7,080	16,27	7,080	160,00
			6,983	**6,981**	23,14	6,983	230,00
			7,499	**7,486**	305,67	7,499	3.000,00
			7,245	**7,238**	68,89	7,245	690,00
100	4950	10	7,759	**7,757**	139,11	7,835	1.400,00
			7,849	7,849	23.965,71	7,943	48.000,00
			7,904	7,930	504,045	7,961	5.000,00
			7,977	**7,973**	8.704,78	8,048	17.500,00
			8,164	8,176	336,92	8,204	3.400,00
250	31125	15	12,231	12,283	11.356,87	12,446	22.700,00
			12,016	12,123	8.978,35	12,307	18.000,00
			12,004	**11,999**	6.006,41	12,132	12.000,00
			12,462	12,472	11.373,14	12,700	22.800,00
			12,233	12,272	9.261,90	12,444	18.600,00

Para os experimentos mostrados na Tabela 4, foi escolhido uma instância de cada grupo de 50, 70, 100, 250, 500 e 1000 nós da OR-Library. Nessa tabela é mostrada, para cada instância selecionada, a quantidade $|V|$ de vértices, a quantidade $|E|$ de arestas e o valor D do diâmetro. Na quarta coluna é apresentado o valor alvo usado como critério de parada para cada instância. As três últimas colunas apresentam o tempo médio em segundos sobre cinco execuções do AUDI-AGMD em 15, 30 e 60 máquinas.

Tabela 4. Resultados do AUDI-AGMD em 15, 30 e 60 processadores do Grid Sinergia.

				AUDI-AGMD		
$\|V\|$	$\|E\|$	D	Alvo	15 CPUs	30 CPUs	60 CPUs
50	1225	5	7,599	1,96	3,72	10,65
70	2415	7	6,983	9,34	8,89	7,16
100	4950	10	7,981	49,23	35,11	24,08
250	31125	15	12,450	3.723,27	2.344,49	1.317,78
500	124750	20	17,063	2.627,14	2.496,45	2.006,08
1000	499500	25	24,609	3.837,79	3.401,71	2.916,25

Como pode ser visto na Tabela 4, a implementação paralela AUDI-AGMD é escalável. A redução no tempo de processamento ocorreu de acordo com o aumento no número de máquinas disponíveis para execução. Isso significa que foi possível aproveitar a capacidade de processamento disponível através do aumento no número de

máquinas. No entanto, para a instância com 50 vértices, 1.225 arestas e diâmetro cinco (primeira linha da Tabela 4) essa redução no tempo de processamento não ocorreu, pois essa instância muito rapidamente converge para um ótimo local. Conseqüentemente, o tempo necessário para criar novos processos em um número razoável de máquinas e gerenciálos é maior do que o tempo de execução em um número menor de máquinas. Isso significa que, nesse caso o aumento no poder computacional não se faz mais necessário, pois a instância atingiu um ponto de saturação em relação a quantidade de recursos. Para as instâncias com uma carga de processamento maior, o aproveitamento de um maior número de recursos disponíveis na grade torna-se evidente.

6 Conclusão

Este trabalho abordou o desenvolvimento e a execução eficiente de metaheurísticas paralelas em ambientes de grade. Foram propostos uma estratégia original de paralelização e o *middleware* de gerência metaEasyGrid. A integração entre os dois viabilizou a construção de metaheurísticas paralelas autonômicas para grades.

A estratégia hierárquica distribuída mostrou ser eficaz no desenvolvimento de metaheurísticas paralelas porque proporciona cooperação entre instâncias de metaheurísticas seqüências, sem acarretar considerável sobrecarga no desempenho. A cooperação ocorre independentemente de onde os processos estejam sendo executados.

Outra característica da estrutura hierárquica distribuída é a sua capacidade de naturalmente proporcionar intensificação e diversificação na execução das metaheurísticas. A intensificação ocorre devido ao uso de vários *pools* de soluções de elite associados aos sítios do ambiente. Por outro lado, a diversificação é garantida através do *pool* de cooperação que mantém sempre as melhores soluções encontradas por todos os sítios envolvidos na execução.

Além disso, a simplicidade de implementação da estratégia hierárquica distribuída em grades foi possível devido à sua integração com o metaEasyGrid. Assim, ambas as implementações de excelentes metaheurísticas seqüências foram capazes de aproveitar o poder computacional oferecido na grade, sem que o desenvolvedor tivesse que se deter na gerência do ambiente. Esse melhor desempenho foi refletido na qualidade das soluções encontradas, pois as duas implementações foram capazes de melhorar algumas das melhores soluções da literatura.

Referências

[1] A. Araújo, S. Urrutia, C. Boeres, V. Rebello, and C. Ribeiro. Towards grid implementations of metaheuristics for hard combinatorial optimization problems. In C. Amorim, G. Silva, V. Rebello, and J. Dongarra, editors, *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages 19–26, Rio de Janeiro, 2005. IEEE Press.

[2] A. P. F. Araújo. *Paralelização Autonômica de Metaheurísticas em Ambientes de Grid*. PhD thesis, Departamento de Informática, PUC-Rio, 2008.

[3] J. Beasley. Welcome to OR-Library. Disponível em http://people.brunel.ac.uk/mastjjb/jeb/info.html, última visita em 20 de Maio de 2008.

[4] C. Boeres, A. Sena, A. Nascimento, J. Silva, D. Q. Vianna, and V. Rebello. On the advantages of an alternative MPI execution model for the grids. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, pages 575–582, Rio de Janeiro, 2007. IEEE Press.

[5] V.-D. Cung, S. Martins, C. Ribeiro, and C. Roucairol. Strategies for the parallel implementation of metaheuristics. In C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 263–308. Kluwer Academic Publishers, 2002.

[6] K. Easton, G. Nemhauser, and M. Trick. The traveling tournament problem: Description and benchmarks. In T. Walsh, editor, *Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 580–589. Springer, 2001.

[7] K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In E. Burke and P. Causmaecker, editors, *Selected Papers from the 4th International Conference on the Practice and Theory of Automated Timetabling*, volume 2740 of *Lecture Notes in Computer Science*, pages 100–109. Springer, 2003.

[8] I. Foster. The grid: A new infrastructure for 21st century science. *Physics Today*, 55:42–47, 2002.

[9] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[10] L. Gouveia and T. L. Magnanti. Network flow models for designing diameter-constrained minimum-spanning and Steiner trees. *Networks*, 41:159–173, 2003.

[11] M. Gruber and G. Raidl. Variable neighborhood search for the bounded diameter minimum spanning tree problem. In *18th Mini Euro Conference on Variable Neighborhood Search*, pages 1–11, Tenerife, 2005.

[12] H. Lourenço, O. Martins, and T. Stutzle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, 2003.

[13] M. Resende and C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.

[14] C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 179:775–787, 2007.

[15] A. Santos. *Modelos e Algoritmos para o Problema da Árvore Geradora de Custo Mínimo com Restrição de Diâmetro*. PhD thesis, Departamento de Informática, PUC-Rio, 2006.