Algoritmo *Branch-and-Bound* Distribuído e Tolerante a Falhas para um Ambiente em Grade

Alexandre Gonçalves[1], Lúcia Drummond[1], Eduardo Uchoa[2], Maria Clicia S. de Castro[3]
[1]*Departamento de Ciência da Computação – UFF*
[2]*Departamento de Engenharia de Produção – UFF*
[3]*Departamento de Informática Ciência da Computação – UERJ*
{agonçalves,lucia}@dcc.ic.uff.br, uchoa@producao.uff.br, clicia@ime.uerj.br

Abstract

This work introduces a new fault tolerant and distributed branch-and-bound algorithm, applied to the Steiner Problem in Graphs (SPG), to be run on computational Grids. Many Grids are composed of cluster of processors connected via high-speed links and the clusters, geographically distant, are connected through low-speed links, in a hierarchical fashion. The algorithm proposed has the following features: i) it does not employ the usual master-worker paradigm; ii) it considers the hierarchical structure of such Grids in its procedures; and iii) it contains load balance and fault tolerance mechanisms. Good speepuds were obtained, allowing the resolution of hard instances in very reasonable times.

1. Introdução

A técnica *branch-and-bound* tem sido uma das mais utilizadas para encontrar a solução ótima de problemas de otimização NP-difíceis. Os algoritmos *branch-and-bound* percorrem o espaço de soluções através de árvores de enumeração. Eles possuem um grande potencial de paralelização porque as computações das sub-árvores podem ser realizadas de modo quase independente.

Este trabalho propõe um novo algoritmo *branch-and-bound* distribuído e tolerante a falhas, aplicado ao Problema de Steiner em Grafos (SPG), para grades computacionais. Em geral, as grades computacionais são compostas por diversos *clusters* de processadores, geograficamente distantes, conectadas por canais de baixa velocidade. Entretanto, existe uma estrutura hierárquica nas grades, considerando que a comunicação entre processadores num mesmo *cluster* é muito mais rápida do que entre processadores de diferentes *clusters*. Dessa forma, propomos um algoritmo *branch-and-bound* distribuído cujos procedimentos exploram esta estrutura hierárquica das grades.

Vários trabalhos recentes abordam algoritmos *branch-and-bound* paralelos para ambientes em grade [11]. Muitos deles adotam uma abordagem centralizada, na qual um único processador mantém a árvore e envia tarefas para os demais processos.

Segundo Aida *et al* [1] esta estratégia não é escalável e nem conveniente para ambientes em grade. Pode ocorrer uma degradação significativa no desempenho por causa do alto *overhead* de comunicação entre os processos. Eles propõem um algoritmo *branch-and-bound* distribuído, baseado em um paradigma mestre-escravo hierárquico. Um processo supervisor controla a ativação de múltiplos processos, cada um dos quais é composto de um mestre e vários escravos. Essa estratégia não é completamente distribuída e não aborda tolerância à falhas.

O controle central pode ser também um obstáculo à tolerância à falhas. Nesse contexto, Iamnitch e Foster [5] propõem um algoritmo *branch-and-bound* completamente distribuído, e com mecanismos de tolerância à falhas. Esse algoritmo é baseado em tabelas mantidas em todos os processadores, contendo informações de sub-árvores resolvidas, e em mensagens usadas para propagar as tabelas. A recuperação de falhas é atingida quando um processo sem trabalho verifica sua tabela local e escolhe uma sub-árvore não resolvida para solucioná-la. Essa solução pode gerar muito trabalho redundante, ainda que não ocorram falhas.

Em [2] alguns melhoramentos foram propostos no modelo original, visando principalmente a estratégia de balanceamento de carga. Essa estratégia foi aplicada para a solução de instâncias difíceis do Quadratic Assignment Problem.

Resumindo, nosso algoritmo difere dos anteriores nos seguintes pontos: i) não utiliza o paradigma usual mestre-escravo; ii) considera a estrutura hierárquica das grades nos seus procedimentos principais; e iii) inclui mecanismos de balanceamento de carga e tolerância à falhas. Desenvolvemos um algoritmo distribuído baseado em um algoritmo *branch-and-bound* seqüencial, já existente para SPG, que é um problema NP-difícil clássico. De um conjunto de 400 instâncias SPG difíceis, propostas por Duin [3] como um *benchmark* e um desafio para algoritmos futuros, o algoritmo seqüencial foi capaz de solucionar quase todas as instâncias com tempos razoavelmente pequenos. As instâncias estão disponíveis no

repositório SteinLib [7], e somente 20 instâncias incidentes estão ainda em aberto. Um dos objetivos do algoritmo *branch-and-bound* distribuído é solucionar estas instâncias restantes. Este artigo está organizado da seguinte forma. As Seções 2 e 3 descrevem os algoritmos *branch-and-bound* seqüencial e distribuído, respectivamente. O ambiente experimental, os resultados preliminares e trabalhos futuros são apresentados na Seção 4.

2. *Branch-and-Bound* Seqüencial para SPG

Um algoritmo *branch-and-bound* soluciona um problema de otimização realizando operações de ramificação que dividem um problema em dois subproblemas similares. Cada subproblema (nó) trabalha com uma diferente parte do espaço de solução. A solução dos subproblemas, também, inclui operações de ramificação. Assim, o algoritmo induz a uma árvore de enumeração. O cálculo dos limites para o valor da solução ótima é uma parte fundamental de um algoritmo *branch-and-bound*. Os limites são usados para limitar o crescimento da árvore.

O SPG é um dos problemas NP-difíceis mais estudados e pode ser computacionalmente muito custoso. Wong [14] propôs um algoritmo denominado *dual ascent*, que rapidamente encontra uma solução aproximada. Soluções aproximativas, também, fornecem limites inferiores válidos. Poggi de Aragão *et al* [9] propuseram três heurísticas adicionais para melhorar os limites do *dual ascent*, denominadas: *dual scaling*, *dual adjustment* e *active fixing by reduced costs*. O algoritmo *branch-and-bound* que utilizamos é baseado nestas heurísticas, e está completamente descrito em Uchoa [13]. Naquela época, ele pôde solucionar cerca de 60 instâncias incidentes pela primeira vez, a maior com 640 vértices e 200.000 arestas. Somente 20 instâncias incidentes da SteinLib não puderam ser solucionadas em tempo satisfatório (menos de um dia em um PC de 350 MHz). Com as máquinas atuais mais rápidas, o mesmo algoritmo poderia solucionar 5 dessas instâncias incidentes, em aberto, em no máximo uma semana de tempo de CPU.

3. *Branch-and-Bound* Distribuído para SPG

O algoritmo *branch-and-bound* possui um grande potencial de paralelização porque as computações das sub-árvores podem ser realizadas de modo quase independente. O algoritmo proposto é composto pelos seguintes procedimentos: distribuição inicial, balanceamento de carga, difusão do primal, detecção de terminação e tolerância a falhas, que estão descritos a seguir. Certos procedimentos necessitam de um líder em cada *cluster*. O processo líder é aquele que possui o menor número de identificação.

Note que assumimos que apenas um processo é atribuído a cada processador. A partir desse ponto usamos indistintamente essas duas palavras no restante do texto.

3.1 Distribuição Inicial

A fase de distribuição inicial corresponde à distribuição de sub-árvores entre os *clusters*. Em cada operação de ramificação, um líder de *cluster x* envia uma mensagem com um novo nó para outro *cluster*. Esse *cluster* é definido como $nextcluster = x + 2^{level}$. Onde *level* representa a profundidade do líder *x* na árvore. Quando *nextcluster* atinge um valor maior que o número de *clusters* disponíveis na grade, os processos iniciam o compartilhamento de suas cargas locais com outros processos dentro do mesmo *cluster*. Mais especificamente, *level* é reiniciado com zero, e cada processo *i* envia um novo nó para o processo $nextproc = i + 2^{level}$. Por exemplo, considere um *cluster* com 8 processadores. Identificamos cada processo como $cl_0,...,cl_7$. Inicialmente, cl_0 envia metade de sua carga local para cl_1, e continua processando a outra metade de sua árvore. No segundo nível da árvore, cl_0 compartilha sua carga agora com cl_2, enquanto cl_1 compartilha sua carga com cl_3, e assim por diante. A Figura 1 mostra a árvore resultante desta fase.

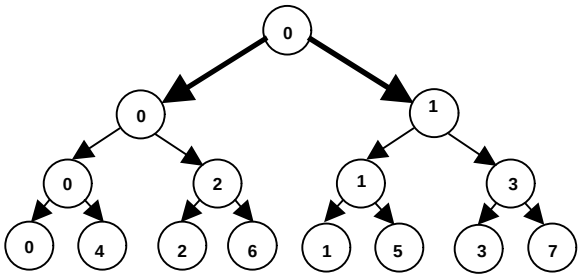

Figura 1. Exemplo de distribuição de carga inicial entre *clusters*

3.2 Balanceamento de carga

O tempo para processar uma sub-árvore não é conhecido *a priori* e pode variar muito, o que torna a utilização de um algoritmo de balanceamento de carga fundamental para este problema. Neste trabalho implementamos três diferentes algoritmos para o balanceamento de carga, denominados: GlobalLB; LocalLB e HybriLB.

No algoritmo GlobalLB, sempre que um processo se torna ocioso, este pede carga (sub-árvore) para qualquer processo, não havendo distinção entre processos de diferentes *clusters*.

A Figura 2 ilustra o algoritmo GlobalLB.

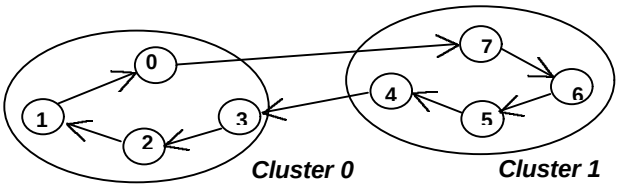

Figura 2. Exemplo de pedido de carga no algoritmo GlobalLB

No algoritmo LocalLB, é considerado somente o envio de carga entre processos pertencentes ao mesmo *cluster*.

O HybridLB é um algoritmo híbrido que prioriza o balanceamento de carga entre processos de um mesmo *cluster*, mas permite também que haja transferência de carga entre *clusters* diferentes, quando todos os processos dentro de um *cluster* se tornam ociosos.

As Figuras 3 e 4 mostram a comunicação, para o algoritmo HybridLB, com os pedidos de carga dentro de um *cluster* e entre *clusters*, respectivamente.

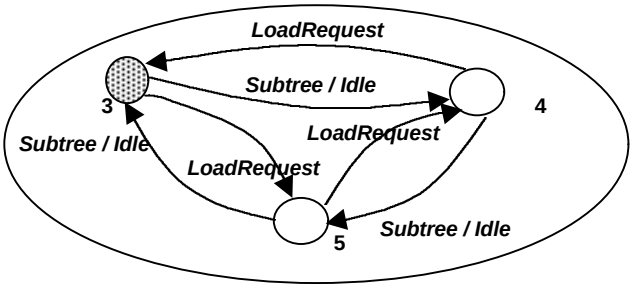

Figura 3. Exemplo de pedido de carga dentro de um *cluster* no algoritmo HybridLB

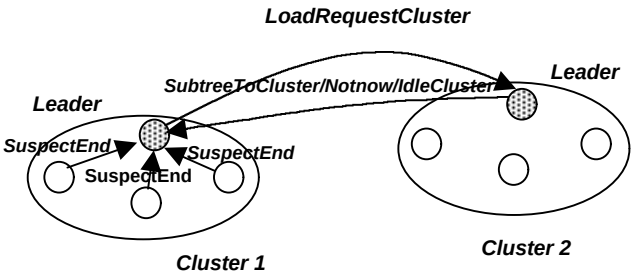

Figura 4. Exemplo de pedido de carga entre clusters no algoritmo HybridLB

3.3 Difusão do primal e Detecção de terminação

Em ambos os procedimentos propomos um algoritmo hierárquico. O procedimento de difusão do primal é invocado quando um processo encontra um limite melhor do que o limite atual conhecido. Ele deve ser disseminado entre os processadores através da grade, permitindo a poda da árvore de enumeração. Um processo envia o limite do primal para os processos do *cluster* a que pertence, e o líder do *cluster* é responsável por enviar o limite do primal para os outros líderes dos *clusters* da grade, que o difundem entre seus processadores.

A Figura 5 ilustra a difusão do limite do primal para os processos dentro de um *cluster*. O seu líder, ao receber a mensagem é o responsável pela disseminação entre os outros líderes.

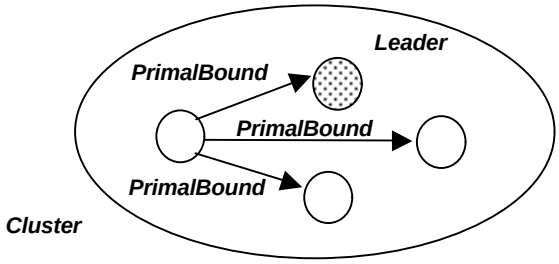

Figura 5. Exemplo de difusão do limite do primal entre os processos de um *cluster*

Quando um processo atinge sua condição de terminação local, isto é, não é capaz de obter sub-árvores dos seus vizinhos, ele informa o fato ao seu líder. Quando todos os processos daquele *cluster* atingirem a mesma condição, e o líder não for capaz de obter sub-árvores de outros líderes, considera-se que o *cluster* atingiu sua condição de terminação. Assim, o líder do *cluster* envia uma mensagem aos outros líderes indicando a condição de terminação e pode terminar (assim como os processos do seu *cluster*) após o receber a mesma mensagem de todos os outros líderes. A Figura 6 exemplifica o procedimento de detecção e terminação, quando é atingida a mesma condição por todos os processos.

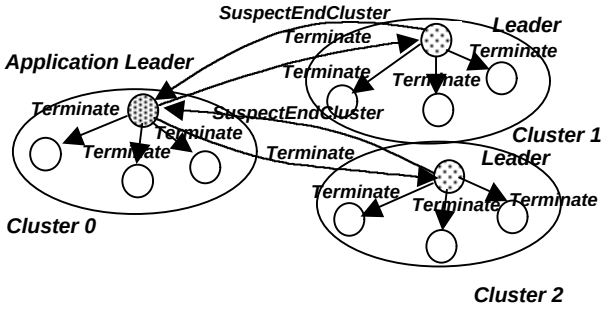

Figura 6. Exemplo de detecção e terminação de processos

3.4 Tolerância a falhas

Os procedimentos de tolerância a falhas devem permitir que uma aplicação continue a sua execução

mesmo na presença de falhas. Usualmente, essa meta é atingida pela utilização de técnicas de *checkpoint* e *rollback recovery*, Elnozazhy *et al.* [4].

Em nosso algoritmo cada processo é executado independentemente. Por isso, optamos por *checkpoints* não coordenados, onde cada processo decide independentemente quando gravar o seu *checkpoint*. Nossa estratégia é capaz de tratar uma única falha permanente por *cluster*.

Neste procedimento, um processo *i* periodicamente envia uma mensagem contendo seu *checkpoint* para outro processo vizinho, denominado sucessor, no mesmo *cluster*, $j = (i - 1) \bmod (cs_x) + id_{clx}$. Após receber uma mensagem de *checkpoint*, o processo vizinho armazena o último *checkpoint* recebido.

O *checkpoint* é composto de informação referente ao último nó executado e um vetor cuja dimensão é igual ao nível deste nó na árvore. Cada posição *i* do vetor indica se uma mensagem de ramificação, referente à sub-árvore de nível *i*, foi enviada para ser resolvida em outro processo (se não foi enviada, precisará ser resolvida futuramente).

Durante a recuperação, o processo com o *checkpoint* do processo que falhou assume os serviços que lhe foram atribuídos. A detecção de falhas é baseada no mecanismo *heartbeat*, onde processos trocam mensagens em intervalos de tempo regulares indicando que eles estão ativos [10].

Esta estratégia é estendida de modo a detectar falha de *cluster*, substituindo processadores por *clusters* (representados pelo seu líder), tendo *checkpoints* representando estados de *clusters* (ao invés de processos) e empregando o mecanismo de *heartbeat* entre líderes de *clusters*.

A Figura 7 ilustra a comunicação no procedimento de tolerância a falhas.

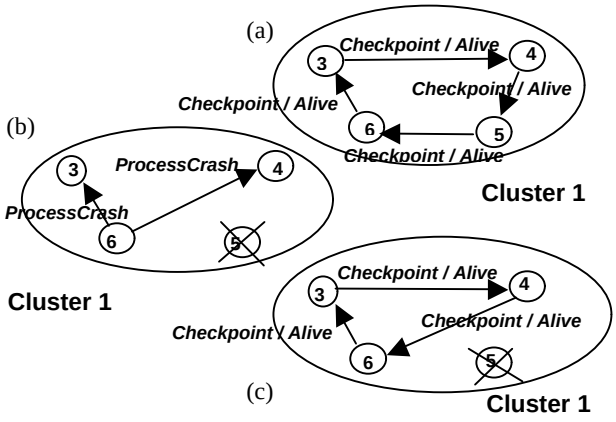

Figura 7. Exemplo de comunicação de no procedimento de tolerância a falhas

A figura exemplifica o procedimento de tolerância a falhas dentro de um único *cluster* com quatro processos, cujos números de identificação são 3, 4, 5 e 6.

Na figura 7 (a) cada processo envia aos seus sucessores, em intervalos regulares, uma mensagem de *Checkpoint*, para ser gravada, ou uma mensagem *Alive*, que informa a ausência de falhas no processador (mecanismo *heartbeat*).

Na Figura 7 (b) supomos a ocorrência de falha no processador 5. Um processador considera que seu predecessor falhou após alguns segundos sem receber qualquer mensagem dele. Dessa forma, o processador 6, após detectar a falha, envia uma mensagem de *ProcessCrash* para os outros processadores do mesmo *cluster* (processadores 3 e 4). Além disso, o processador 6 assume os serviços que lhe foram atribuídos pelo último *checkpoint* enviado pelo processador 5.

Podemos observar na Figura 7 (c) que, depois do procedimento de recuperação, os processadores restantes 3, 4 e 6 continuam com as trocas de mensagens, dando continuidade a execução da aplicação.

4. Resultados Preliminares

Nosso algoritmo está desenvolvido em C++ e usa MPICH-G2, uma versão da biblioteca MPI para o Globus, Foster e Kessekman [6] e Snir *et al.* [12].

Os resultados vem sendo obtidos no ambiente GridRio, que é uma iniciativa para criar uma grade computacional através de cinco universidades do Estado do Rio de Janeiro. Como a GridRio ainda não está totalmente operacional, nossos experimentos preliminares foram executados em uma pequena grade composta de *clusters* de somente duas universidades: PUC-Rio (com 16 processadores Pentium de 1.7GHz) e UFF (com 10 processadores Athlon de 1.3GHz). O algoritmo distribuído foi analisado para 5 instâncias incidentes, particularmente difíceis de um conjunto de 20 instâncias SPG em aberto, onde o algoritmo seqüencial gastaria cerca de uma semana de tempo de CPU.

Utilizamos em nossos experimentos, da biblioteca SteinLib, as instâncias incidentes de i640-211 a i640-215. Elas possuem 640 vértices, 4135 arcos e 50 terminais.

Para analisar o balanceamento de carga proposto, executamos nosso algoritmo, inicialmente, no *cluster* da PUC com 15 processadores, para 5 instâncias. Obtivemos os seguintes resultados (instância/valor ótimo): i640-211/11984; i640-212/11795; i640-213/11879; i6402-14/11898; e i640-215/12081.

Os *speedups* médios do algoritmo foram 13.5 e 3.3, quando comparado com a versão seqüencial e a versão distribuída sem balanceamento de carga, respectivamente.

Outros testes foram realizados usando os *clusters* da PUC e da UFF. Executamos o algoritmo distribuído com as três versões de balanceamento de carga:

GlobalLB, LocalLB e HybridLB. Observamos que o algoritmo LocalLB apresentou tempos de execução médios 41% piores do que GlobalLB, e o algoritmo HybridLB é ligeiramente melhor (9%) do que GlobalLB.

Os *speedups* obtidos foram bons, porém apresentaram irregularidade. Por exemplo, para a instância i640-214 o *speedup* foi quase duas vezes o número de processos. Essa anomalia nos algoritmos *branch-and-bound* paralelos são usuais, Lai e Sahni [8] e Bruin *et al*. [2]. Se uma solução é, por chance, descoberta cedo, as sub-árvores podem ser melhores podadas e os tempos de execução são muito reduzidos. O oposto, também, é verdade.

O procedimento de tolerância à falhas tem apresentado funcionamento correto nas simulações de falhas. O custo adicional no tempo de execução da aplicação com a inclusão deste procedimento, quando não ocorrem falhas, não ultrapassou 8,3% do tempo de execução da aplicação sem a inclusão do mesmo.

5. Considerações Finais e Trabalhos Futuros

Neste trabalho propusemos um algoritmo *branch-and-bound* distribuído e tolerante à falhas. Ele difere dos trabalhos encontrados na literatura porque não utiliza o paradigma usual mestre-escravo, considera a estrutura hierárquica das grades nos seus procedimentos principais e inclui mecanismos de balanceamento de carga e tolerância à falhas.

Nosso algoritmo *branch-and-bound* distribuído foi desenvolvido baseado em um algoritmo seqüencial, já existente para SPG [13], que utiliza três heurísticas denominadas: *dual scaling*, *dual adjustment* e *active fixing by reduced costs*.

Os resultados preliminares obtidos demonstram que os procedimentos propostos possuem um grande potencial para explorar a hierarquia dos ambientes de grades computacionais.

O desempenho, na maioria dos experimentos realizados, melhora quando aumentamos o número de processadores o que demonstra que os procedimentos propostos tornam o algoritmo escalável.

Outros pontos importantes que observamos é que: i) o desempenho de nosso algoritmo é muito dependente do tamanho das sub-árvores recebidas por cada processo, e ii) sub-árvores grandes minimizam os tempos ociosos.

Como trabalhos futuros pretendemos realizar testes mais detalhados para melhor analisar os procedimentos propostos, principalmente o de balanceamento de carga e o *overhead* introduzido pelo procedimento de tolerância à falhas.

Para isso, pretendemos utilizar as 15 instâncias incidentes em aberto restantes no ambiente GridRio. Estas instâncias, pertencentes à biblioteca SteinLib, necessitam de muito maior poder computacional e se beneficiara dos nossos procedimentos. Serão avaliados o impacto dos procedimentos propostos de balanceamento de carga, primal, difusão e detecção de terminação que consideram a estrutura hierárquica da grade no desempenhoiados.

Nossa percepção indica que se melhorarmos os procedimentos de distribuição inicial e balanceamento de carga poderemos obter uma melhor adequação entre os sub-problemas e os *clusters*. Atingindo esse objetivo esperamos melhorar significativamente os resultados atuais.

Um outro ponto extremamente importante, quando se utiliza um ambiente em grade, que pretendemos realizar é verificar a confiabilidade do procedimento de tolerância à falhas, além de estendê-lo para admitir duas ou mais falhas por *cluster*.

12. Referências

[1] K. Aida, W. Natsume, Y. Futakata, "Ditributed Computing with Hierarchical Master-Worker Paradigm for Parallel Branh-and-Bound Algorithms", *In the Proceedings of the Third International Symposium on Cluster Computing and Grid*, 2003, p. 156-162.

[2] A. Bruin, G.A.P. Kindervater, H.W.J.M. Trienekens, "Asynchronous Parallel Branh-and-Bound and Anomalies", Erasmus University, Department of Computer Science, EUR-CS-95-05, Roterdam, Holand, 1995.

[3] C. Duin, C. "Steiner´s Problems in Graphs", Ph.D. thesis, University of Amsterdam, Holand, 1993.

[4] E. Elnozazhy, D. Johnson, Y. e Wang, "A Survey of Rollback-Recovey Protocols in Message Passing Systems", Technical Report, CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[5] A. Iamnitch, I. Foster, "A Problem Specific Fault-Tolerance Mechanism for Asynchronous, Ditributed System", *In the Proceedings of the International Conference on Parallel Processing*, 2000, p. 4-14.

[6] I. Foster, C. Kessekman, "The Globus Project: a Status Report", *In the Proceedings of the Seventh Heterogeneous Computing Workshop (HCW´98)*, 1998, p. 4-18.

[7] T. Koch, A. Martin, S. Voss, "SteinLib: An Update Library on Steiner Problems in Graphs", Konrad-Zuse-Zentrum für Informationstechnik Berlin, ZIB-Report 00-37, 2003, http://elib.zib.de/steinlib, last visited Oct 30th, 2003.

[8] T.-H. Lai, S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms", *Communication of the ACM*, 27, 1984, p. 594-602.

[9] M. Poggi de Aragão, E. Uchoa, R.S. Wernwck, "Dual Heuristics on the Exact Solution of Large Steiner Problems", *Eletronic Notes in Discrete Mathematics*, 7, 2001.

[10] A. Robertson, "Linux-HA: Heartbeat System Design", *In the Proceedings of the Fourth Annual Linux Showcase and Conference (ASL)*, 2000.

[11] C. Roucairol, V. Cung, N. Yafoufi, "Design, Implementation and Robustness in Parallel Branch-and-Bound", Techinical Repórt PRISM 2000/19, l´Université de Versailles Saint-Quentin, 2000.

[12] M. Snir, S.M. Otto, S. Huss-Lederman, J. Dongarra, *MPI: The Complete Reference*, The MIT Press, 1996

[13] E. Uchoa, "Algoritmos para Problemas de Steiner com Aplicações em Projeto de Circuitos VLSI", Ph.D. thesis, Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, 2001.

[14] R. Wong, "Dual Ascent Approach for Steiner Tree Problems on a Discrete Graph", *Mathematical Programming*, 28, 1984, p. 271-287.