Um algoritmo branch-and-bound distribuído para o Problema de Steiner em Grafos para execução em Grids

Alexandre Domingues Gonçalves

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

Orientadora: Lúcia Maria de A. Drummond

Niterói, Junho de 2005

Um algoritmo branch-and-bound distribuído para o Problema de Steiner em Grafos para execução em Grids

Alexandre Domingues Gonçalves

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

Aprovada por:

_____

Profa. Lúcia Maria de A. Drummond / IC-UFF (Presidente)

_____

Prof. Bruno Richard Schulze / LNCC

_____

Prof. Eduardo Uchoa Barboza / Eng. de Produção – UFF

_____

Prof. Valmir Carneiro Barbosa / Coppe/Sistemas – UFRJ

_____

Prof. Wagner Meira Júnior / DCC - UFMG

Resumo da Tese apresentada à UFF como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação (M.Sc.)

Um algoritmo branch-and-bound distribuído para o Problema de Steiner em Grafos para execução em Grids

Alexandre Domingues Gonçalves
Jun/2005

Orientadora: Lúcia Maria de A. Drummond
Programa de Pós-Graduação em Computação

Esta dissertação apresenta um algoritmo *branch-and-bound* distribuído para execução em *Grids* computacionais. Geralmente as *grids* são organizadas de modo hierárquico: processadores de um mesmo cluster são conectados através de *links* de alta velocidade, enquanto os clusters são geograficamente distantes entre si e se conectam através de *links* de baixa velocidade.

O algoritmo apresentado não emprega o paradigma usual de mestre-escravo e leva em consideração a estrutura hierárquica das *Grids* no balanceamento de carga e procedimentos de tolerância a falhas. Este algoritmo foi aplicado sobre um código existente para resolver o Problema de Steiner em Grafos. Experimentos em condições reais de *Grid* demonstraram sua eficiência e escalabilidade.

Abstract of Thesis presented to UFF as a partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

A distributed branch-and-bound algorithm for the Steiner Problem in Graphs to be run on computational Grids

Alexandre Domingues Gonçalves
Jun/2005

Advisor: Lúcia Maria de A. Drummond
Departament: Computer Science

This work introduces a distributed branch-and-bound algorithm to be executed on computational Grids. Grids are often organized in a hierarchical fashion: clusters of processors connected via high-speed links, while the clusters themselves are geographically distant and connected through slower links.

The algorithm does not employ the usual master-worker paradigm and it considers the hierarchical structure of Grids in its load balance and fault tolerance procedures. This algorithm was applied over an existing code for the Steiner Problem in graphs. Experiments on real Grid conditions have demonstrated its efficiency and scalability.

Sumário

Lista de tabelas

Lista de figuras

Capítulo 1

Introdução

Branch-and-bound é uma técnica largamente utilizada para solução de problemas de otimização do tipo NP-difícil. Tal algoritmo percorre um espaço de soluções através de enumeração de árvores. O processamento de cada sub-árvore pode ser resolvido quase independentemente, portanto, este algoritmo é bem adaptável ao paralelismo.

Este trabalho apresenta um algoritmo *branch-and-bound* distribuído a ser executado em uma *WAN (wide-area network)* provida de uma infra-estrutura que permite a execução de aplicações paralelas que requerem uma grande quantidade de poder computacional, chamada de *Grid* [14]. Foi observado que, apesar do "espírito de grande área", *Grids* típicas contêm *clusters* de processadores fisicamente conectados entre si através de *links* de alta velocidade. Portanto, há uma estrutura hierárquica nas *Grids*, considerando que a comunicação entre os processadores de um mesmo *cluster* é muito mais rápida do que entre dois processadores distantes.

Uma segunda observação é que processadores individuais e *links* em tais ambientes são muito suscetíveis a falhas. Um alto grau de tolerância a falhas é essencial

em aplicações em *Grids*. Caso contrário, a ocorrência de falha em um processador ou *link* tornaria todo o processamento não seguro.

Existem diversos artigos recentes sobre algoritmos *branch-and-bound* paralelos para *Grids*. Muitos adotaram uma abordagem centralizada, na qual um único processador mantém a árvore completa e distribui sub-árvores como tarefas aos escravos. Em *Goux et al*.[16], um *framework* baseado neste modelo mestre-escravo foi proposto. Falhas são também tratadas pelo mestre, que simplesmente redireciona as tarefas originalmente alocadas aos processos que tenham apresentado falhas. A falha do mestre é controlada com *checkpoints*. Em *Anstreicher et al*. [2] algumas melhoras foram propostas ao modelo previamente descrito em relação principalmente à estratégia de balanceamento de carga. Este algoritmo foi aplicado com sucesso para solução de instâncias do *Quadratic Assignment Problem.*

Embora a paralelização do tipo mestre-escravo seja conceitualmente mais simples, é notório o seu problema de escalabilidade. De acordo com *Aida, Natsume e Futakata* [1], isto pode ser particularmente mais grave em ambientes de *Grid*, onde a sobrecarga de comunicação entre processos mestre e escravos conectados por uma *WAN* é maior. Então, eles propuseram um *branch-and-bound* distribuído baseado em um algoritmo hierárquico de dois níveis, onde um processo supervisor controla um conjunto de múltiplos mestres. Esta estratégia não é totalmente distribuída e o problema de tolerância a falhas não foi discutido.

Iamnitchi e Foster [18] propuseram um algoritmo totalmente distribuído com mecanismos de tolerância a falhas a ser executado em *Grids*. Em seu algoritmo, cada processador guarda tabelas contendo informações sobre as sub-árvores já executadas, e mensagens são usadas para propagar estas tabelas. A recuperação da falha é realizada quando um processador livre seleciona uma sub-árvore não completada, consultando sua tabela local e a resolve. Esta estratégia pode resultar em uma grande quantidade de trabalho redundante, mesmo quando falhas não ocorrem.

O algoritmo *branch-and-bound* proposto difere dos anteriores e tem as seguintes características:

(i) não utiliza o paradigma mestre-escravo; muito pelo contrário, é totalmente distribuído,

(ii) considera a estrutura hierárquica natural de *Grids* em seus principais procedimentos, incluindo balanceamento de carga e tolerância a falhas, para reduzir a comunicação em *links* de baixa velocidade.

(iii) seu procedimento de tolerância a falhas, que permite que o algoritmo se recupere da falha de um simples processador, ou mesmo de um cluster inteiro (falha de *link*), foi projetado para evitar o máximo possível o trabalho redundante.

Este algoritmo distribuído foi implementado tomando como base um algoritmo *branch-and-bound* para solucionar o Problema de *Steiner em Grafos*, um clássico problema NP-difícil [9]. Vários experimentos com o *branch-and-bound* paralelo resultante foram feitos utilizando-se cinco instâncias da literatura obtidas a partir da Steinlib [20]. Para estas instâncias, o algoritmo seqüencial leva um tempo extenso para execução, chegando a até uma semana de tempo de CPU.

Nesta dissertação, é mostrado que o algoritmo proposto faz muito bom uso do potencial total dos processadores disponíveis em um ambiente real de *Grid,* sendo eficiente e escalável. É também observado que o mecanismo proposto de tolerância a falhas é realmente eficiente, permitindo a recuperação de uma falha sem impacto significativo no desempenho.

O restante deste trabalho está organizado como a seguir.

O Capítulo 2 descreve o Problema de Steiner em Grafos, um algoritmo de *branch-and-bound* seqüencial para a solução deste e as instâncias utilizadas como *benchmark* para os algoritmos propostos nesta dissertação.

O Capítulo 3 descreve o algoritmo distribuído, detalhando os procedimentos de: distribuição inicial de carga, balanceamento de carga, difusão da melhor solução encontrada e tolerância a falhas.

O Capítulo 4 apresenta os resultados obtidos em quatro etapas de testes. A primeira etapa consiste em testes em um único *cluster*. Na segunda etapa, são feitos testes em uma pequena *Grid* . Na terceira etapa são avaliados os procedimentos de tolerância a Falhas e na quarta etapa é avaliada a escalabilidade do algoritmo proposto.

No Capítulo 5, são apresentadas as conclusões e propostas de estudos futuros.

Capítulo 2

Um Algoritmo *Branch-and-Bound* Seqüencial para Solução do Problema de Steiner em Grafos

Um algoritmo de *branch-and-bound* resolve problemas de otimização utilizando operações de *branch* que dividem o problema original em dois subproblemas com espaços de solução diferenciados. Como a solução desses subproblemas podem também incluir operações de *branch*, o algoritmo induz uma enumeração de árvores onde cada nó representa um subproblema cuja solução é a melhor encontrada na sub-árvore enraizada a partir daquele nó. Para tentar evitar um crescimento excessivo da árvore de enumeração esse algoritmo também calcula em cada nó limites inferiores e superiores (*bounds*) para o valor da solução ótima.

O algoritmo seqüencial de *branch-and-bound* utilizado nesta dissertação foi elaborado com o objetivo de resolver problemas de Steiner em grafos, problemas da categoria NP-difícil e que em muitos casos necessitam de um grande poder computacional para serem resolvidos.

2.1 O Problema de Steiner em Grafos

O problema de Steiner em Grafos pode ser definido da seguinte forma:

> *Dado um grafo não-orientado G = (V,E) com custos positivos associados às arestas e um conjunto de terminais $T \subseteq V$, encontrar um subgrafo conexo de G de custo mínimo que contenha todos os vértices de T.*

Como os custos são positivos, a solução ótima será sempre uma árvore, chamada de árvore de Steiner mínima. Esse problema tem aplicações práticas em diversas áreas, tais como:

- Projeto de Rede: Árvores de Steiner representam em geral a solução de menor custo para o projeto de diversos tipos de rede, como telefônica, ferroviária ou de águas pluviais. O objetivo é sempre o mesmo: ligar da maneira mais eficiente possível o conjunto dos pontos que devem ser servidos (os terminais), utilizando outros pontos de bifurcação se necessário.

- Roteamento de Pacotes em Redes em *Multicast*: Nesse caso, os nós de grafo representam máquina e as arestas, os *links* entre elas. O custo de uma aresta é uma medida de sua velocidade ou capacidade; os terminais representam as máquinas que devem receber o pacote.

O Problema de Steiner em Grafos é NP-difícil [9], o que significa que provavelmente não existe um algoritmo polinomial capaz de resolvê-lo. Dada a sua importância prática, no entanto, é plenamente justificável que se tente resolver o problema da melhor forma possível, seja através de algoritmos aproximados, seja através de algoritmos exatos (ainda que, no pior caso, tenham tempo de execução superpolinomial).

O Problema de Steiner em Grafos, nesta dissertação, será chamado pela sua sigla em inglês, SPG (*Steiner Problem in Graphs*). A entrada do problema é constituída por

um grafo não-orientado G = (V, E), sendo V o conjunto de vértices, e E de arestas e por um conjunto T de terminais, sendo T $\subseteq$ V. O grafo é positivamente ponderado, o que significa que a toda aresta e $\in$ E está associado um custo (ou peso) c(e) maior que zero. Veja o exemplo da Figura 2.1.1, onde os vértices são representados pelos círculos, os terminais pelos quadrados e as arestas pelas linhas que interligam os círculos e quadrados, onde cada uma possui seu custo representado pelo número próximo a esta.

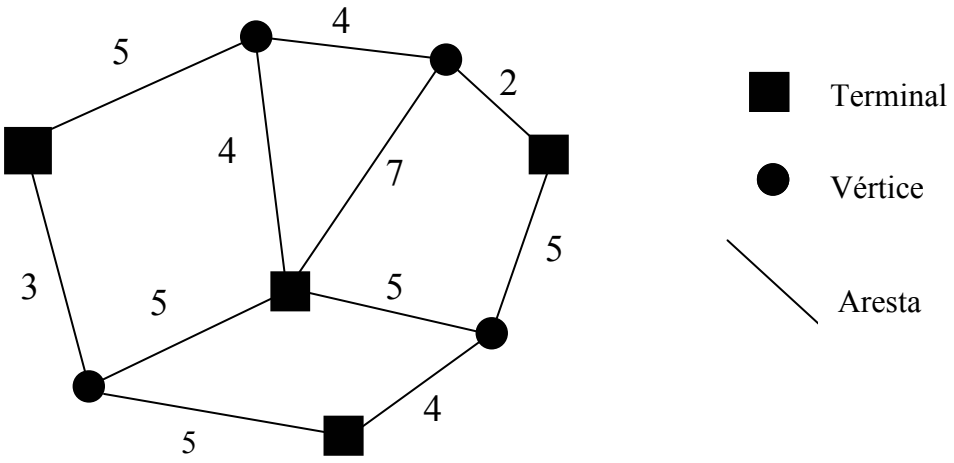


Figura 2.1.1 : Instância de um Problema de Steiner em Grafos

Para resolver o Problema de Steiner em Grafos devemos encontrar um sub-grafo conectado (V' , E') de G com T $\subseteq$ V' minimizando $\sum_{e \in E'} c_e$. Em outras palavras, encontrar a árvore de custo mínimo contendo todos terminais, sendo possível conter também alguns vértices não terminais. A Figura 2.1.2 ilustra uma solução ótima para o exemplo apresentado anteriormente.

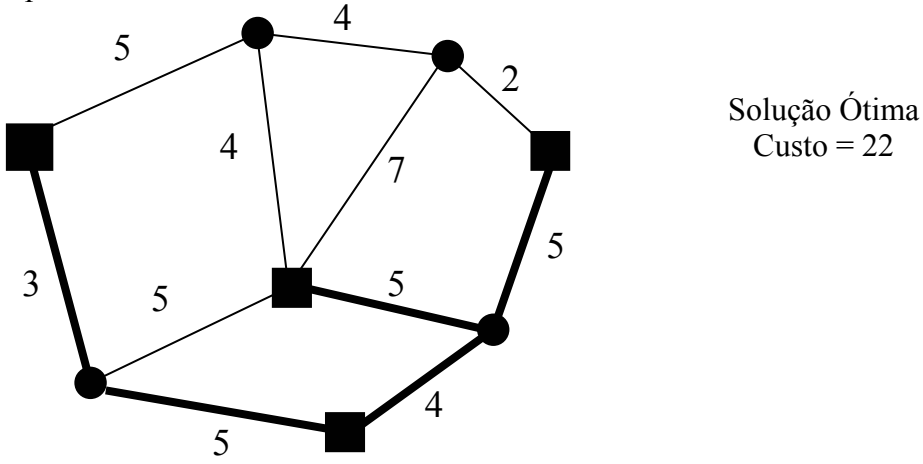


Figura 2.1.2 : Árvore de Steiner mínima

2.2 *Branch-and-bound*

Um algoritmo de *branch-and-bound* é um método de enumeração implícita de todas as possíveis soluções de uma instância de entrada. Ele requer duas rotinas básicas: uma para calcular limites superiores, outra para calcular limites inferiores. Inicialmente aplicam-se as rotinas sobre a instância de entrada. Se o limite inferior for igual ao superior, o algoritmo termina. Se não for o caso, faz-se um ramificação (ou *branching*): o problema é particionado em dois ou mais subproblemas que, juntos, são equivalentes ao problema original.

No caso de um problema de otimização discreta 0 - 1, a ramificação é normalmente feita escolhendo-se uma variável de *branching*, que tem seu valor fixado em 0 em um dos subproblemas, e em 1 no outro. A melhor das soluções desses subproblemas é uma solução do problema original. Observe que o *branch-and-bound* constrói uma árvore de resolução durante sua execução. Cada nó da árvore dá origem a dois nós-filhos, a menos que o seu limite inferior seja maior ou igual ao menor limite superior global conhecido (supondo um problema de minimização). Nesse caso, o subproblema é considerado resolvido. Diz-se que o ramo que tem esse nó como raiz foi podado, ou que se fez o *pruning* do nó.

Assim sendo, o número de nós efetivamente visitados na árvore (que é exponencial no pior caso) depende da qualidade dos limites inferiores e superiores calculados ao longo da execução do algoritmo. Com limites melhores, o número de ramificações feitas até que se possa fazer um *pruning* é menor.

Para unificar a explicação de algoritmos de *branch-and-bound* tanto para problemas de minimização quanto para problemas de maximização, é usual definir limites duais como sendo limites inferiores no caso de minimização ou limites superiores no caso de maximização. Da mesma forma, define-se limites primais como sendo limites superiores em caso de minimização ou inferiores no caso de maximização. Um pseudo-código de um algoritmo de *branch-and-bound* para um caso de minimização é mostrado na Figura 2.1.3. A variável global `GLP` é inicializada com

valor infinito. Chama-se o *procedure branch-and-bound* tendo como parâmetro uma instância `G,` cada chamada recursiva a esse procedimento define um novo nó na árvore de enumeração. Ao final do processo, a variável global `GLP` terá o valor da solução ótima do problema. É importante notar que o teste de *pruning* pode usar limites primais já encontrados em qualquer nó já explorado da árvore.

```
Global GLP = infinito;  /*Valor do melhor limite primal já encontrado*/

procedure branch-and-bound (G) {
     LD ← limite dual para G;
     LP ← limite primal para G;
     se (LP < GLP) {GLP ← LP;}
     se (LD ≥ GLP) {retorne;}   /* Pruning   */
    senão {                    /* Branching */
           particione G em dois subproblemas, G1 e G2;
           branch-and-bound (G1);
           branch-and-bound (G2);
     }
}
```

Figura 2.1.3 : Pseudo-código do *Branch-and-bound*

2.2.1 Limites Duais

O algoritmo seqüencial para o SPG usado nesta dissertação obtém os limites inferiores através de uma heurística construtiva baseada no algoritmo *dual Ascent*, proposto inicialmente por Wong [31].

Este algoritmo se baseia na formulação por cortes dirigidos para o problema. A formulação por cortes dirigidos (DCF) do SPG trabalha no grafo dirigido $G_D = (V, A)$ obtido substituindo-se cada aresta em E por dois arcos de direções opostas e

escolhendo-se um terminal qualquer para ser uma raiz r. Seja W a coleção de todos os conjuntos de vértices w contendo algum terminal mas não contendo a raiz e seja $\delta^-(w)$ o corte dirigido constituído pelos arcos de entrada em w. Seja x_a uma variável binária indicando se o arco a pertence à solução (uma arborescência com raiz em r) ou não. A relaxação linear do DCF (P), e seu programa linear dual (D), são, respectivamente:

$$
\begin{aligned}
(P) \qquad & Min \sum (c_a x_a : a \in A) \\
& s.t.\ x(\delta^-(w)) \geq 1, \ \forall w \in W \\
& x_a \geq 0, \ \forall a \in A
\end{aligned}
$$

$$
\begin{aligned}
(D) \qquad & Max \sum (y_w : w \in W) \\
& s.t.\ \sum (y_w : a \in \delta^-(w)) \leq c_a, \ \forall a \in A \\
& y_w \geq 0, \ \forall w \in W
\end{aligned}
$$

A solução de (P) geralmente fornece limites duais muito próximos ao valor inteiro ótimo, entretanto, devido ao número exponencial de restrições, este programa linear deve ser resolvido por geração de cortes. Quando a geração de cortes está embutida em um algoritmo *branch-and-bound*, temos um algoritmo *branch-and-cut*. Para as instâncias do SPG mais difíceis, esses algoritmos podem ser ineficientes: devido a problemas de convergência na geração de cortes, pode-se precisar de muito tempo para resolver cada nó.

Wong [31] propôs um algoritmo combinatorial rápido (*dual ascent*) para encontrar soluções aproximadas para (D). O valor de tais soluções são limites duais válidos. O *dual ascent* pode ser descrito da seguinte forma. Seja y uma solução válida de (D). Denote o custo reduzido de um arco a em relação a y por :

$$c_a(y) = c_a - \sum(y_w : a \in \delta(w)).$$

Inicialize $y = 0$ e a cada iteração incremente a variável y_w tanto quanto possível, onde w corresponde ao conjunto maximal de vértices que podem alcançar um terminal, mas não

r, por arcos de custo reduzido zero. Boas soluções inteiras podem ser obtidas a partir de um conjunto final de arcos de custo reduzido zero, chamados arcos saturados.

Vejamos um exemplo de execução do algoritmo de *dual ascent*. Este trabalha com dois grafos direcionados: G (Figura 2.2.1.1 – lado esquerdo da figura), o grafo original, e G_y (Figura 2.2.1.1 – lado direito da figura), subgrafo de G contendo apenas os arcos saturados em relação à solução dual *y*. O grafo G permanece imutável ao longo de todo o algoritmo, apenas os custos reduzidos associados aos seus arcos são alterados. Ao grafo G_y (grafo de saturação), são adicionados novos arcos no decorrer da execução. Um conceito essencial no algoritmo de Wong é o de componente-raiz. Seja *R* uma componente fortemente conexa de G_y com conjunto de vértices *V(R)*. *R* será uma componente-raiz se obedecer simultaneamente às seguintes condições:

1) *V (R)* contém pelo menos um terminal;

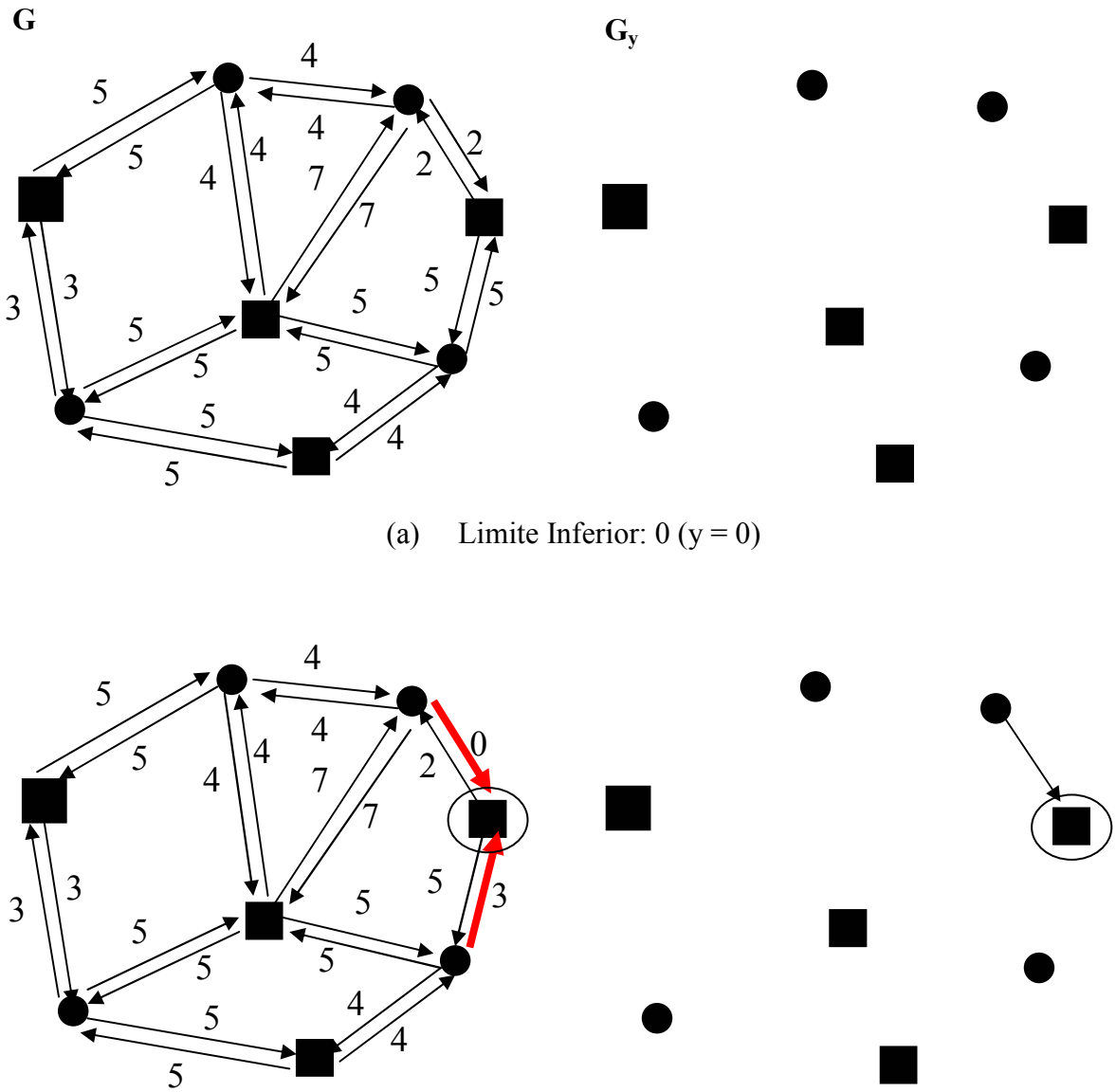
2) *V (R)* não contém a raiz *r*;

3) Não existe um caminho em G_y de um terminal $t \notin R$ até *R*.

Note que *w(R)* é o conjunto dos vértices que alcançam *R* em G_y, incluindo os próprios vértices de *R* e que a variável dual associada ao corte definido por *w(R)* é denotada por $y_{w(R)}$.

Considere a execução do algoritmo representada na Figura 2.2.1.1. Considere que y = 0 no início, como mostrado na Figura 2.2.1.1(a). Como todas as variáveis duais são nulas, não há arcos saturados em G e G_y não contém arco algum. Nesse momento, cada terminal (exceto a própria raiz) é uma componente-raiz. Em cada iteração, o algoritmo escolhe uma componente-raiz e aumenta o valor da variável dual a ela associada até que pelo menos um arco esteja saturado. Na Figura 2.2.1.1, de (b) até (j), os componentes raízes escolhidos são destacados. A cada iteração pelo menos um novo arco é adicionado a G_y, o que pode provocar o aumento de algumas componentes raízes e/ou a destruição de algumas outras (em razão de passarem a ser alcançáveis a partir de

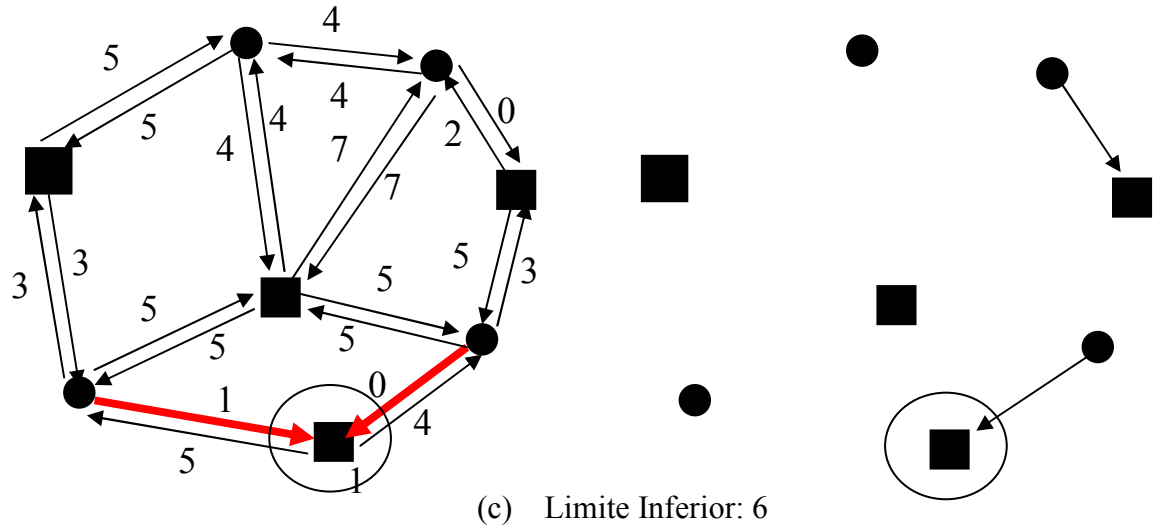
terminais). Ao final de no máximo O(|E|) iterações, todos os terminais serão alcançáveis a partir de *r* em G_y. Não havendo mais componentes raízes, o algoritmo termina.



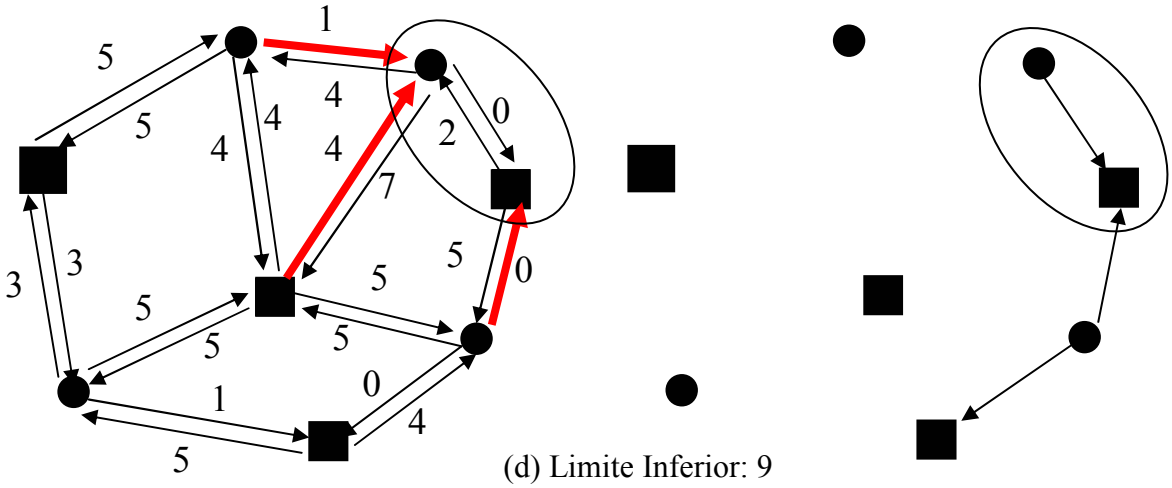
(a) Limite Inferior: 0 (y = 0)

(b) Limite Inferior: 2

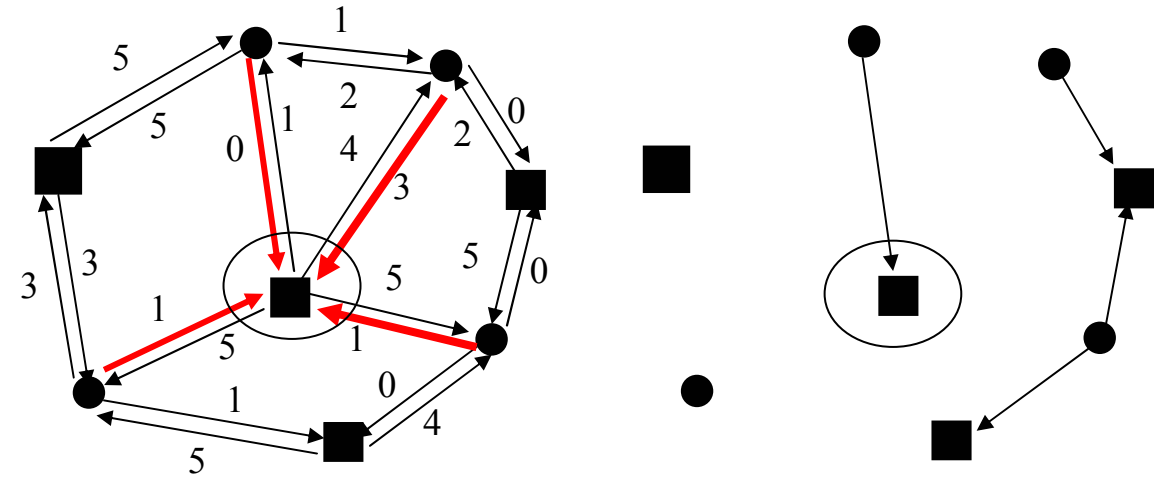
Figura 2.2.1.1 : Aplicação da técnica de *dual ascent*



(c) Limite Inferior: 6

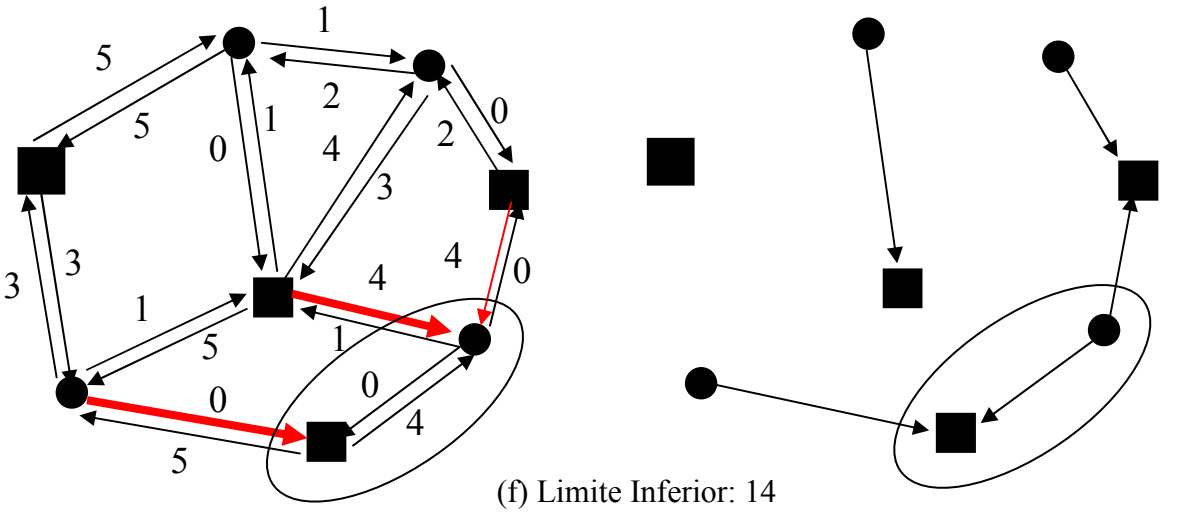


(d) Limite Inferior: 9

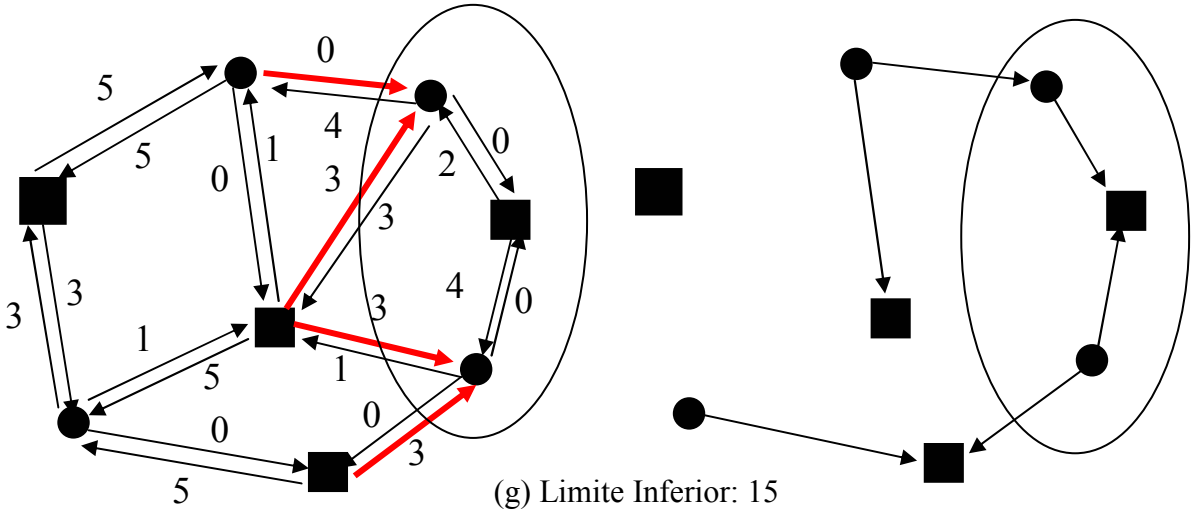


(e) Limite Inferior: 13

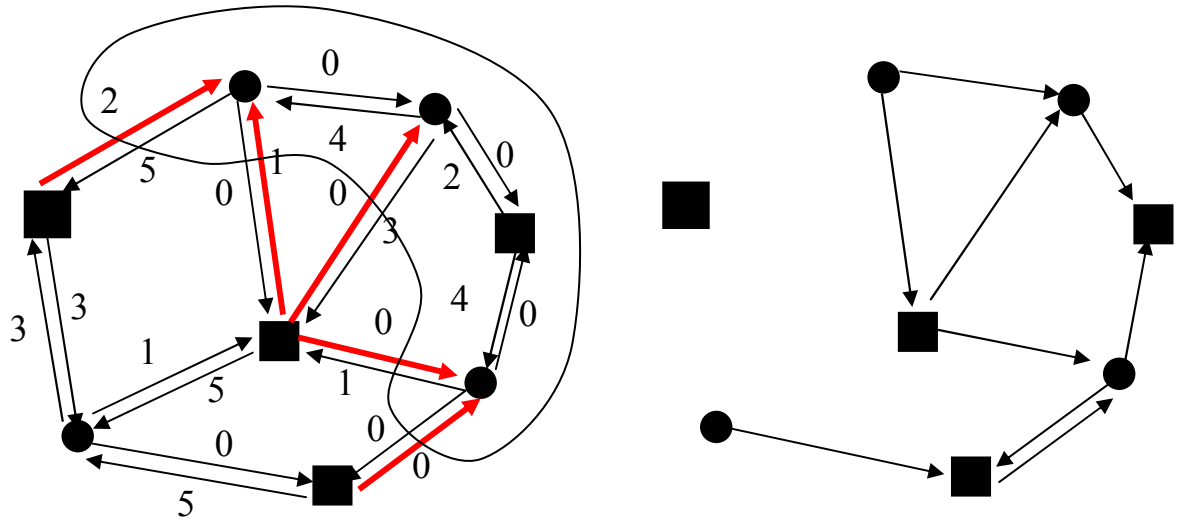
Figura 2.2.1.1 : Aplicação da técnica de *dual ascent*



(f) Limite Inferior: 14

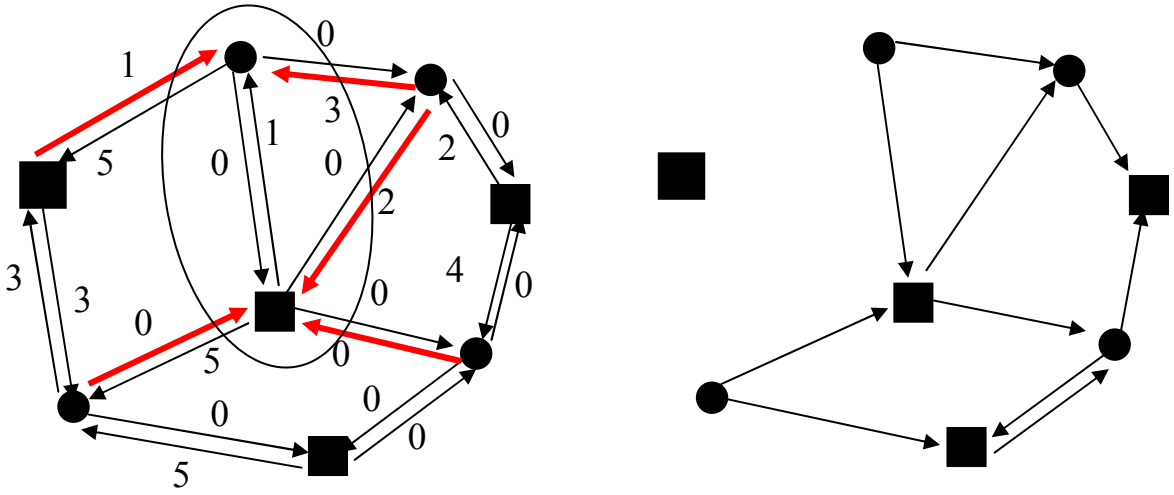


(g) Limite Inferior: 15

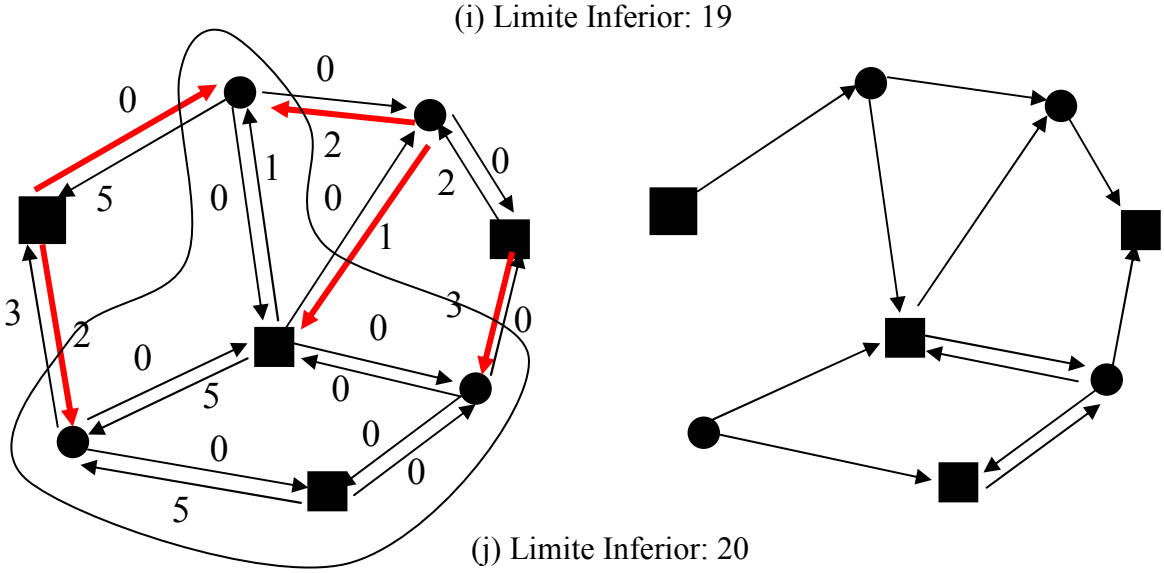


(h) Limite Inferior: 18

Figura 2.2.1.1 : Aplicação da técnica de *dual ascent*



(i) Limite Inferior: 19



(j) Limite Inferior: 20

Figura 2.2.1.1 : Aplicação da técnica de *dual ascent*

Poggi de Aragão, Uchoa e Werneck [22] propuseram três heurísticas combinatórias para melhorar os limites duais dados pelo *dual ascent*: *dual scaling*, *dual adjustment*, e *active fixing by reduced costs*. Essas heurísticas mais complexas fazem parte do algoritmo *branch-and-bound* descrito em [26] e que é usado nesta dissertação.

2.2.2 Limites Primais

O valor de qualquer solução válida para o SPG fornece um limite primal. Tais limites são obtidos do procedimento de *reverse delete step,* que consiste em remover os arcos de G_y (Figura 2.2.2.1), na ordem inversa em que foram saturados, tomando-se o cuidado de não desconectar o grafo. Não se exige que todo o grafo G, permaneça conexo; exige-se apenas que sejam preservados caminhos da raiz a todos os terminais do grafo.

O resultado será uma arborescência contendo todos os terminais, que facilmente pode ser convertida em uma árvore de Steiner no grafo não-direcionado original. Para tornar as soluções primais ainda melhores, são usados métodos de busca local baseados em nós de Steiner e caminhos-chaves, descritos em Ribeiro, Uchoa e Werneck [24].

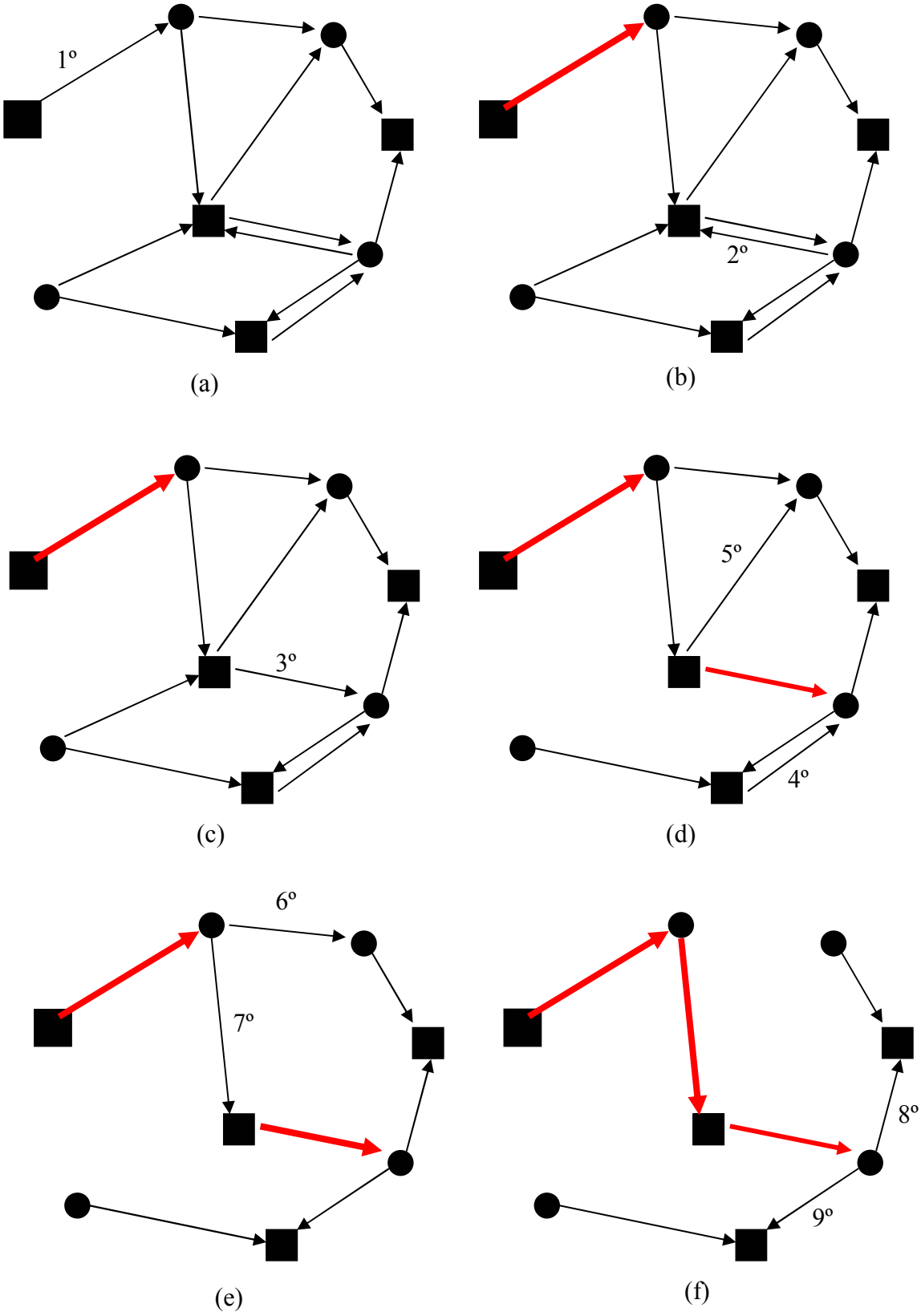


Figura 2.2.2.1 : Aplicação do procedimento de *reverse delete step*

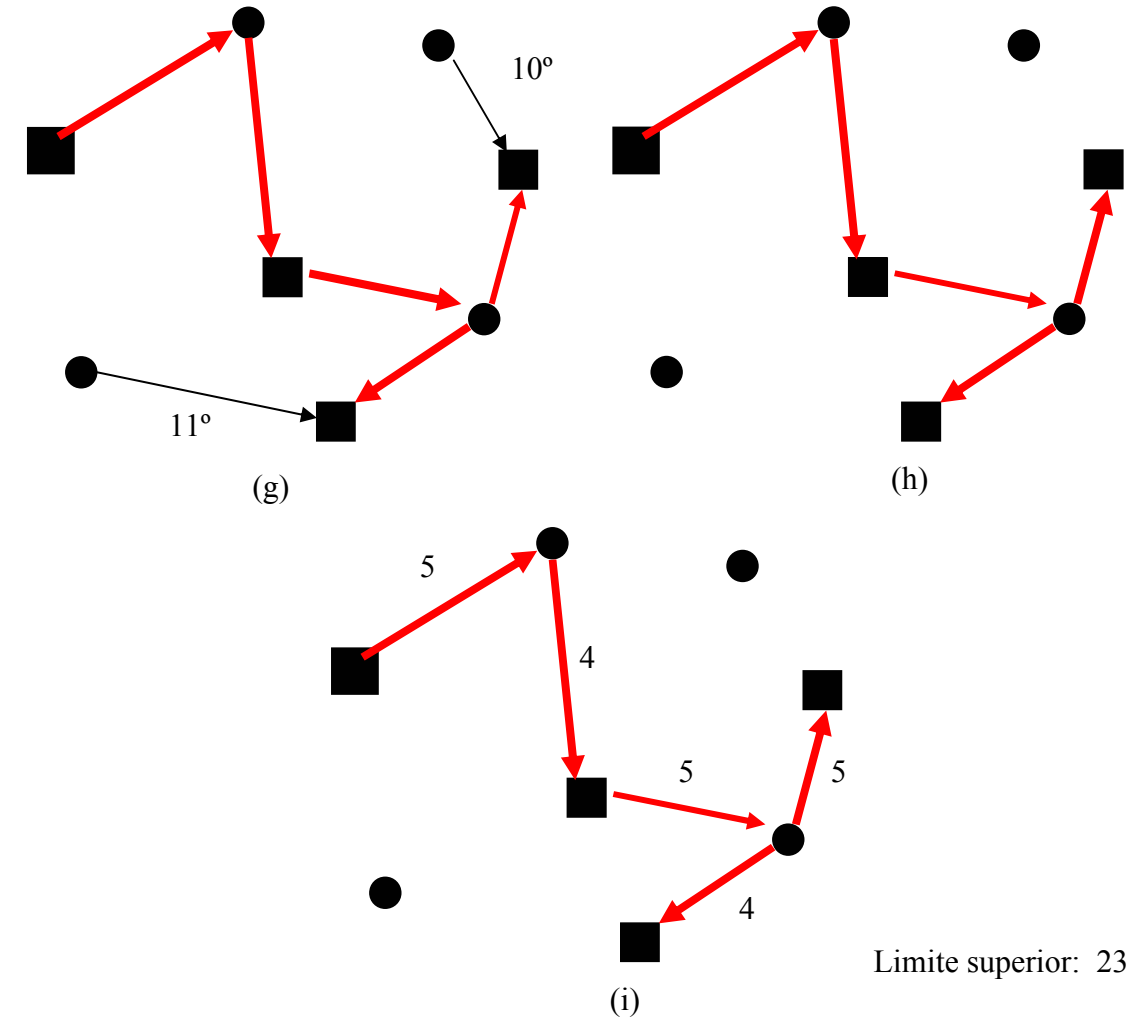


Figura 2.2.2.1. Aplicação do procedimento de *reverse delete step*

2.2.3 *Branching*

O princípio básico utilizado é de se fazer uma ramificação binária com base nos vértices do grafo. Seja v o vértice escolhido para ramificação (um não-terminal, obrigatoriamente). Em um dos subproblemas (ramos), exige-se que v faça parte da solução, o que é feito por meio da transformação do vértice em um terminal. No outro problema, exige-se que o vértice não faça parte de qualquer solução, o que se faz eliminando o vértice (e os arcos associados) do grafo. Veja a Figura 2.2.3.1, que apresenta alguns passos de *branching*.

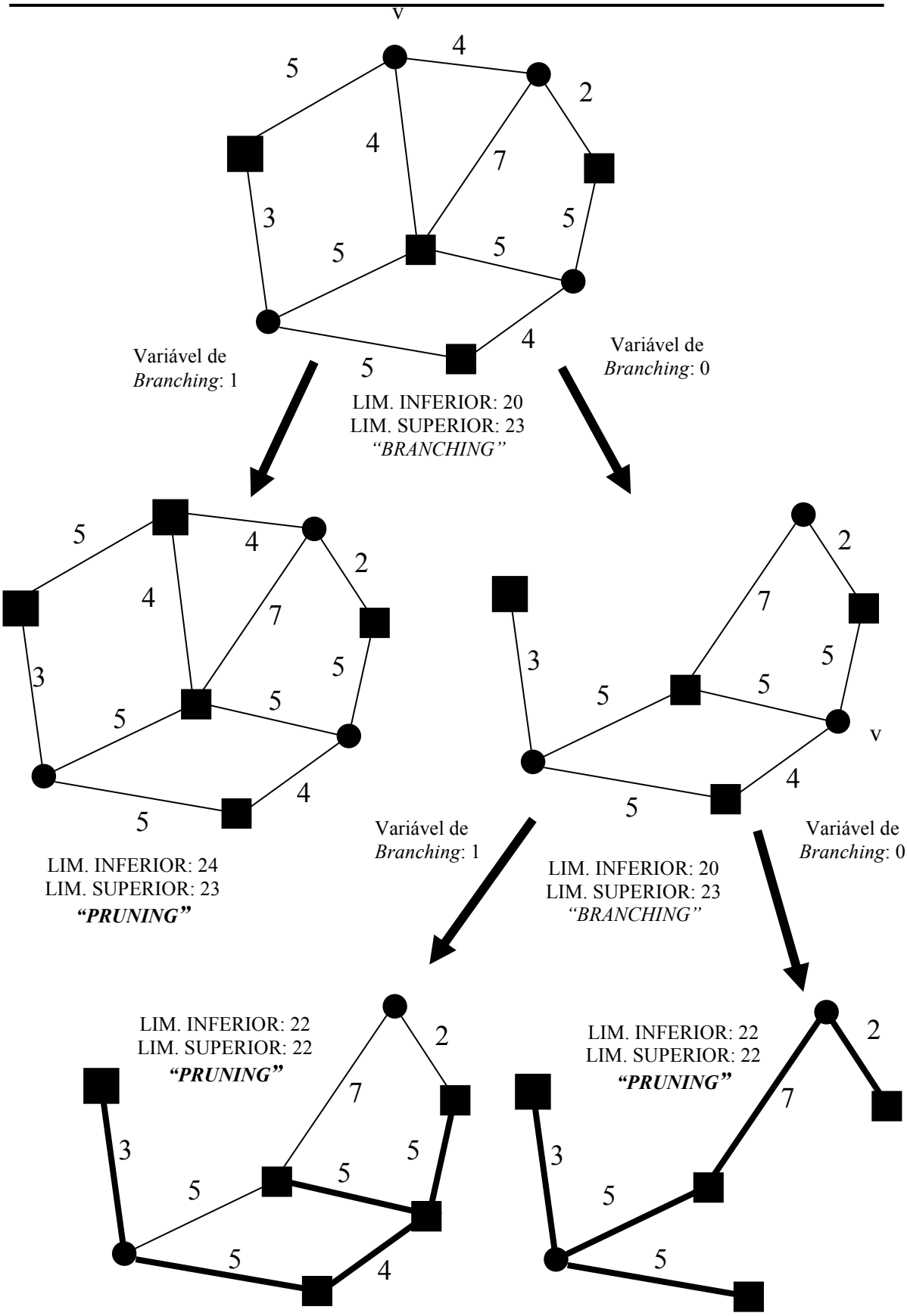


Figura 2.2.3.1 : Busca da solução com etapas de *branching* e *prunning*.

2.3 As Instâncias de Teste e o Algoritmo Seqüencial

As instâncias para o problema de Steiner são muito dependentes da aplicação a que se referem. Para o projeto de circuitos VLSI, por exemplo, normalmente as instâncias são retilineares, com um número relativamente grande de vértices (e às vezes de terminais) e baixa densidade (o grau dos vértices é limitado a quatro). Em outros casos, o número de vértices é menor, mas o número de arestas pode se multiplicar. Quando se desenvolve um algoritmo de propósito geral, é interessante que seja testado em um conjunto de instâncias que seja tão diversificado quanto possível.

Propostas por Duin [9], as instâncias de incidência são construídas sobre grafos aleatórios e têm como principal propriedade o fato de que suas arestas têm pesos de incidência, isto é, o peso de uma aresta é um inteiro escolhido em uma distribuição normal cuja média depende do número de terminais em que a aresta é incidente (0, 1 ou 2). Se ambas as extremidades da aresta forem não terminais, a média é 100; se a aresta for incidente em exatamente um terminal, a média é 200; se for incidente em dois terminais, a média é 300. Essa escolha de pesos tinha a finalidade de tornar ineficazes os testes de redução normalmente utilizados no SPG (Duin e Volgenant [10]). Os novos testes propostos em Uchoa, Poggi de Aragão e Ribeiro [24] também não reduzem tais instâncias.

A classe de instâncias de incidência está dividida em quatro séries, de acordo com o número de vértices (n): i080, i160, i320 e i640. O algoritmo seqüencial no qual esta dissertação se baseia foi capaz de resolver em tempos razoáveis (até 24 horas) 380 dessas 400 instâncias de incidência (Uchoa [27,28], Werneck [30]). Das 20 instâncias restantes (todas da série i640), 5 ainda não foram resolvidas e as 15 restantes podem ser resolvidas através de longos tempos de processamento, chegando a uma semana de CPU em um computador Pentium IV 1.7GHz com 256Mb de memória RAM. Por sua dificuldade, 5 instâncias, das 15 resolvidas de longo tempo de processamento, foram escolhidas para os testes com os algoritmos distribuídos apresentados nos capítulos seguintes dessa dissertação. A tabela 2.3.1 apresenta as características de cada uma

dessas instâncias e o seu valor de solução ótimo. Também é apresentado o tempo de CPU gasto pelo algoritmo seqüencial numa máquina Pentium IV 1.7GHz com 256Mb de memória RAM.

INSTÂNCIAS	VÉRTICES (V)	ARESTAS (E)	TERMINAIS (T)	SOLUÇÃO ÓTIMA	TEMPO SEQÜENCIAL (segundos)
I640-211	640	4135	50	11984	545387
I640-212	640	4135	50	11795	43479
I640-213	640	4135	50	11879	38187
I640-214	640	4135	50	11898	393260
I640-215	640	4135	50	12081	166188

Tabela 2.3.1 : Instâncias de incidência utilizadas.

Encontramos na literatura diversos trabalhos sobre paralelização do *branch-and-bound* [3,4,5,7,15]. Nosso código seqüencial *branch-and-bound,* utilizado como base para este trabalho, tem características que o tornam muito apropriado para paralelização em *Grids*. Em primeiro lugar, existem muitos nós na árvore, mas cada nó é resolvido rapidamente, o que favorece o balanceamento de carga. Além disso, não se exige o uso de resolvedores de programação linear. Resolvedores comerciais de programação linear, tais como CPLEX ou XPRESS, são muitos caros e não podem ser considerados como existentes em todas as máquinas em uma *Grid*. Infelizmente, não existem resolvedores de programação linear gratuitos com desempenho comparável. Finalmente, a quantidade de dados necessários para especificar os subproblemas que devem ser resolvidos pelo processador é muito pequena. Nenhuma mensagem grande precisa transitar na *Grid*.

Cada mensagem correspondente a um subproblema em nosso algoritmo consiste de um pequeno subconjunto de vértices. Quando um nó neste algoritmo SPG necessita fazer uma ramificação, este escolhe um certo vértice $i \in V$. Fixar i em 0 significa que i não deve estar na solução daquele subproblema. Este vértice e suas arestas incidentes

são removidas do grafo. Fixar *i* em 1 significa que *i* deve pertencer à solução daquele subproblema. Um processador pode enviar um subproblema de nível *l* para ser resolvido por um outro processador através de uma mensagem contendo uma lista de cerca de *l* números. Mesmo na instâncias mais difíceis, *l* raramente ultrapassa o valor de 40.

Veja o exemplo da Figura 2.3.1. Após um *branching*, o próprio processador deverá ficar encarregado de processar um ramo da sub-árvore, por exemplo, a sub-árvore 0; e o ramo 1 deverá ser enviado o outro processador, através de uma mensagem indicando os vértices percorridos até então. No primeiro *branching*, consideramos que o ramo 1 será enviado, e que X é o vértice a ser marcado como terminal, a mensagem deverá ser: {X , 1}. No segundo *branching,* também enviando o ramo 1 a partir do processador que tenha processado o ramo 0 do primeiro *branching*, e sendo Y o número do próximo vértice a ser considerado como terminal, a mensagem deverá ser {X , 0, Y , 1}.

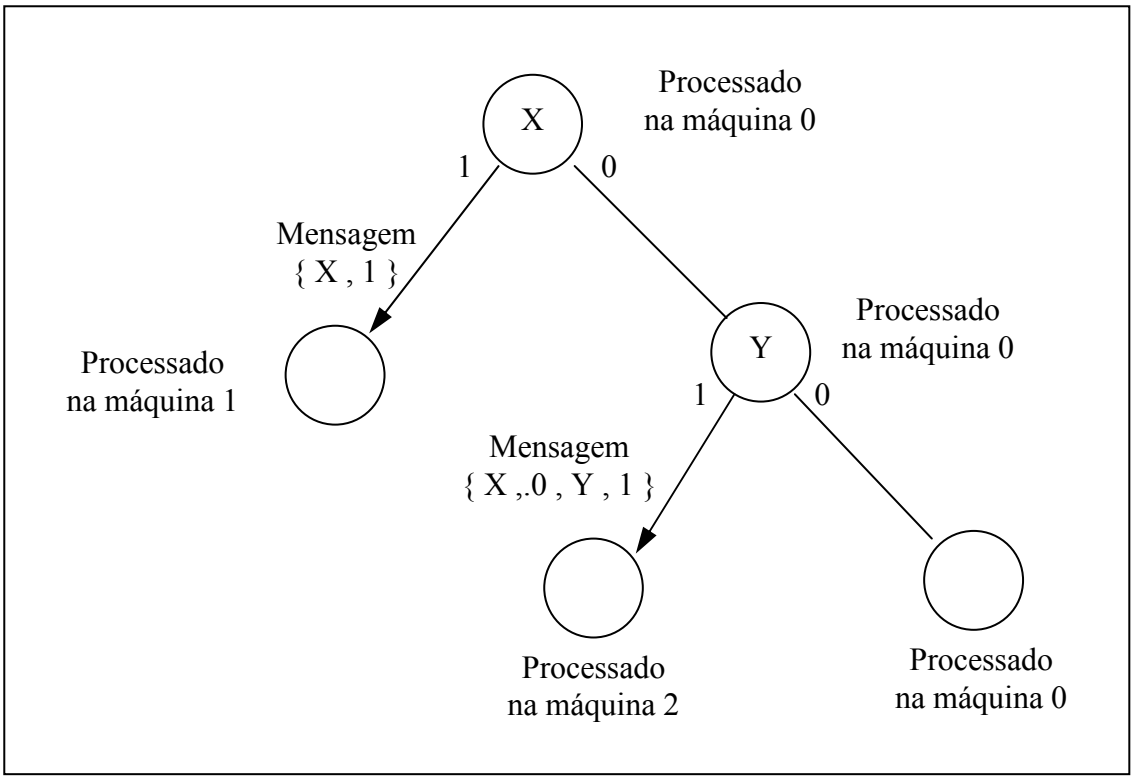


Figura 2.3.1 : Mensagens enviadas em uma distribuição de carga do *branch-and-bound*

Nós poderíamos incluir mais informações auxiliares nas mensagens na tentativa de ajudar as tarefas do processador que recebe o subproblema: uma lista de arestas que já foram reduzidos a 0 por custos reduzidos. Nós decidimos não fazê-lo pelas seguintes razões: (I) o tamanho das mensagens trocadas cresceria demasiadamente (existem instâncias de incidência com mais de 200.000 arestas); (II) nós descobrimos que quase todas estas arestas são fixadas novamente em 0 logo no primeiro nó resolvido pelo processador. Em outras palavras, enviar estas grandes mensagens poderia somente acelerar a solução do primeiro nó de um subproblema. Como nós enviamos para cada processador subproblemas com muitos nós, estas pequenas reduções do tempo de CPU não compensariam o aumento do custo de comunicação.

Capítulo 3

Um algoritmo *Branch-and-Bound* Distribuído

O algoritmo que nós propomos é composto dos seguintes procedimentos: distribuição inicial de carga, balanceamento de carga, detecção de terminação, difusão do limite *primal*, e tolerância a falhas, descritos nas seções seguintes.

O algoritmo assume que existe uma estratégia de alocação estática de tarefas que atribui exatamente um único processo a cada processador físico. Então estas duas palavras serão usadas indistintamente durante este trabalho. Todo processador está alocado em um cluster. Cada processo tem um único número de identificação de *0* até *m - 1*, onde *m* é o número de processadores disponíveis. Processos no mesmo *cluster* têm números consecutivos. Cada *cluster* também tem um único número de identificação de 0 até *n* - 1, onde *n* é a quantidade de clusters. Cada *cluster* tem seu líder, que corresponde ao processo com o menor número de identificação dentro do *cluster*. O líder do *cluster* 0 é responsável por iniciar e finalizar a aplicação, mas seu papel não pode ser confundido com o atribuído a um processador mestre em um paradigma mestre-escravo. Os líderes comportam-se como quaisquer outros processos.

Considere a seguinte notação:

- *c(j)* – identificação do *cluster* que contém o processo *j ;*
- *s(i)* – tamanho do *cluster i* ;
- *l(i)* - identificação do processo líder do *cluster i*;
- *sp(j); pp(j)* – sucessor e antecessor do processo *j* em seu *cluster*, onde
  $sp(j) = (j - l(c(j)) + 1) \bmod s(i) + l(c(j))$ e
  $pp(j) = (j - l(c(j)) - 1) \bmod s(i) + l(c(j))$;
- *sc(i); pc(i)* – sucessor e antecessor do *cluster i, onde*
  $sc(i) = (i+1) \bmod n$ *e*
  $pc(j) = (i - 1) \bmod n.$

3.1 Distribuição Inicial

A fase de distribuição inicial consiste na distribuição das sub-árvores pelos processadores. O procedimento inicia no *cluster* 0. Depois de um *branch* no nível *l*, o líder do *cluster i* envia a sub-árvore direita resultante para o líder do cluster *nextclust* $= i + 2^l$. Se *nextclust* for maior do que $n - 1$, um procedimento análogo é iniciado para distribuir a carga de *l(i)* com os outros processos no *cluster*. Mais especificamente, o nível *l* é reinicializado com zero e cada processo *j* envia uma sub-árvore para o processo *nextproc,* definido como $nextproc = j + 2^l$, até *que* $nextproc \geq l(i) + s(i)$.

Considere este procedimento com onze processadores divididos em quatro *clusters*. O *cluster* de identificação 0 tem processos {0,1,2}; o *cluster 1* tem {3,4,5,6}; *cluster* 2 possui {7,8} e *cluster* 3 possui {9, 10}. Inicialmente, o processo *l(0)* = 0 que executa o nó raiz, faz um *branch*, encaminha a sub-árvore da direita para o processo *l(1)* = 3 e passa a processar a sub-árvore esquerda. Depois de fazer o *branch* pela segunda vez, o processo 0 compartilha sua carga, agora, com o *l(2)* = 7, enquanto 3 compartilha sua carga com *l(3)* = 9. No terceiro nível, o *nextclust* calculado por 0 é 4, que é maior que *n* -1. Portanto, 0 inicia a distribuição entre processadores em seu *cluster* alocando metade de sua carga ao processador 1.

Esta distribuição é apresentada na Figura 3.1.1, onde os círculos representam os processos, e os números em seus interiores as suas identificações. Os processos estão agrupados em *clusters* indicados pelas linhas pontilhadas. O processo 0, líder da aplicação, inicia a divisão de carga, na primeira etapa, entre os clusters representados pelos seus líderes, obedecendo a fórmula descrita acima. Quando não existe mais *cluster* disponível, este inicia a divisão de carga em seu próprio *cluster*.

A distribuição inicial entre os líderes do *cluster* emprega a mensagem *SubtreeToCluster*. A distribuição entre processos em um mesmo cluster utiliza a mensagem *Subtree*. Ambas as mensagens contêm uma lista de vértices 0 e 1 em ordem para definir um subproblema específico para cada *cluster*/processo.

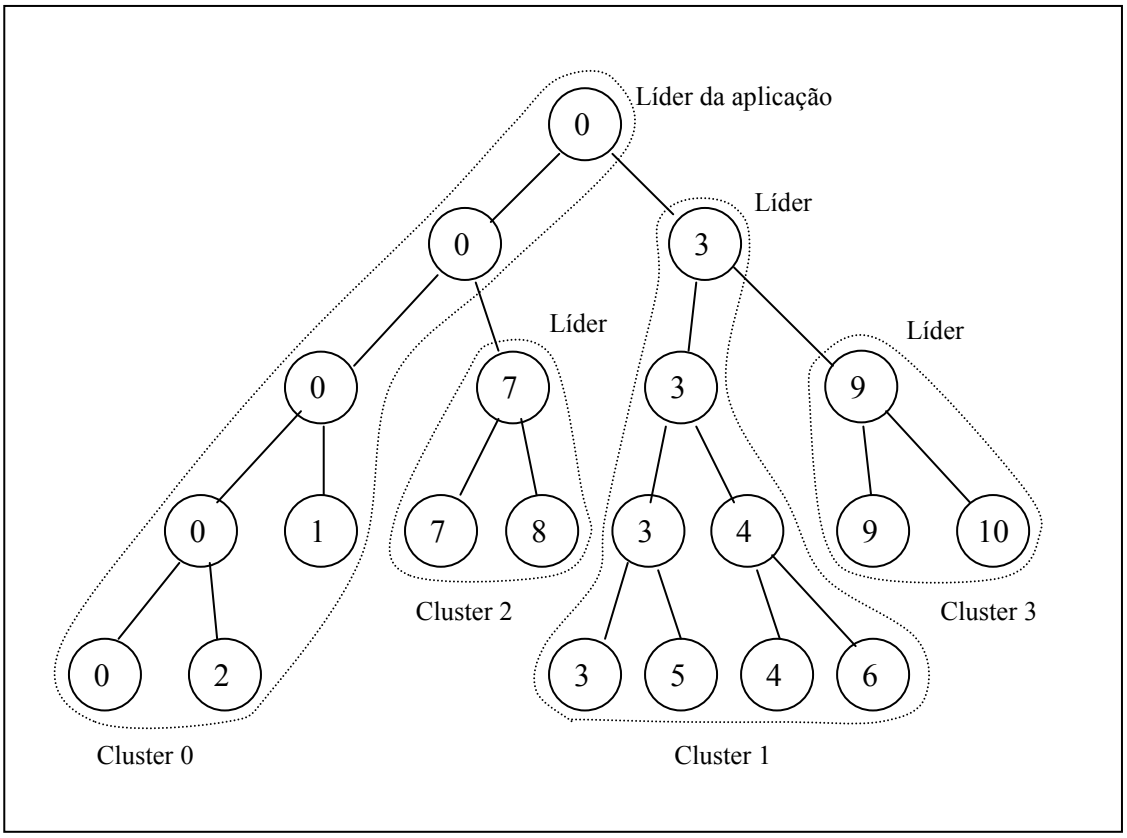


Figura 3.1.1 : Distribuição inicial

Se um processo finaliza seu subproblema precocemente, de tal modo que não tenha uma sub-árvore para um cluster, *nextclust* < *n* ou processo *nextproc* < *l(i)* + *s(i),*

este envia uma mensagem *AwakeCluster / Awake***.** É possível que o processo que receba esta mensagem, tenha de enviar uma outra mensagem de *AwakeCluster* **/** *Awake***.**

De qualquer forma, estes processos iniciam suas atividades pedindo carga aos outros processos, como será discutido na próxima seção.

3.2 Balanceamento de carga

O tempo requerido para um processador executar uma determinada sub-árvore não é conhecido *a priori* e pode ter significativa variação. Portanto, um procedimento de balanceamento de carga entre os processadores deve existir. A fim de evitar comunicação desnecessária, optamos por usar algoritmos de balanceamento de carga "*receiver-initiated*" [11]. Assim, ao finalizar a execução de sua sub-árvore, um processo pede ao seu vizinho no *cluster* parte de sua sub-árvore. Desta forma, um processo tenta obter carga de processo *pp(j).* Se este processo estiver ocioso, *j* tentará *pp(pp(j))* e assim por diante.

Quando um processador alcança sua condição local de terminação, isto é, não é capaz de obter sub-árvores de seus vizinhos, este processador informa este fato ao seu líder. Quando todos os processos no *cluster i* alcançam esta condição, o líder do *cluster* tenta obter carga do líder do *cluster pc(i)*, se não for possível, tentará novamente com o líder do *cluster pc(pc(i))* e assim por diante. Se *l(i)* obtém a requerida carga, esta é distribuída entre os processadores no *cluster* por um procedimento similar ao da distribuição inicial. Caso contrário, o *cluster* alcança sua condição local de terminação e informa ao líder da aplicação.

A mensagem de *LoadRequest* é usada pelos processos para obter uma sub-árvore de outros processos no mesmo *cluster* conforme Figura 3.2.1. O recebimento desta mensagem é respondido por meio de uma mensagem de *SubTree* ou uma mensagem de *Idle*.

Se um receptor estiver correntemente resolvendo uma sub-árvore, ele envia uma mensagem de *SubTree* contendo um nó não resolvido de um nível mais baixo. E uma

resposta *Idle* significa que o processo não tem carga para compartilhar, e deverá estar, também, requisitando carga ou iniciando o pedido de carga aos outros processos.

Um processo que não foi capaz de obter carga de seus vizinhos inicia um procedimento de terminação local enviando mensagem de *SuspectEnd* ao líder do seu *cluster* .

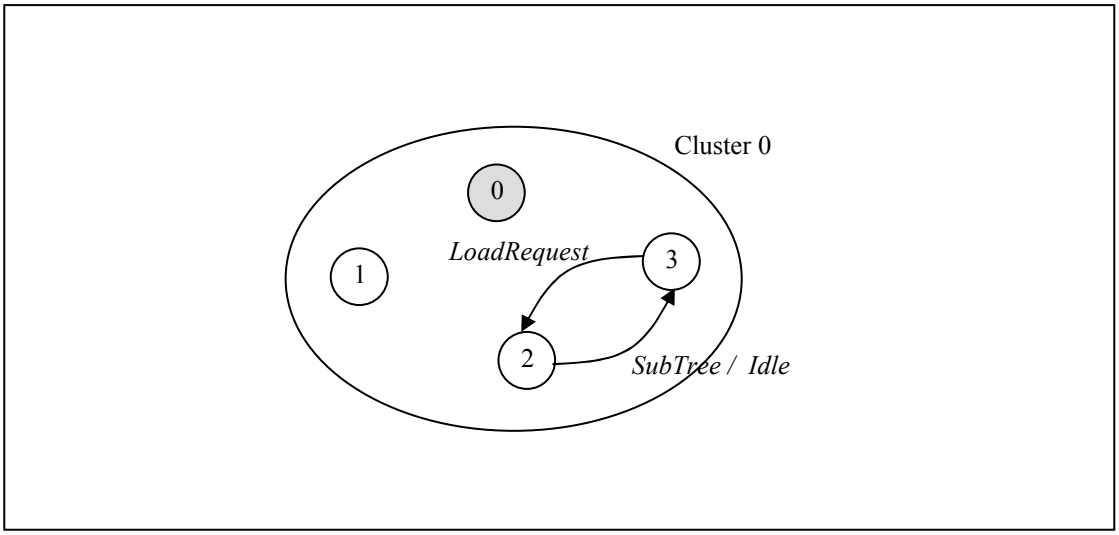


Figura 3.2.1 : Pedido de carga através da mensagem *LoadRequest*

O líder, após receber esta mensagem de todos os processos dentro de seu *cluster* envia uma mensagem de *LoadRequestCluster* para os líderes dos outros *clusters*. Esta é respondida com uma mensagem de *SubtreeToCluster,* ou uma mensagem de *NotNow* ou uma mensagem de *IdleCluster* (Figura 3.2.2).

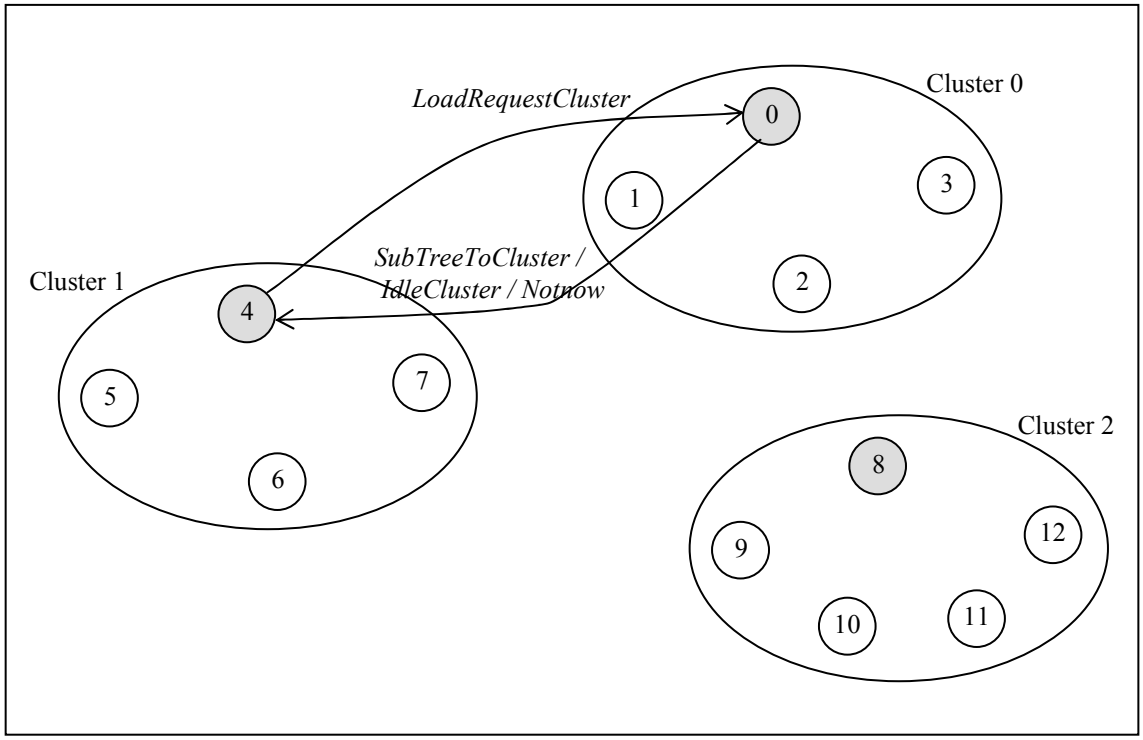


Figura 3.2.2 : Balanceamento de carga entre *clusters* diferentes

Uma mensagem de *NotNow* significa que o líder não tem carga para compartilhar naquele momento, mas pode ser capaz de obter carga dentro de seu *cluster* e enviar parte desta para outro *cluster* em um momento mais adiante. A mensagem de *IdleCluster* é a resposta enviada de um *cluster* que tenha atingido sua condição local de terminação. Um líder não capaz de obter carga de qualquer outro *cluste*r envia uma mensagem de *SustectEndCluster* para o líder da aplicação (Figura 3.2.3).. Depois de receber tal mensagem de todos os líderes de *clusters*, o líder da aplicação difunde a mensagem de *Terminate* para todos os líderes (Figura 3.2.4), e estes, por sua vez, enviam para os processos dentro de seus *clusters*.

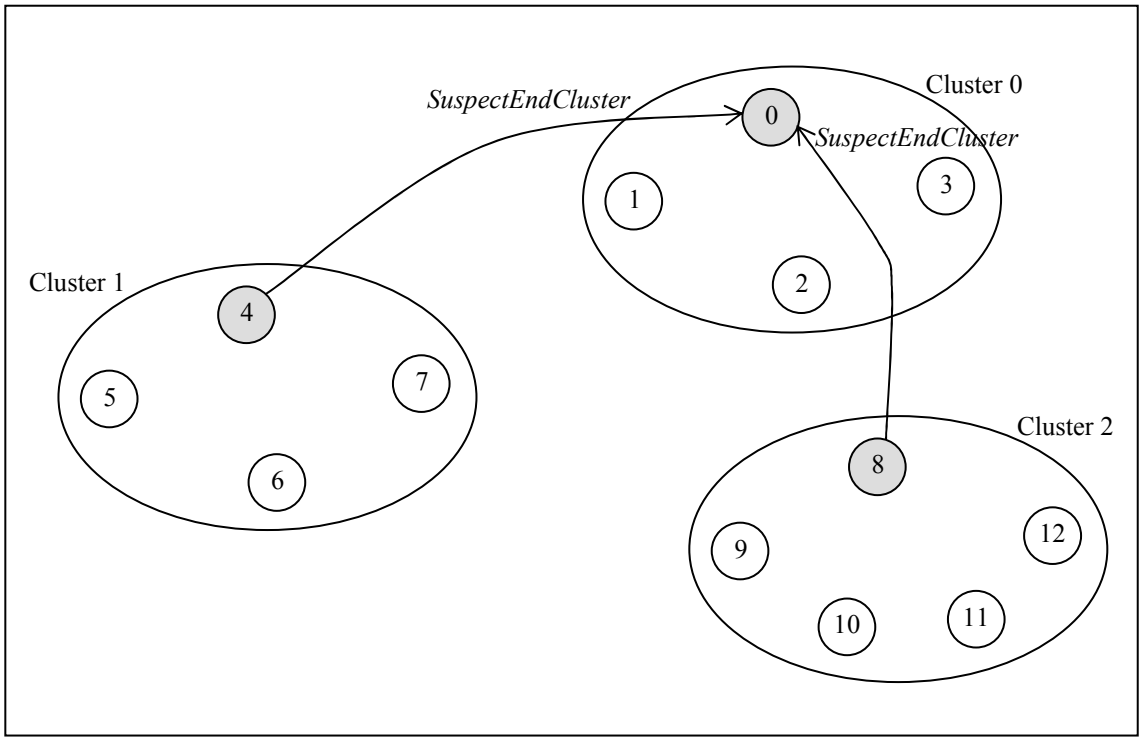


Figura 3.2.3 : Detecção de terminação entre clusters

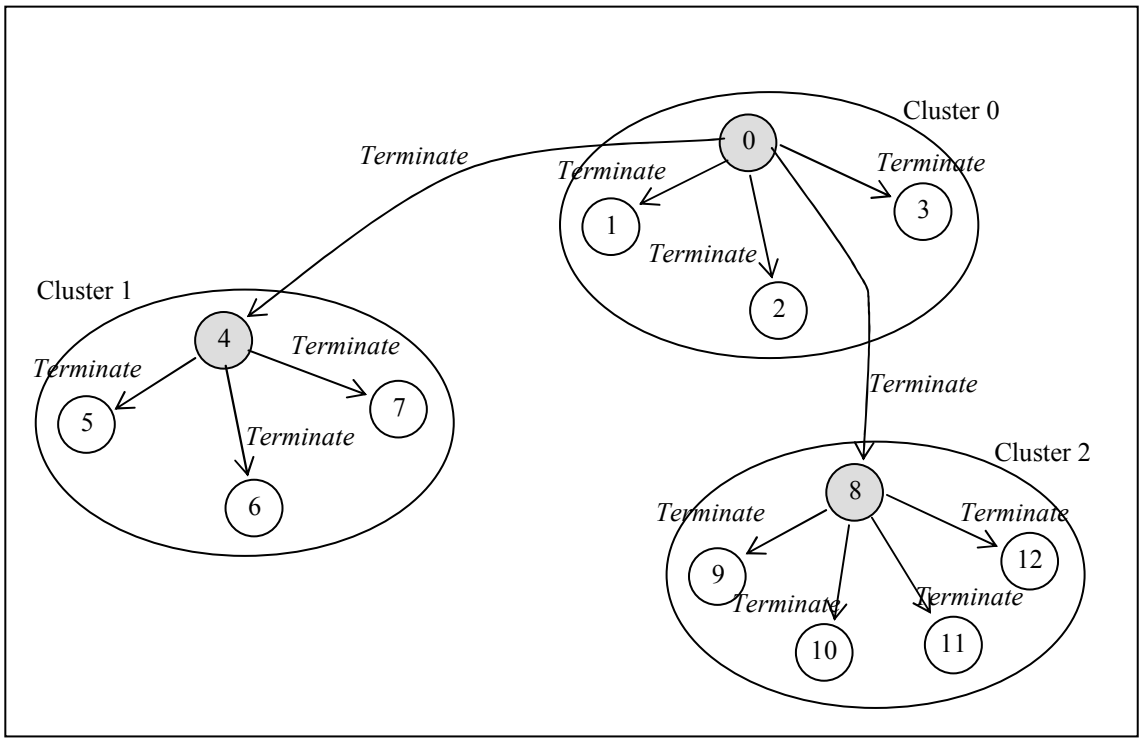


Figura 3.2.4 : Encerramento da aplicação.

Foram executados testes com uma variação simples do balanceamento de carga descrito anteriormente, onde o algoritmo não distingue processos em diferentes *clusters*, isto é, é assume-se que existe apenas um *cluster* lógico.

Como o escalonamento estático é o mesmo, os antecessores e sucessores da maior parte dos processos estão no mesmo *cluster* físico. Portanto, muitas mensagens ainda são *intra-clusters*. Esta estratégia foi chamada de *GlobalLB* (veja a Figura 3.2.5). Enquanto, a variante hierárquica descrita previamente foi chamada de *HierLB*.

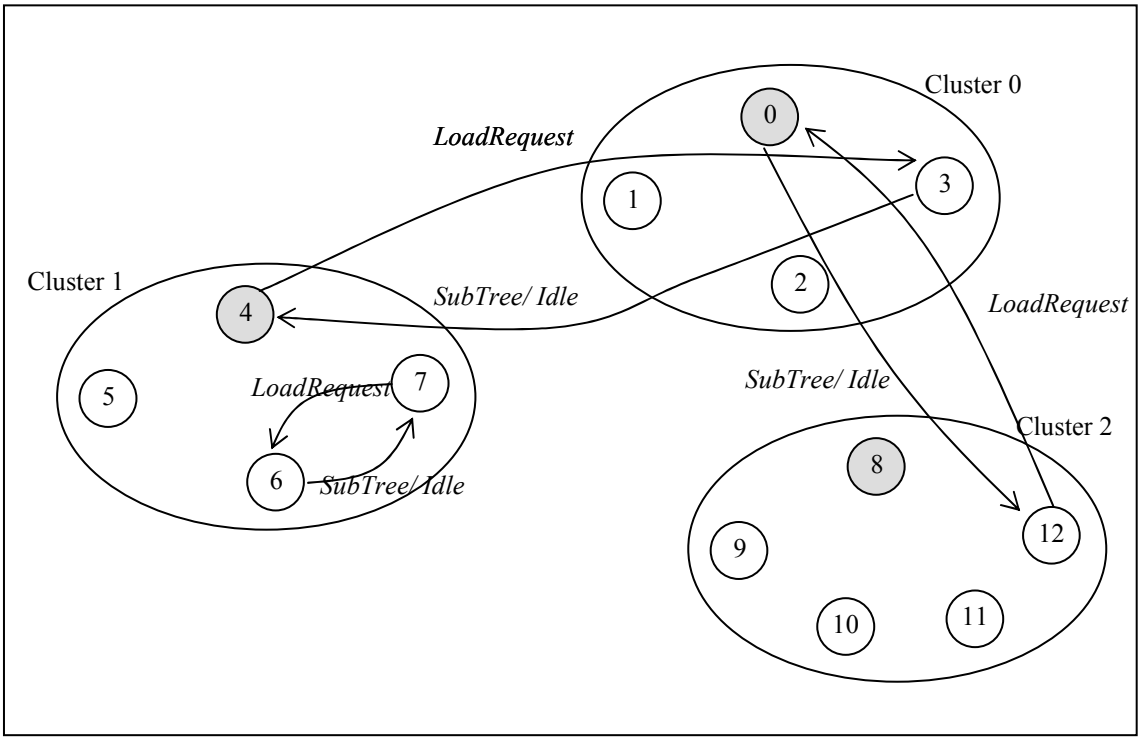


Figura 3.2.5 : Pedido de carga entre *clusters* na técnica *GlobalLB*

3.3 Difusão do limite *Primal*

O procedimento de difusão do limite *primal* é feito quando um processador encontra uma solução com valor melhor do que o limite *primal* até então conhecido (Figura 3.3.1). Este valor é difundido através da *Grid* porque poderá ser usado por outro processador para limitar o crescimento de sua enumeração de árvores. Um processo

executa uma difusão do limite *primal*, através da mensagem de *PrimalBound,* entre os processo em seu *cluster* e o líder do *cluster* é responsável pela difusão entre os outros líderes de *cluster* (Figura 3.3.2) enviando a mensagem de *PrimalBoundCluster*. Naturalmente, um processador pode receber uma mensagem *PrimalBound* contendo um valor que é pior do que seu limite *primal* local. Neste caso, a mensagem é ignorada.

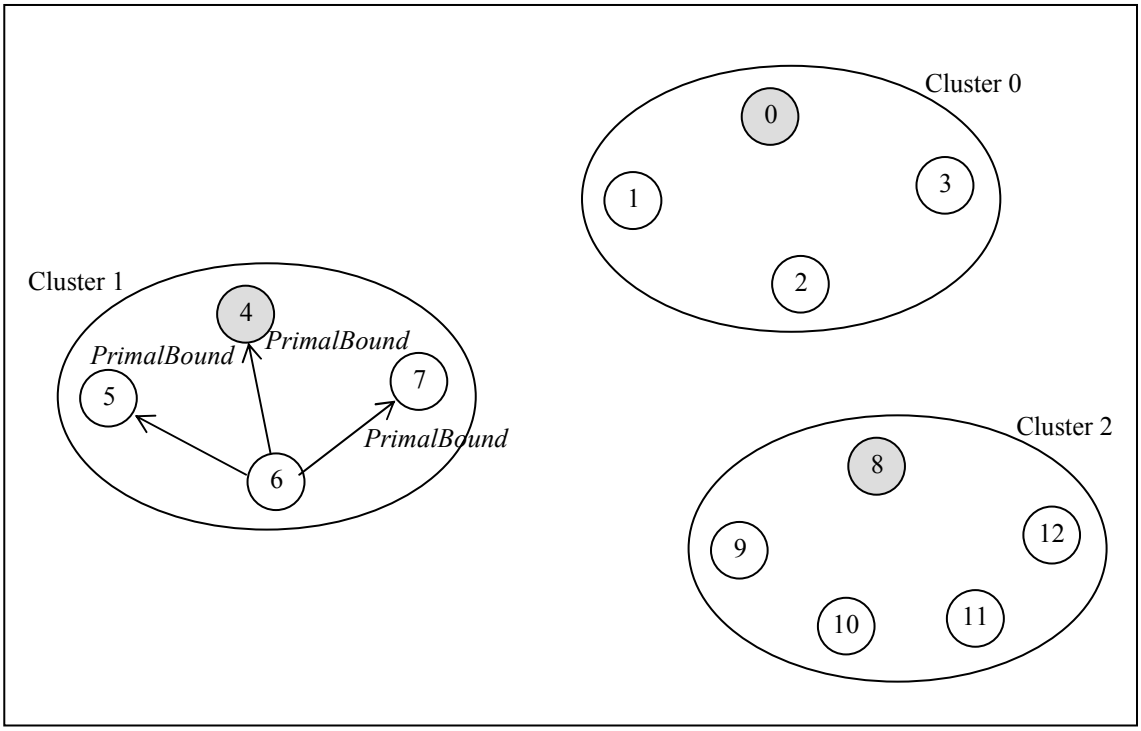


Figura 3.3.1 : Difusão do limite *primal* no próprio *cluster*

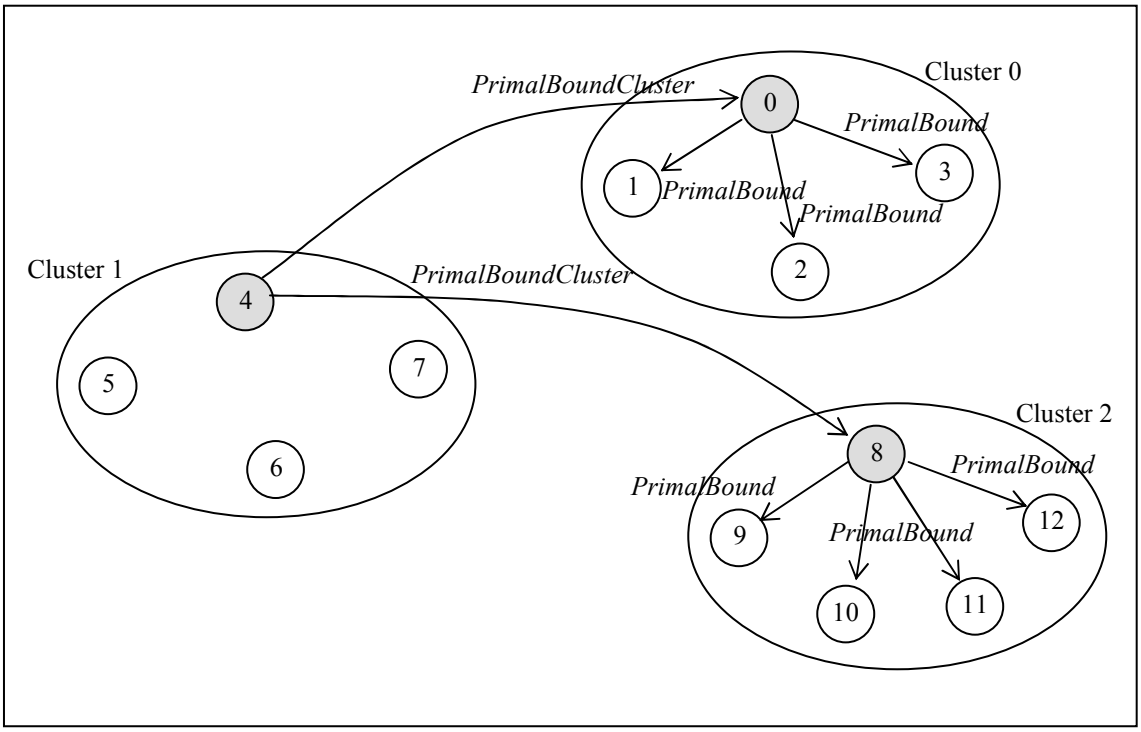


Figura 3.3.2 : Difusão do limite *primal* entre *clusters*

3.4 *Tolerância a falhas*

Um procedimento de tolerância a falha deve permitir que o algoritmo continue a execução apesar da presença de falhas. Para prover serviços seguros em meios não seguros, os trabalhos nesta área costumam adotar uma das seguintes estratégias:

(i) mecanismos de tolerância a falhas embutidos dentro da camada de *middleware*, ou;

(ii) mecanismos de tolerância a falhas dentro do algoritmo.

Embora a primeira abordagem seja bastante comum, optamos por usar a segunda que tem se apresentado como uma tendência atual e também devido ao fato de que resulta em procedimentos mais simples e eficientes, aumentando o desempenho de um modo geral [28, 29].

Neste trabalho, falhas de *crash* são resolvidas com técnicas de *checkpoint* e *rollback recovery* dentro do algoritmo. Um *checkpoint* global é um conjunto de *checkpoints* locais, um de cada processo que constitui o processamento distribuído. Um *checkpoint* global ou estado global é consistente se para todos os *checkpoints* locais, não existem mensagens (ou uma cadeia de mensagens) enviadas depois de um *checkpoint* local e recebidas antes de um outro.

Muitos algoritmos foram propostos para determinar *checkpoints* globais e eles são divididos em duas classes de acordo com o modo como são determinados: coordenados ou não coordenados [12]. Em algoritmos coordenados, a determinação de *checkpoints* locais é sincronizada de tal forma que o resultado de um *checkpoint* global é garantidamente consistente. Em algoritmos não coordenados, *checkpoints* locais são feitos independentemente. Então, quando um *checkpoint* global consistente é requerido, tem que ser construído a partir de um conjunto disponível de *checkpoints* locais. Como não existe a certeza de que um *checkpoint* global consistente pode ser realmente construído, este procedimento está sujeito ao efeito dominó ou *rollback* em cascata.

No algoritmo apresentado nesta dissertação os processos são independentes, ou seja, cada processo pode executar corretamente sua tarefa independente dos outros. Uma mensagem de limite *primal* pode afetar um processo reduzindo o número de nós resolvidos em um subproblema, mas não interfere no resultado. Se a solução ótima estiver nesse subproblema, esta será encontrada da mesma forma. Portanto, os processos não são realmente dependentes destas mensagens. Então, optamos por *checkpoints* não coordenados onde cada processo decide quando fazer seu *checkpoint* sem o risco de obter estados globais inconsistentes. Somente o último *checkpoint* precisa ser guardado nos processos.

Assim, cada processo *j* periodicamente envia uma mensagem contendo seu *checkpoint* para um outro processo no mesmo *cluster*, definido como *sp(j).* No recebimento de uma mensagem de *checkpoint,* o processo atualiza seu registro de *checkpoint*. Como o *branch-and-bound* empregado utiliza a estratégia de busca em profundidade, o *checkpoint* local é constituído de alguns poucos *bytes*, como ilustrado

no exemplo a seguir. Considere uma parte da sub-árvore de busca, mostrada na Figura 3.4.1. Os números abaixo dos nós indicam o vértice de *branch* escolhido, no filho esquerdo este vértice foi fixado em 0, no da direita em 1.

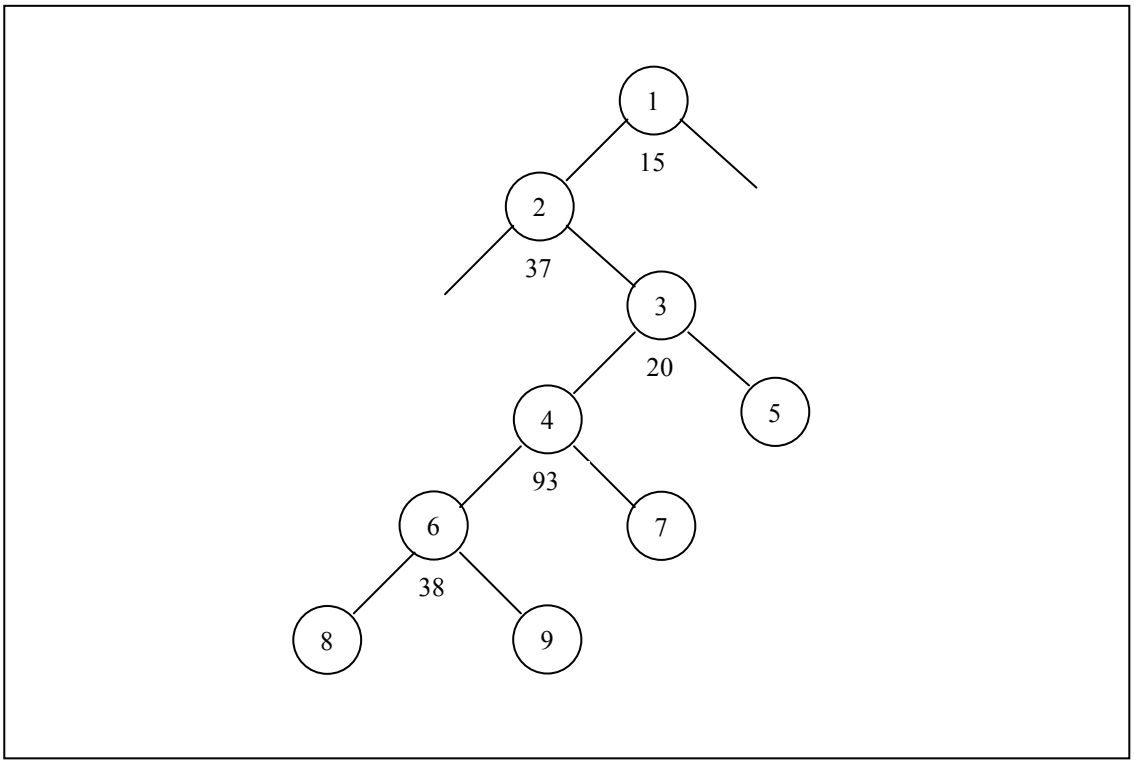


Figura 3.4.1: Parte de uma árvore de *Branch-and-Bound*

Suponha que o processador *j* recebeu de algum processador, o subproblema especificado por uma mensagem com a lista { (15 , 0) , (37 , 1) }. Em um dado momento, *j* já resolveu os nós 3, 4, 6, a sub-árvore inteira enraizada em 8 e já enviou o nó 5 para ser resolvido por um outro processador. Ele ainda não resolveu os nós 7 e 9. A mensagem de *checkpoint* que resume este seu estado enviado para *sp(j)* tem a seguinte informação: {(15; 0) ; (37 ; 1)); ((20 ; 0) ; (93 ; -1) ; (38, 1)} Então, suponha que depois do envio deste *checkpoint, j* falhe. Quando *sp(j)* nota que *j* falhou, ele sabe que deve resolver os subproblemas {(15 ; 0) ; (37 , 1) ; (20 ; 0) ; (93 ; 1)} e {(15; 0) ; (37 ; 1) ; (20 ; 0) ; (93 ; 0) ; (38 ; 1)}. Isto significa que durante a recuperação, o processo

com o *checkpoint* do nó que falhou assume o serviço que foi dado a ele. No caso de falha de líder, o processo também assume o serviço de tal líder.

De acordo com [17] o tempo apropriado entre consecutivos *checkpoints* em uma computação geral, t_{bc}, é dado pela fórmula:

$$t_{bc} = \sqrt{\frac{2t_r}{p_f}}$$

onde t_r é o tempo de gravação do *checkpoint* e p_f é a probabilidade de ocorrência de uma falha durante o tempo de execução da computação.

No caso do algoritmo proposto, *checkpoints* são feitos independentemente pelos processos. O tempo apropriado entre *checkpoints* consecutivos em um processo *j* deveria usar como t_r o tempo de preparo e gravação de um *checkpoint,* mais a sobrecarga de comunicação de cada mensagem de *checkpoint*, definida como o tempo em que um processador é empregado para a transmissão ou recepção dessa mensagem [6]. Note que parte deste tempo ocorre no transmissor e outra parte no receptor desta mensagem. É também assumido que *j* envia e recebe praticamente o mesmo número de mensagens de *checkpoints*. A probabilidade p_f neste caso deveria ser a probabilidade de falhas durante a execução da aplicação. Na seção 4.3, são apresentados os valores obtidos para neste trabalho.

A detecção de falha é baseada no mecanismo de *heartbeat* [25]. Cada processo *j* envia para seu sucessor *sp(j)* em intervalos de tempo regulares uma mensagem de *checkpoint* ou uma mensagem indicando que está vivo, que permite que o processo considere que seu antecessor tenha falhado depois de alguns segundos sem receber qualquer mensagem de tal processo. Quando a falha é detectada, o processo detector informa este fato para os outros processos naquele *cluster*. Eles devem atualizar *s(i),l(i)* e as definições de *sp(j)* e *pp(j)* para ignorarem o processo que falhou.

Um *cluster* inteiro em uma *Grid* pode também falhar. Isto é em geral resultado de uma falha de *link* que faz com que o *cluster* fique inalcançável. Para resolver tais eventos a estratégia acima é estendida a fim de detectar e recuperar falha de *cluster*, substituindo os processadores por *clusters* (representados pelos seus líderes), tendo *checkpoints* representando os estados dos *clusters* (ao invés de processos) e o mecanismo de *heartbeat* empregado entre os líderes dos *clusters.*

As seguintes mensagens são usadas pelo procedimento de tolerância a falhas no nível *intra-cluster*. A mensagem *Alive* apenas indica que um processo está vivo, a mensagem *CheckStore* contém o *checkpoint* de um processo e a mensagem *ProcessCrash* é difundida para informar que um processo falhou, como mostrado na Figura 3.4.2. A parte (a) mostra a fluxo inicial de mensagens de detecção de falhas. Na parte (b), o processo 7 descobre que o processo 6 teve falha e difunde a mensagem de *ProcessCrash*. A parte (c) mostra o novo fluxo de mensagens de tolerância a falhas.

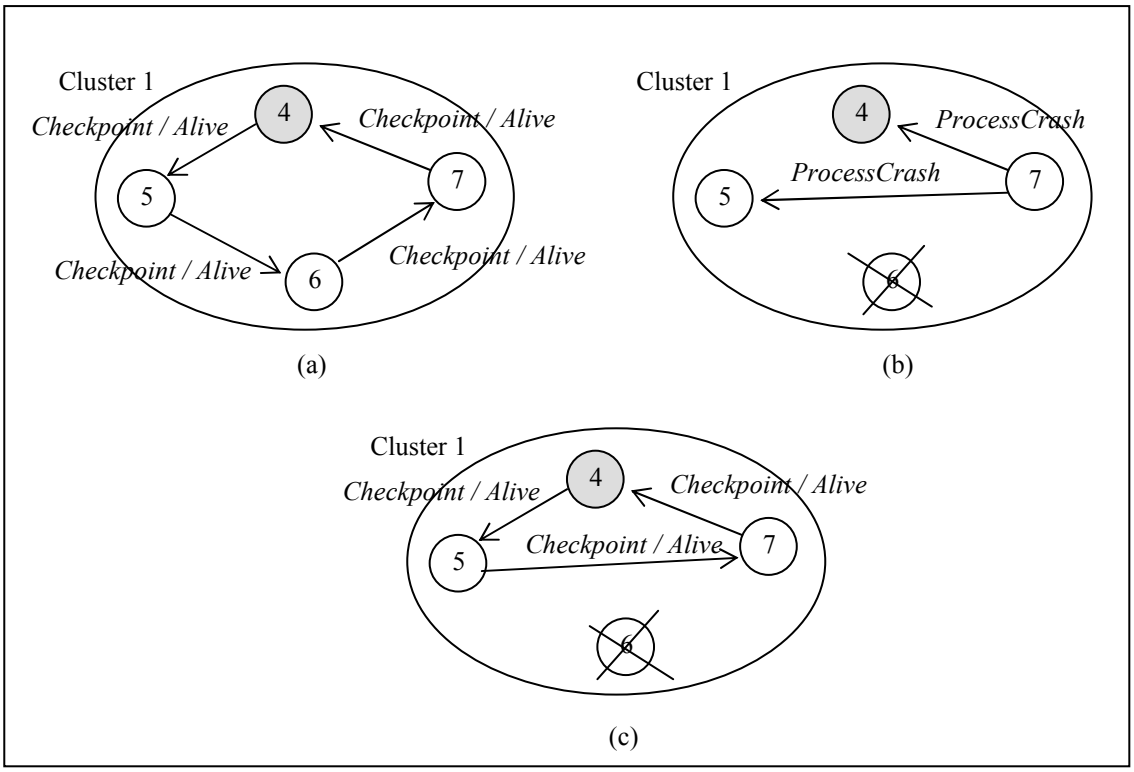


Figura 3.4.2 : Detecção de falha de processo

Quando uma falha resulta em um novo líder, este fato deverá ser informado para os líderes dos outros *clusters,* conforme mostra a Figura 3.4.3. O processo que detecta a falha do líder é o mesmo que armazena o seu *checkpoint* e terá a função, além do envio das mensagens de *ProcessCrash* para os membros do *cluster*, de informar aos líderes dos outros *clusters* da alteração do líder através da mensagem de *LeaderClusterCrash*. O processo que detectou a falha do líder será considerado o novo líder do *cluster.*

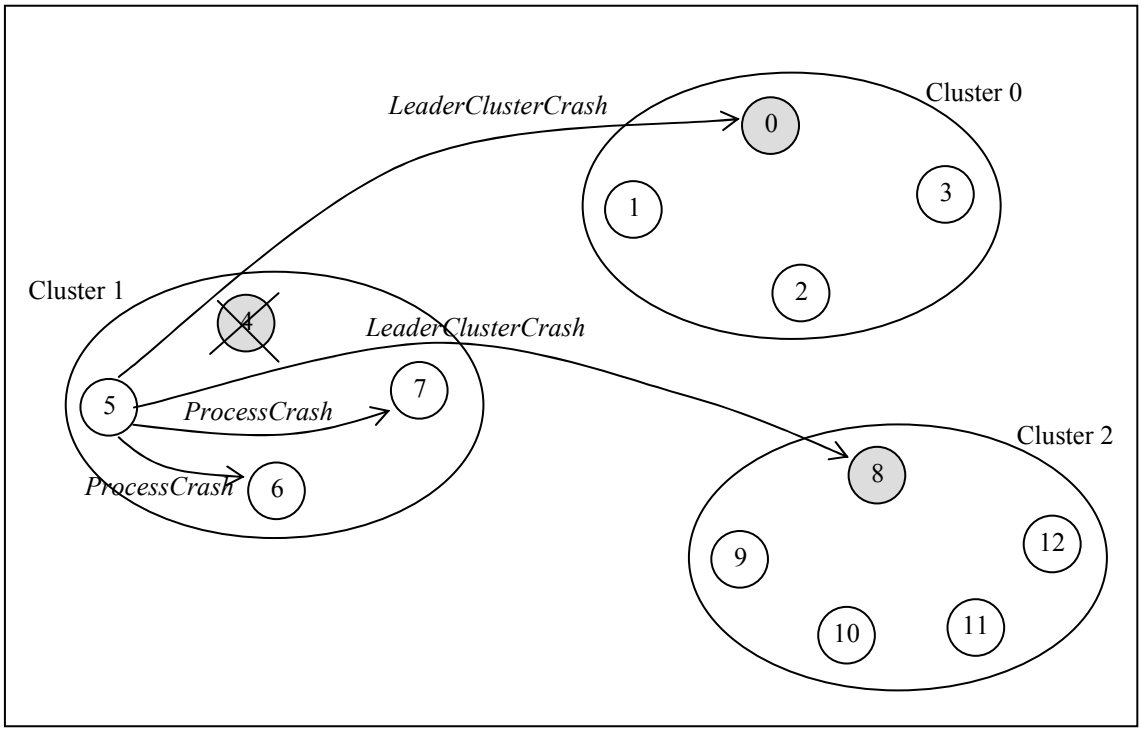


Figura 3.4.3 : Ocorrência de falha de líder de *cluster*

Foi incorporada uma nova mensagem dentro do procedimento de balanceamento de carga para que um processo *j*, que envia uma mensagem de *Subtree*, tenha a garantia de que sua sub-árvore seja processada mesmo na ocorrência de falha. O processo *k* que recebe a mensagem de *Subtree* responde com uma mensagem de *AckSubtree* imediatamente depois do envio de sua próxima mensagem de *checkpoint*. Quando *j* recebe tal mensagem, ele poderá ter certeza que a sub-árvore enviada será processada, ou por *k* ou pelo seu sucessor.

A fim de tratar falhas de *cluster*, foi utilizado um conjunto análogo de mensagens (*AliveCluster***,** *CheckpointCluster***,** *ClusterCrash*). A Figura 3.4.4 mostra o fluxo normal de mensagens de CheckpointCluster / Alive entre *clusters*. A Figura 3.4.5 apresenta a falha de um *cluster* e o informe aos outros *clusters* desta falha. E a Figura 3.4.6 apresenta o novo fluxo de mensagens entre os *clusters*.

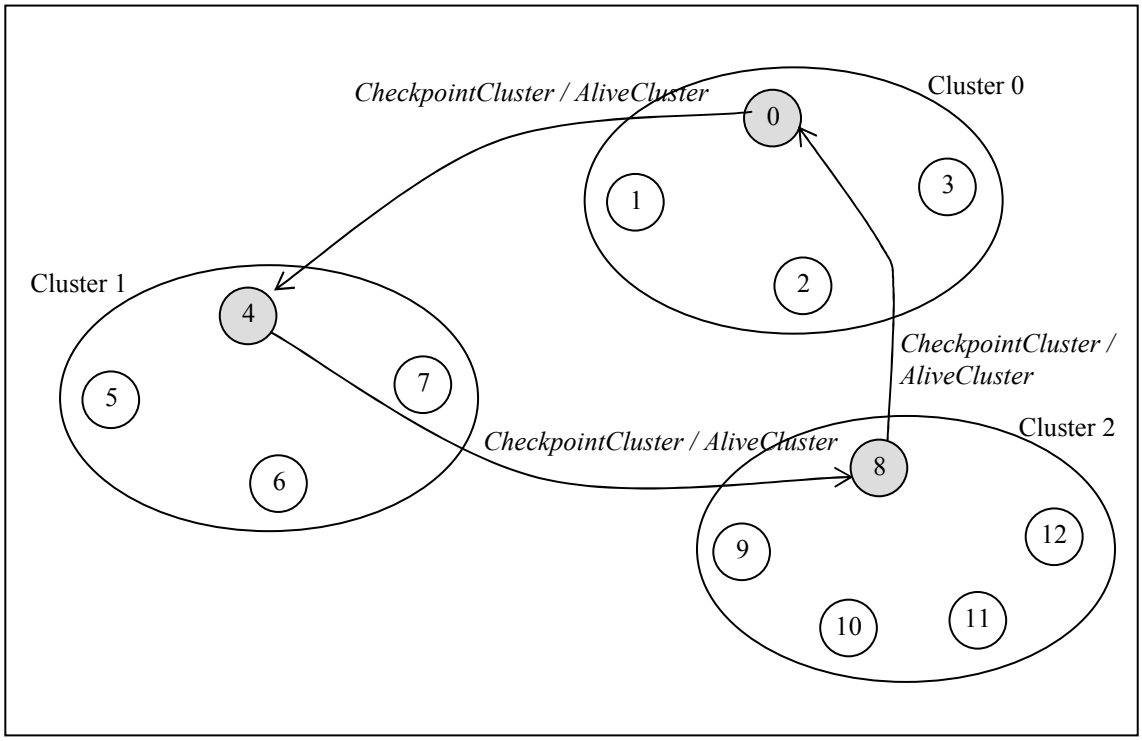


Figura 3.4.4 : Fluxo de mensagens de *CheckpointCluster* e *AliveCluster*

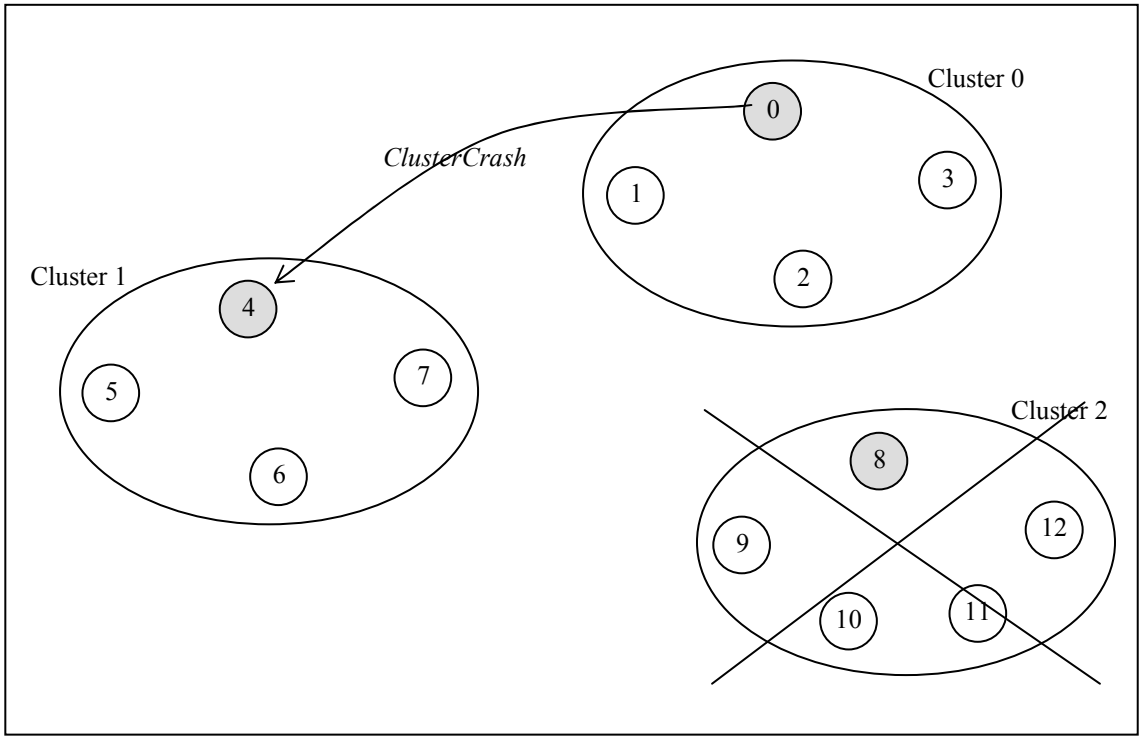


Figura 3.4.5 : Ocorrência de falha de *cluster*

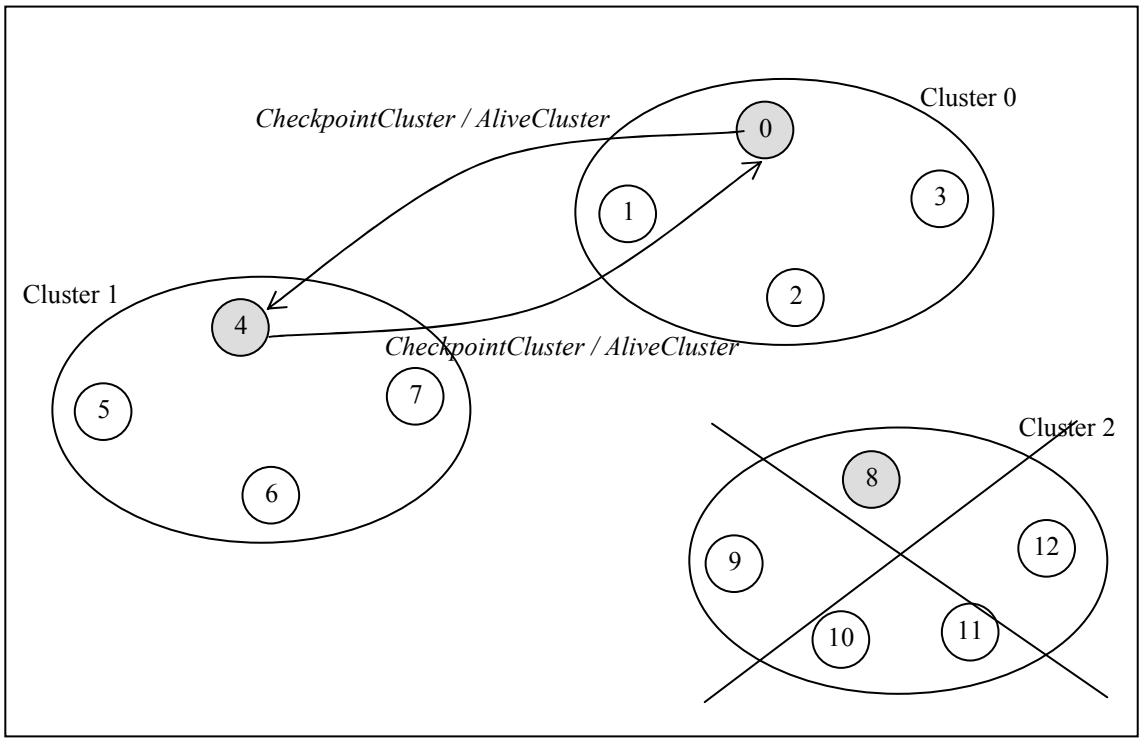


Figura 3.4.6 : Fluxo de mensagens após ocorrência de falha de um *cluster*

Capítulo 4

Resultados experimentais

O algoritmo paralelo proposto foi implementado em C++ e usa MPICH-G2 v1.2.5, uma versão habilitada para o *Globus* do MPI [13,19]. Os testes foram executados na *Grid* Sinergia, que é uma iniciativa de criar um ambiente computacional de *GRID* voltado para pesquisa entre universidades e institutos de pesquisa brasileiros, incluindo *clusters* das universidades UFF e PUC. Os testes aqui descritos foram executados usando as instâncias de incidência de *Steiner,* da i640-211 até i640-215, que constituem as 5 instâncias mais difíceis já resolvidas. Todas essas instâncias possuem 640 vértices, 4135 arestas e 50 terminais.

4.1 Testes em Clusters

Os primeiros testes são uma análise do balanceamento de carga proposto, em um único *cluster* e sem o procedimento de tolerância a falhas. Os resultados são apresentados na Tabela 4.4.1. Estes foram executados em um subconjunto de 16 processadores Pentium IV 2.4 GHz com 256Mb de memória RAM do *cluster* da UFF exclusivamente dedicados para a aplicação. O algoritmo seqüencial e a versão

distribuída somente com a distribuição inicial mas sem o procedimento de balanceamento de carga foram também executados.

Note que não foi usado nenhum limite superior externo (por exemplo, uma solução obtida por uma heurística) para iniciar o algoritmo. As soluções foram encontradas pelo próprio *branch-and-bound*. A coluna "Ótimo." na Tabela 4.1.1 apresenta as soluções ótimas encontradas.

O algoritmo seqüencial toma um número aleatório de decisões que afetam a ordem de busca na árvore no algoritmo. Executando cinco vezes com diferentes sementes, nós obtivemos uma média mais estável mostrada na coluna SEQ, que apresenta as médias dos tempos de relógio de parede (ou, simplesmente, nesta dissertação, tempo de relógio) obtidas nessas execuções, que usam diferentes sementes para o gerador aleatório interno do algoritmo. As próximas colunas apresentam o tempos de relógio para execuções dos algoritmos paralelos sem balanceamento de carga (PAR1) e com balanceamento de carga (PAR2). Os correspondentes *speedups* são dados nas colunas SP1 e SP2. A última coluna é a tradicional medida de eficiência paralela, ou seja, *speedup* dividido pelo número de processadores, de PAR2.

Os resultados apresentados na Tabela 4.1.1 claramente mostram que o procedimento de balanceamento de carga é realmente necessário nos algoritmos paralelos de *branch-and-bound*.

Podemos observar também que os *speedups* para instâncias individuais são muito irregulares. Essas anomalias, já bem conhecidas [4,21], são devidas à natureza do algoritmo *branch-and-bound*. Se uma boa solução é, por sorte, descoberta logo nas primeiras sub-árvores, as próximas sub-árvores podem ser mais bem "podadas" e o tempo de execução pode ser reduzido. Por outro lado, se o algoritmo não tiver sorte, pode perder uma boa parte do tempo explorando sub-árvores mais profundas sem nenhuma boa solução. Note que um *branch-and-bound* seqüencial está mais sujeito a ser afetado "por sorte" que sua versão paralela, onde, diversas partes da árvore estão sendo exploradas simultaneamente.

.

Instância	Ótimo	SEQ (seg/hora)		PAR1 (seg/hora)		PAR2 (seg/hora)		SP1	SP2	EFF
i640-211	11984	259708	72,14	50676	14,08	13265	3,68	5,12	19,58	1,22
i640-212	11795	20704	5,75	2614	0,73	526	0,15	7,92	39,36	2,46
i640-213	11879	18588	5,16	2985	0,83	1234	0,34	6,23	15,07	0,94
i640-214	11898	187267	52,02	28455	7,90	5434	1,51	6,58	34,47	2,15
i640-215	12081	79137	21,98	17023	4,73	4234	1,18	4,65	18,69	1,17

Tabela 4.1.1 : Testes em um cluster com as instâncias i640-211 a i640-215

4.2 Testes em *Grids*

Os testes seguintes foram executados em uma pequena *Grid*, usando algumas máquinas da PUC e UFF. Este pequeno ambiente nos permite realizar um monitoramento e análise dos procedimentos propostos. No entanto, como típico em *Grids*, essas máquinas não foram dedicadas à nossa aplicação. De fato, durante todos os nossos testes, as máquinas estiveram também executando um número desconhecido de aplicações de outros usuários. Portanto, é difícil medir o *speedup* e a tradicional eficiência paralela, já que o tempo de relógio de duas execuções diferentes da mesma instância varia significativamente dependendo de fatores fora de nosso controle.

No entanto, o MPICH-G2 implementa recebimento via *poll*. Isto significa que o processo realiza espera ocupada. Por exemplo, na nossa aplicação um processador *j*, tendo finalizado seu problema, envia um uma mensagem de *LoadRequest.* Enquanto espera pela resposta, *j* ainda consome praticamente toda sua fatia de tempo de CPU, da mesma forma que faria se estivesse resolvendo um problema. Portanto, o tempo de CPU de cada processo reflete com boa aproximação a sobrecarga de comunicação. A medida principal de desempenho usa o *tempo de aplicação normalizado,* definido como:

$$tnat = \sum_{J=0}^{m-1} \alpha(j) \,.\, tcpu(j)$$

Onde $\alpha(j)$ é um fator de normalização associado com a máquina na qual j está executando. Estes fatores são obtidos pela execução de uma instância de *benchmark* em cada máquina e comparando os tempos de execuções com o tempo gasto pelo mesmo *benchmark* na máquina padrão. Foi escolhido o processador Pentium IV 1.7GHz com 256Mb de RAM da PUC como a máquina padrão devido a estar em maior quantidade nos testes. Foi definido eficiência paralela como:

$$eff = \frac{tseq}{tnat}$$

Onde *tseq* é o tempo de CPU do algoritmo seqüencial executado na máquina padrão.

O segundo experimento é uma comparação das estratégias de balanceamento de carga global (GlobalLB) e hierárquico (HierLB), sem o procedimento de tolerância a falhas. Este foi avaliado no que foi chamado de *Grid16*, composto pelas máquinas da PUC (8 processadores Pentium IV 1.7 GHz) e UFF (8 processadores *Athlon* com tempos distintos, entre 1.2 e 2.0 GHz), com um poder computacional de cerca de 23,3 máquinas padrão.

O tempo médio de envio de mensagens internas ao cluster da UFF (mensagem MPI com 1 Kb) foi apurado em cerca de 0,6 ms, mesmo valor também encontrado no cluster na PUC, já o envio de mensagens de mesmo tipo entre os clusters da UFF e PUC tem seu valor médio em torno de 100ms, embora este valor possa variar demasiadamente de acordo com a utilização da rede.

A Tabela 4.2.1 contém os resultados usando o *GlobalLB*, enquanto a Tabela 4.2.2 contém o resultado usando o *HierLB*. A coluna de "Mensagens Totais" corresponde ao número total de mensagens enviadas por todos os processos, a coluna "Inter-cluster" corresponde às mensagens enviadas através da WAN. Colunas *tnat* e *eff* refletem o desempenho do algoritmo. É interessante comparar a coluna EFF da Tabela 4.1.1 com da coluna *eff* da Tabela 4.2.1, o algoritmo é o mesmo em ambos casos.

Instância	Seqüencial	Mensagens totais	Mensagens Inter-Cluster	*tnat*	*eff*	Tempo relógio (seg)
i640-211	545387	29923	7455	611060	0,89	47456
i640-212	43479	6053	2397	21993	1,98	1882
i640-213	38187	6477	2814	39926	0,96	3432
i640-214	393260	20129	5883	220288	1,79	13852
i640-215	166188	9670	2879	147427	1,13	8514

Tabela 4.2.1 : Resultados de execuções com o *GlobalLB*

Instância	Seqüencial	Mensagens totais	Mensagens Inter-Cluster	*tnat*	*eff*	Tempo relógio (seg)
i640-211	545387	19372	311	469295	1,16	37502
i640-212	43479	2574	111	17573	2,47	1622
i640-213	38187	9552	403	39966	0,96	3843
i640-214	393260	9306	260	172055	2,29	10255
i640-215	166188	11902	396	155654	1,07	12378

Tabela 4.2.2 : Resultados de execuções com o *HierLB*

É possível testemunhar a degradação do desempenho quando mudamos de um "ambiente ideal" (dedicado, com máquinas homogêneas conectadas por *links* de alta-velocidade) para uma *Grid* real. Comparando as tabelas 4.2.1 e 4.2.2 observamos que *HierLB* realmente reduz uma boa parte de sobrecarga de comunicação na *WAN*. Por outro lado, no *HierLB* um *cluster* inteiro pode ficar ocioso enquanto seu líder está pedindo por carga em um outro *cluster*. E ainda, como esta carga é compartilhada entre todos os processos em seu *cluster*, cada processo usualmente recebe sub-árvores menores. Então eles pedem por carga a seus vizinhos mais freqüentemente, podendo aumentar o número de mensagens *intra-cluster*, em alguns casos.

A técnica *GlobalLB* se mostrou mais eficiente em instâncias mais fáceis, enquanto *HierLB* tende a ser melhor em instâncias que necessitam de maior tempo de

processamento, como observado no gráfico da Figura 4.2.1, onde os resultados obtidos para as maiores instâncias (i640-211 e i640-214) são melhores na técnica *HierLB*.

Com relação a mensagens, fica claro que a quantidade de mensagens inter-cluster é muito superior na técnica *GlobalLB* em relação a *HierLB*. Esta superioridade aumenta acentuadamente em instâncias que necessitam de um maior tempo de execução, como visto no gráfico da Figura 4.2.2.

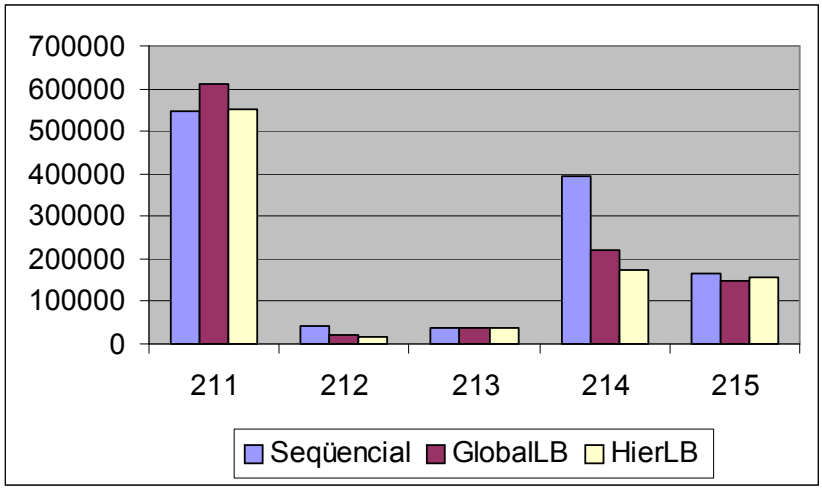


Figura 4.2.1 : Gráfico comparativo do *tnat*

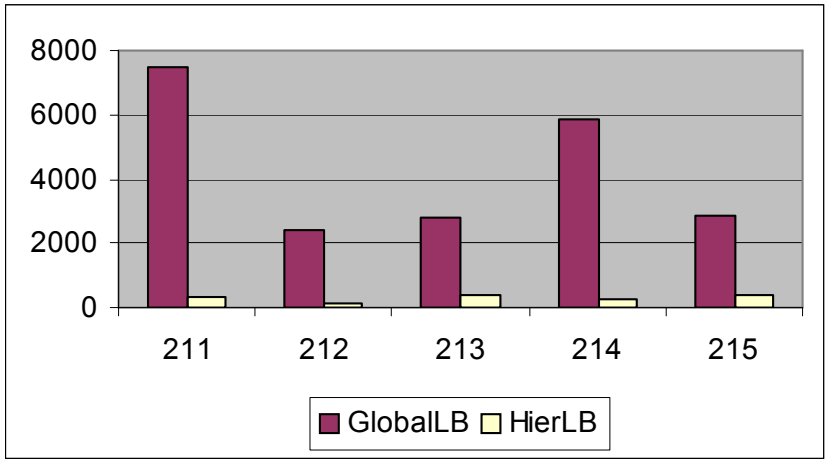


Figura 4.2.2 : Gráfico das mensagens *inter-clusters*

Nas Tabelas 4.2.3 a 4.2.12 são apresentados os principais resultados nas duas técnicas: *GlobalLB* e *HierLB* em cada uma das instâncias testadas. Nestas tabelas apresentamos, para cada processador, seu tempo de CPU, seu tempo de CPU normalizado, o total de nós resolvidos, tempo de CPU para execução da primeira sub-árvore e, finalmente, o número de nós desta primeira sub-árvore.

O tempo de CPU pode variar muito em função da utilização dos processadores por outras aplicações. Percebe-se, por exemplo, na Tabela 4.2.3 (*GlobalB*) que embora os três primeiros processadores, P0, P1 e P2, sejam iguais, P2 foi muito menos utilizado pela nossa aplicação.

Como já foi explicado na Seção 4.2, os processadores utilizados não possuem a mesma capacidade de processamento e por isso os tempos de CPU foram normalizados em relação ao processador padrão. Utilizamos como *benchmark* o programa seqüencial executado sobre uma mesma instância e adotando o valor de semente igual a um, evitando assim, o efeito aleatório do programa. Com isso, o algoritmo percorre sempre a mesma seqüência de sub-árvores, resultando no mesmo tempo de CPU, em várias execuções em um mesmo processador. Os tempos de CPU normalizados são apresentados na linha de "tempo normalizado" presente nas Tabelas 4.2.3 até 4.2.12.

Ainda nestas tabelas, são apresentados o total de nós executados por cada processador, bem como o tempo de CPU da primeira sub-árvore recebida e o total de nós processados nesta sub-árvore, antes de iniciar o balanceamento de carga. Observamos que a primeira sub-árvore dos primeiros processadores a receberem carga do Procedimento de Distribuição Inicial de carga correspondem a cerca de 70% a 90% do trabalho total destes processadores. Os últimos processadores a receberem carga do Procedimento de Distribuição Inicial de carga recebem sub-árvores de cerca de 50% de sua carga total. Podemos também ter uma estimativa de tempo de execução de cada nó da árvore que varia de 0,5 a 1 segundo em média.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Tempo de CPU	33342	45474	15407	14975	14585	45442	43650	22909	31784	31986	31339	30924	31043	29508	25768	30777	
Tempo de CPU normalizado	71820	97678	33116	20913	20369	47341	62836	13857	31784	31986	31339	30924	31043	29508	25768	30777	611059 *(tnat)*
Total de nós resolvidos	98442	82273	45944	29347	42716	94339	35737	38911	42924	43450	46275	50888	51977	45850	39843	50795	839711
Tempo CPU 1ª sub-árvore	31059	41122	13455	12363	12028	29332	35167	13948	23413	21715	20101	9446	20074	10092	17029	9741	
Nós 1º sub-árvore	91452	74241	40265	24268	35562	61794	28819	24346	31586	29992	30577	16471	36311	16181	27170	16827	585862

Tabela 4.2.3: Resultados obtidos por processo na técnica *GlobalLB* para instância i640-211

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Média
Tempo de CPU	17143	11825	14278	11409	11667	11031	11596	11431	24710	36937	24566	36116	36187	35888	35718	23358	
Tempo de CPU normalizado	36857	25424	30698	24529	25084	23717	24931	24577	24710	36937	24566	36116	36187	35888	35718	23358	469295 *(tnat)*
Total de nós resolvidos	46165	32389	39535	32612	32044	31987	37326	34478	30777	47399	31177	49234	46860	49156	48843	32679	622656
Tempo CPU 1ª sub-árvore	16692	11452	13405	9282	9229	8190	8190	7120	15265	22609	12215	20641	20321	15252	15701	11429	
Nós 1º sub-árvore	44965	31367	37103	26548	25323	23750	26392	21527	19013	29013	15502	28141	26293	20899	21471	15987	413294

Tabela 4.2.4 : Resultados obtidos por processo na técnica *HierLB* para instância i640-211

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Tempo de CPU	1781	1768	882	587	553	1537	1672	794	841	834	819	810	793	801	746	739	
Tempo de CPU normalizado	3836	3798	1896	820	772	1601	2407	480	841	834	819	810	793	801	746	739	21993 *(tnat)*
Total de nós resolvidos	4317	3035	2038	926	1208	2382	1171	829	716	1132	803	1087	891	1052	861	841	23289
Tempo CPU da 1ª sub-árvore	1750	1507	594	395	372	922	875	496	715	663	446	539	530	430	396	357	
Total de Nós da 1º sub-árvore	4285	2659	1326	657	801	1453	586	486	618	998	359	792	637	600	480	487	17224

Tabela 4.2.5: Resultados obtidos por processo na técnica *GlobalLB* para instância i640-212

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	1339	1380	672	438	405	1021	1234	568	769	789	761	720	717	736	708	705	
Tempo de CPU normalizado	2884	2964	1457	612	567	1064	1776	344	769	789	761	720	717	736	708	705	17573 *(tnat)*
Total de nós resolvidos	3008	2028	1492	647	775	1220	827	671	968	988	854	808	968	1049	939	921	18163
Tempo CPU da 1ª sub-árvore	1149	754	337	220	209	347	793	228	698	483	524	306	567	346	228	173	
Total de Nós da 1º sub-árvore	2602	1090	724	338	386	425	589	301	793	571	533	322	793	488	277	221	10453

Tabela 4.2.6: Resultados obtidos por processo na técnica *HierLB* para instância i640-212

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	2654	3207	1597	1048	1061	3006	3236	1492	1639	1626	1588	1549	1517	1512	1436	1383	
Tempo de CPU normalizado	5717	6889	3433	1464	1482	3132	4658	902	1639	1626	1588	1549	1517	1512	1436	1383	39927 *(tnat)*
Total de nós resolvidos	8102	7247	5248	2329	3612	7359	3019	2738	2496	2679	2537	2569	2335	2478	2292	2215	59255
Tempo CPU da 1ª sub-árvore	889	3201	1585	1000	416	2093	2652	1159	1195	1074	1038	1012	385	724	720	676	
Total de Nós da 1º sub-árvore	2331	7243	5206	2205	1162	4866	2460	2150	1794	1794	1667	1743	473	1276	1210	1124	38704

Tabela 4.2.7 : Resultados obtidos por processo na técnica *GlobalLB* para instância i640-213

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	2715	2946	1450	923	955	2472	2742	1208	1801	1897	1885	1874	1855	1848	1824	1813	
Tempo de CPU normalizado	5848	6328	3117	1289	1334	2575	3947	731	1801	1897	1885	1874	1855	1848	1824	1813	39966 *(tnat)*
Total de nós resolvidos	8134	5908	4336	1982	2841	4905	2252	2136	2928	3207	2915	3219	3236	3243	3220	3071	57533
Tempo CPU 1ª sub-árvore	957	2329	504	713	581	1329	846	616	1393	1894	1470	1444	1168	1431	859	1352	
Total de Nós da 1º sub-árvore	2867	4657	1508	1525	1747	2638	695	1089	2265	3206	2277	2480	2037	2522	1517	2300	35330

Tabela 4.2.8 : Resultados obtidos por processo na técnica *HierLB* para instância i640-213

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	13054	12865	5494	4225	9787	12333	11904	6495	12468	12755	12436	12342	12336	12258	12269	12380	
Tempo de CPU normalizado	28119	27634	11809	5900	13668	12848	17136	3929	12468	12755	12436	12342	12336	12258	12269	12380	220287 *(tnat)*
Total de nós resolvidos	32575	24755	14901	8172	26345	25218	9177	10148	14947	17973	15897	19192	16118	17332	18058	18673	289481
Tempo CPU da 1ª sub-árvore	11718	10374	4245	3030	7979	6130	3611	1157	6593	5441	4567	5415	7250	6790	2147	1592	
Total de Nós da 1º sub-árvore	29266	20863	11601	6093	21929	14189	2868	1884	7319	8483	6203	10175	10107	10997	3635	2801	168413

Tabela 4.2.9: Resultados obtidos por processo na técnica *GlobalLB* para instância i640-214

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	9619	9996	4898	3328	9641	9197	9451	4838	9712	9563	9481	9344	9386	9253	9199	9173	
Tempo de CPU normalizado	20720	21471	10528	4648	13464	9581	13605	2926	9712	9563	9481	9344	9386	9253	9199	9173	172055 *(tnat)*
Total de nós resolvidos	23448	17306	14039	6342	25985	17737	7564	7413	13458	12793	13665	13586	13384	14018	13009	14002	227749
Tempo CPU da 1ª sub-árvore	9254	5661	3567	963	3108	2246	5689	1123	6851	4200	5011	1594	3554	3757	5843	1784	
Total de Nós da 1º sub-árvore	22558	9819	10223	1836	8404	4331	4548	1721	9480	5619	7227	2318	5068	5691	8221	2723	109787

Tabela 4.2.10 : Resultados obtidos por processo na técnica *HierLB* para instância i640-214

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	8409	8329	4191	2772	8246	8163	7892	4031	8079	8263	8207	8105	8093	7988	8055	7934	
Tempo de CPU normalizado	18113	17891	9008	3871	11516	8504	11361	2438	8079	8263	8207	8105	8093	7988	8055	7934	147427 *(tnat)*
Total de nós resolvidos	26524	16387	11798	6087	27708	18810	7033	6806	13253	14416	14515	14559	13348	13503	13041	14019	231807
Tempo CPU da 1ª sub-árvore	8404	7505	2984	1784	5200	2028	5851	2167	5976	5094	5025	4501	4295	2637	3390	3212	
Total de Nós da 1º sub-árvore	26521	14523	7945	4032	17693	4730	5113	3766	9804	8855	8676	8402	7000	4079	4696	6061	141896

Tabela 4.2.11 : Resultados obtidos por processo na técnica *GlobalLB* para instância i640-215

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Total
Tempo de CPU	11547	11776	5846	3915	3898	11268	11749	5574	6643	6498	6419	6372	6166	6019	6009	5860	
Tempo de CPU normalizado	24873	25295	12566	5467	5444	11739	16913	3372	6643	6498	6419	6372	6166	6019	6009	5860	155654 *(tnat)*
Total de nós resolvidos	37231	23387	19941	7904	14006	27786	12117	10346	8620	10231	10297	10099	9429	10052	8895	9372	229713
Tempo CPU da 1ª sub-árvore	9362	6569	5836	3543	3674	6293	8180	2195	5422	2632	2002	2897	3984	3819	2323	2841	
Total de Nós da 1º sub-árvore	30179	13033	19908	7166	13187	15519	8436	4063	7036	4144	3212	4584	6092	6370	3439	4555	150923

Tabela 4.2.12: Resultados obtidos por processo na técnica *HierLB* para instância i640-215

Nas Tabelas 4.2.13 a 4.2.22 são apresentadas as mensagens enviadas nas duas técnicas *GlobalLB* e *HierLB* para cada uma das instâncias (Veja no apêndice o resumo das descrições das mensagens utilizadas no algoritmo). Nas tabelas do *GlobalLB*, as mensagens *inter-cluster* receberam o sufixo *Cluster,* como por exemplo nas mensagens *SuspectendCluster e IdleCluster*. Este acréscimo foi feito apenas para destacar nas tabelas as mensagens enviadas nos *links* externos, pois de fato não há necessidade de diferenciá-las, já que esta técnica considera a *Grid* como um único *cluster* lógico. Note também que as mensagens de pedido de carga predominam em relação às outras.

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	84	42	77	21	35	49	56	7	28	70	63	14	70	77	14	56	763
PrimalBound	15	0	45	0	0	0	15	0	0	7	7	0	0	0	14	7	110
SubTree	154	177	307	412	502	229	216	6	412	420	569	556	439	294	426	0	5119
Idle	133	120	132	257	342	229	96	19	825	783	659	725	625	435	295	7	5682
LoadRequest	24	189	239	236	495	987	697	457	7	461	844	887	1256	1384	1088	1543	10794
SuspectendCluster	96	48	88	24	40	56	64	8	32	80	72	16	80	88	16	64	872
PrimalBoundCluster	0	0	0	0	0	0	0	0	0	8	8	0	0	0	16	8	40
SubTreeCluster	3	5	21	39	78	185	356	531	6	5	21	8	27	24	42	131	1482
IdleCluster	8	9	12	32	68	137	284	575	26	31	33	58	65	89	112	149	1688
LoadRequestCluster	392	207	124	34	31	35	21	8	689	533	408	276	234	191	108	82	3373
TOTAL	909	797	1045	1055	1591	1907	1805	1611	2025	2398	2684	2540	2796	2582	2131	2047	29923

Tabela 4.2.13 : Relação mensagens enviadas por processo– Técnica *GlobalLB* – instância i640-211

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	49	56	91	28	56	70	77	42	14	35	21	7	84	77	98	70	875
PrimalBound	45	30	0	0	0	0	15	30	0	14	7	14	0	7	0	14	176
SubTree	6	27	55	88	62	38	32	3	12	24	31	38	27	40	31	0	514
Idle	12	9	21	49	79	60	32	14	102	83	81	77	76	48	41	8	792
LoadRequest	26	22	20	40	81	143	142	106	7	38	55	82	93	157	109	178	1299
SuspectendCluster	56	64	104	32	64	80	88	48	16	40	24	8	96	88	112	80	1000
PrimalBoundCluster	0	0	0	0	0	0	0	0	0	16	8	16	0	8	0	16	64
SubTreeCluster	5	7	15	26	10	12	24	36	2	0	0	0	0	2	0	0	139
IdleCluster	9	14	20	33	56	66	74	82	18	18	20	20	20	20	22	22	514
LoadRequestCluster	70	40	8	16	8	8	8	8	104	81	78	71	54	46	34	46	680
TOTAL	278	269	334	312	416	477	492	369	275	349	325	333	450	493	447	434	6053

Tabela 4.2.14 : Relação mensagens enviadas por processo– Técnica *GlobalLB* – instância i640-212

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	21	42	28	42	42	14	49	7	56	49	35	63	70	70	77	28	693
PrimalBound	45	30	45	15	0	60	30	0	0	0	14	7	0	7	0	0	253
SubTree	3	7	10	27	34	30	33	4	38	56	79	73	78	88	59	0	619
Idle	10	8	12	15	22	24	18	11	79	75	85	104	110	92	71	7	743
LoadRequest	21	15	8	14	30	49	56	68	7	49	70	102	138	195	238	295	1355
SuspectendCluster	24	48	32	48	48	16	56	8	64	56	40	72	80	80	88	32	792
PrimalBoundCluster	0	0	0	0	0	0	0	0	0	0	16	8	0	8	0	0	32
SubTreeCluster	1	3	2	13	11	12	25	63	0	4	12	11	20	34	47	123	381
IdleCluster	8	8	10	9	23	31	39	58	16	15	19	32	42	60	91	134	595
LoadRequestCluster	609	25	8	8	8	8	8	8	92	51	42	37	34	21	26	29	1014
TOTAL	742	186	155	191	218	244	314	227	352	355	412	509	572	655	697	648	6477

Tabela 4.2.15 : Relação mensagens enviadas por processo– Técnica *GlobalLB* – instância i640-213

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	77	98	91	42	84	7	70	14	35	63	49	21	56	70	28	98	903
PrimalBound	30	15	0	0	0	0	0	0	0	7	7	28	0	0	7	14	108
SubTree	53	74	157	193	195	116	99	7	243	423	365	283	252	266	282	0	3008
Idle	78	60	74	122	131	72	77	16	545	459	530	437	419	368	218	7	3613
LoadRequest	26	88	85	150	283	413	185	287	7	336	489	787	624	673	1031	1150	6614
SuspectendCluster	88	112	104	48	96	8	80	16	40	72	56	24	64	80	32	112	1032
PrimalBoundCluster	0	0	0	0	0	0	0	0	0	8	8	32	0	0	8	16	72
SubTreeCluster	8	6	35	46	60	108	186	362	2	8	9	8	20	22	21	45	946
IdleCluster	8	15	19	52	90	147	235	367	24	25	33	42	48	70	87	108	1370
LoadRequestCluster	241	147	58	49	45	35	8	8	479	400	244	209	162	138	132	108	2463
TOTAL	609	615	623	702	984	906	940	1077	1375	1801	1790	1871	1645	1687	1846	1658	20129

Tabela 4.2.16 : mensagens enviadas por processo– Técnica *GlobalLB* – instância i640-214

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	70	77	63	91	84	98	56	7	35	7	42	49	98	21	14	28	840
PrimalBound	45	15	0	0	0	0	15	0	0	7	21	7	0	35	21	0	166
SubTree	12	25	40	116	66	64	79	0	85	99	108	147	195	225	194	0	1455
Idle	9	11	20	39	55	39	31	10	193	188	185	169	186	173	126	7	1441
LoadRequest	9	16	24	31	125	127	114	163	7	97	157	221	248	408	536	606	2889
SuspectendCluster	80	88	72	104	96	112	64	8	40	8	48	56	112	24	16	32	960
PrimalBoundCluster	0	0	0	0	0	0	0	0	0	8	24	8	0	40	24	0	104
SubTreeCluster	2	10	17	19	29	23	73	195	0	0	0	0	0	0	1	1	370
IdleCluster	8	10	18	33	48	72	95	134	11	11	11	11	11	11	10	12	506
LoadRequestCluster	19	16	16	8	8	8	8	8	304	116	88	92	69	68	61	50	939
TOTAIS	254	268	270	441	511	543	535	525	675	541	684	760	919	1005	1003	736	9670

Tabela 4.2.17 : Relação mensagens enviadas por processo– Técnica *GlobalLB* – instância i640-215

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	1	1	1	1	1	1	1	1	152	62	62	62	62	62	62	62	594
PrimalBound	61	7	0	0	0	0	0	7	22	14	0	14	7	0	14	14	160
SubTree	30	27	24	66	123	148	134	85	372	394	500	532	668	623	407	60	4193
Idle	51	19	22	26	41	52	79	79	294	435	587	709	754	803	632	413	4996
LoadRequest	186	141	58	33	66	148	170	197	992	604	588	721	990	1242	1580	1217	8933
Awake	0	0	0	0	0	0	0	0	39	65	33	48	0	0	0	0	185
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	5	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	12
SubTreeCluster	107	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	1	0	0	0	0	0	0	0	147	0	0	0	0	0	0	0	148
Notnow	40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	40
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	484	195	105	126	231	349	384	369	2027	1574	1770	2086	2481	2730	2695	1766	19372

Tabela 4.2.18 : Relação de mensagens enviadas por processo– Técnica *HierLB* – instância i640-211

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	32	6	6	6	6	6	6	6	1	1	1	1	1	1	1	1	82
PrimalBound	56	7	7	0	0	0	0	7	49	7	14	14	7	0	7	7	182
SubTree	29	38	40	53	62	21	26	8	11	30	28	25	12	30	29	9	451
Idle	32	51	57	73	87	82	67	55	7	12	29	28	23	21	22	14	660
LoadRequest	168	60	63	60	81	127	99	94	37	10	19	42	52	39	57	67	1075
Awake	3	4	3	3	0	0	0	0	0	0	0	0	0	0	0	0	13
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	6	0	0	0	0	0	0	0	6	0	0	0	0	0	0	0	12
SubTreeCluster	1	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	19
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	47	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	48
Notnow	0	0	0	0	0	0	0	0	28	0	0	0	0	0	0	0	28
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	376	166	176	195	236	236	198	170	160	60	91	110	95	91	116	98	2574

Tabela 4.2.19 : Relação de mensagens enviadas por processo– Técnica *HierLB* – instância i640-212

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	141	69	69	69	69	69	69	69	1	1	1	1	1	1	1	1	632
PrimalBound	56	28	0	14	0	0	0	0	56	7	21	0	0	14	0	7	203
SubTree	193	117	80	64	84	75	105	89	3	11	28	25	23	38	38	39	1012
Idle	283	311	341	363	387	408	408	449	8	7	12	30	34	37	40	35	3153
LoadRequest	1118	626	389	346	248	245	300	239	130	11	11	17	35	43	70	84	3912
Awake	67	78	42	50	0	0	0	0	0	0	0	0	0	0	0	0	237
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	7	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0	16
SubTreeCluster	1	0	0	0	0	0	0	0	104	0	0	0	0	0	0	0	105
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	191	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	192
Notnow	0	0	0	0	0	0	0	0	86	0	0	0	0	0	0	0	86
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	2059	1229	921	906	788	797	882	846	400	37	73	73	93	133	149	166	9552

Tabela 4.2.20 : Relação de mensagens enviadas por processo– Técnica *HierLB* – instância i640-213

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	1	1	1	1	1	1	1	1	80	38	38	38	38	38	38	38	354
PrimalBound	63	0	0	7	0	0	0	0	14	0	7	14	7	21	0	0	133
SubTree	4	16	37	85	45	51	91	55	195	183	187	180	196	179	163	54	1721
Idle	10	8	15	31	34	26	27	35	215	267	310	328	335	344	327	271	2583
LoadRequest	129	16	17	28	104	78	76	115	557	404	388	402	411	437	470	511	4143
Awake	0	0	0	0	0	0	0	0	24	40	23	25	0	0	0	0	112
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	2	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	9
SubTreeCluster	59	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	59
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	1	0	0	0	0	0	0	0	123	0	0	0	0	0	0	0	124
Notnow	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	64
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	335	41	70	152	184	156	195	206	1217	932	953	987	987	1019	998	874	9306

Tabela 4.2.21 : : Relação de mensagens enviadas por processo– Técnica *HierLB* – instância i640-214

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	1	1	1	1	1	1	1	1	153	73	73	73	73	73	73	73	672
PrimalBound	63	21	21	14	0	0	7	0	49	7	0	0	0	7	21	0	210
SubTree	40	11	9	29	55	50	128	172	242	137	118	134	129	87	69	15	1425
Idle	54	10	16	20	36	48	56	97	334	394	429	461	507	541	483	453	3939
LoadRequest	321	183	11	12	24	64	75	134	941	620	482	426	444	386	554	376	5053
Awake	0	0	0	0	0	0	0	0	44	71	37	55	0	0	0	0	207
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	7	0	0	0	0	0	0	0	6	0	0	0	0	0	0	0	13
SubTreeCluster	113	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	113
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	1	0	0	0	0	0	0	0	189	0	0	0	0	0	0	0	190
Notnow	76	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	76
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	678	226	58	76	116	163	267	404	1960	1302	1139	1149	1153	1094	1200	917	11902

Tabela 4.2.22 : Relação de mensagens enviadas por processo– Técnica *HierLB* – instância i640-215

As Tabelas 4.2.23 e 4.2.24 fornecem detalhes sobre as primeiras 25 sub-árvores resolvidas por cada processo durante a execução da instância i640-212, no algoritmo *HierLB*. As Colunas N apresentam o tamanho das sub-árvores, em termos de nós. As Colunas T fornecem o tempo de CPU normalizado gasto para resolver cada sub-árvore (incluindo o tempo ocioso enquanto espera-se para receber carga). As Colunas % fornecem o percentual acumulativo de tempo de CPU até então gasto pelo processo, em relação ao tempo total de CPU. As células imediatamente abaixo de cada N correspondem ao número total de nós resolvidos, abaixo de cada T o tempo total e abaixo de cada % o número total de sub-árvores recebidas. Através desta tabela podemos observar a quantidade de carga e o tempo de execução para cada sub-árvore recebida.

O desempenho do algoritmo proposto é muito dependente do tamanho das sub-árvores recebidas por cada processo. Sub-árvores maiores minimizam tempos de espera. Obter pequenas sub-árvores é inevitável perto do fim da execução, mas é melhor evitar tais sub-árvores no meio da execução. O procedimento de balanceamento de carga sempre envia nós não resolvidos do menor nível nas mensagens de *Subtree*. Tais nós mais provavelmente definem sub-árvores maiores. Mas isto pode ser melhorado, pois algumas vezes um processador recebe seqüências de sub-árvores com menos de que 10 nós.

Como a técnica *HierLB* apresentou os melhores resultados para as instâncias mais difíceis, resolvemos incluir neste modelo os procedimentos de tolerância a falhas, cujos resultados são descritos na seção seguinte.

	Processo 0			Processo 1			Processo 2			Processo 3			Processo 4			Processo 5			Processo 6			Processo 7		
SUB	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%
Totais	3008	1339		2028	1380		1492	672		647	438		775	405		1220	1021		827	1234		671	568	
0	2602	1149	85,81%	1090	754	54,64%	724	337	50,15%	338	220	50,23%	386	209	51,60%	425	347	33,99%	589	793	64,26%	301	228	40,14%
1	1	1	85,88%	611	385	82,54%	51	28	54,32%	56	39	59,13%	3	1	51,85%	197	134	47,11%	1	4	64,59%	183	131	63,20%
2	67	32	88,27%	148	122	91,38%	1	1	54,46%	30	29	65,75%	15	10	54,32%	61	61	53,09%	12	23	66,45%	7	11	65,14%
3	1	2	88,42%	1	1	91,45%	3	1	54,61%	9	16	69,41%	16	7	56,05%	77	85	61,41%	11	25	68,48%	1	1	65,32%
4	3	2	88,57%	5	4	91,74%	1	0.5	54,61%	4	3	70,09%	13	17	60,25%	5	16	62,98%	2	5	68,88%	7	15	67,96%
5	1	1	88,65%	131	81	97,61%	1	0.5	54,61%	39	15	73,52%	1	1	60,49%	3	3	63,27%	1	11	69,77%	1	7	69,19%
6	13	6	89,10%	1	1	97,68%	1	1	54,76%	22	14	76,71%	1	1	60,74%	5	2	63,47%	1	32	72,37%	1	5	70,07%
7	1	1	89,17%	1	2	97,83%	1	1	54,91%	12	8	78,54%	1	12	63,70%	7	26	66,01%	1	2	72,53%	1	2	70,42%
8	1	0.7	89,17%	11	8	98,41%	163	82	67,11%	12	6	79,91%	19	16	67,65%	1	1	66,11%	1	18	73,99%	3	9	72,01%
9	17	10	89,92%	18	15	99,49%	27	11	68,75%	13	11	82,42%	1	1	67,90%	5	25	68,56%	7	12	74,96%	3	2	72,36%
10	5	2	90,07%	6	2	99,64%	132	39	74,55%	1	1	82,65%	16	7	69,63%	1	1	68,66%	1	1	75,04%	5	6	73,42%
11	1	1	90,14%	2	2	99,78%	31	15	76,79%	1	0.5	82,65%	6	13	72,84%	1	1	68,76%	12	21	76,74%	27	19	76,76%
12	1	1	90,22%	1	1	99,86%	81	28	80,95%	65	39	91,55%	7	5	74,07%	1	43	72,97%	1	2	76,90%	3	4	77,46%
13	206	95	97,31%	2	2	100,00%	11	8	82,14%	1	1	91,78%	3	2	74,57%	1	1	73,07%	4	7	77,47%	3	3	77,99%
14	1	1	97,39%				101	55	90,33%	24	12	94,52%	1	1	74,81%	23	20	75,02%	1	1	77,55%	3	3	78,52%
15	1	0.7	97,39%				17	6	91,22%	5	7	96,12%	1	1	75,06%	1	3	75,32%	1	1	77,63%	1	1	78,70%
16	59	25	99,25%				19	11	92,86%	1	1	96,35%	39	13	78,27%	35	27	77,96%	1	1	77,71%	1	1	78,87%
17	1	0.7	99,25%				65	25	96,58%	1	1	96,58%	1	1	78,52%	1	2	78,16%	20	44	81,28%	1	1	79,05%
18	1	1	99,33%				5	2	96,88%	1	1	96,80%	13	5	79,75%	1	1	78,26%	4	9	82,01%	11	12	81,16%
19	1	0.8	99,33%				9	2	97,17%	1	2	97,26%	29	6	81,23%	1	1	78,35%	1	2	82,17%	1	2	81,51%
20	1	0.9	99,33%				3	2	97,47%	1	1	97,49%	10	5	82,47%	1	2	78,55%	1	1	82,25%	1	3	82,04%
21	1	0.5	99,33%				2	1	97,62%	2	3	98,17%	1	0.6	82,47%	1	2	78,75%	23	29	84,60%	1	1	82,22%
22	1	1	99,40%				16	5	98,36%	3	2	98,63%	12	5	83,70%	13	6	79,33%	7	8	85,25%	1	2	82,57%
23	8	4	99,70%				12	6	99,26%	3	2	99,09%	1	1	83,95%	3	1	79,43%	3	5	85,66%	1	1	82,75%
24	1	0.7	99,70%				10	3	99,70%	1	1	99,32%	1	0.5	83,95%	21	12	80,61%	1	5	86,06%	54	49	91,37%
25	1	0.8	99,70%				1	0.6	99,70%	1	1	99,54%	30	11	86,67%	8	6	81,19%	1	2	86,22%	10	14	93,84%

Tabela 4.2.23 : Relação de quantidade de nós e tempo de execução nas 25 primeiras sub-árvores, processos 0 a 7, instância i640-212

	Processo 8			Processo 9			Processo 10			Processo 11			Processo 12			Processo 13			Processo 14			Processo 15		
SUB	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%	N	T	%
	968	767		988	789		854	761		808	720		968	717		1049	736		939	708		921	705	
0	793	698	90,91%	571	483	61,22%	533	524	68,86%	322	306	42,50%	793	567	79,08%	488	346	47,01%	277	228	32,20%	221	173	24,54%
1	8	2	91,23%	143	109	75,03%	75	70	78,06%	195	173	66,53%	8	8	80,20%	161	112	62,23%	145	112	48,02%	69	59	32,91%
2	3	8	92,31%	138	79	85,04%	41	35	82,65%	3	25	70,00%	3	9	81,45%	102	82	73,37%	56	30	52,26%	115	93	46,10%
3	31	10	93,66%	136	118	100,00%	45	34	87,12%	7	16	72,22%	31	25	84,94%	16	18	75,82%	46	48	59,04%	5	15	48,23%
4	1	4	94,22%				10	9	88,30%	1	4	72,78%	1	5	85,63%	5	9	77,04%	37	34	63,84%	13	23	51,49%
5	21	3	94,65%				7	6	89,09%	7	6	73,61%	21	14	87,59%	7	5	77,72%	83	52	71,19%	21	7	52,48%
6	1	2	94,93%				25	17	91,33%	1	1	73,75%	1	1	87,73%	1	16	79,89%	35	29	75,28%	17	13	54,33%
7	11	6	95,82%				97	51	98,03%	1	1	73,89%	11	10	89,12%	1	1	80,03%	19	21	78,25%	3	8	55,46%
8	1	2	96,13%				13	8	99,08%	53	46	80,28%	1	5	89,82%	24	15	82,07%	12	6	79,10%	27	10	56,88%
9	5	0.7	96,23%				3	2	99,34%	8	6	81,11%	5	5	90,52%	145	54	89,40%	1	1	79,24%	7	4	57,45%
10	7	0.7	96,33%				4	4	99,87%	1	2	81,39%	7	8	91,63%	1	1	89,54%	7	4	79,80%	3	3	57,87%
11	3	9	97,52%				1	1	100,00	31	17	83,75%	3	2	91,91%	4	5	90,22%	1	2	80,08%	25	15	60,00%
12	1	1	97,75%							1	4	84,31%	1	1	92,05%	1	4	90,76%	1	1	80,23%	15	8	61,13%
13	2	4	98,30%							22	15	86,39%	2	1	92,19%	92	67	99,86%	1	1	80,37%	3	2	61,42%
14	1	4	98,82%							3	4	86,94%	1	0	92,19%	1	1	100,00%	1	1	80,51%	21	15	63,55%
15	1	1	98,98%							15	10	88,33%	1	1	92,33%				15	9	81,78%	13	5	64,26%
16	2	72	100,00							2	4	88,89%	2	6	93,17%				17	13	83,62%	19	13	66,10%
17	1									4	5	89,58%	1	2	93,44%				79	47	90,25%	7	6	66,95%
18	1									13	5	90,28%	1	1	93,58%				1	1	90,40%	5	5	67,66%
19	1									15	14	92,22%	1	1	93,72%				1	1	90,54%	29	31	72,06%
20	3									33	15	94,31%	3	2	94,00%				1	1	90,68%	13	7	73,05%
21	3									15	11	95,83%	3	3	94,42%				1	0.7	90,68%	19	9	74,33%
22	1									9	6	96,67%	1	1	94,56%				2	3	91,10%	1	1	74,47%
23	37									20	8	97,78%	37	19	97,21%				1	1	91,24%	1	1	74,61%
24	7									15	9	99,03%	7	3	97,63%				9	7	92,23%	1	0.9	74,61%
25	1									4	2	99,31%	1	1	97,77%				1	2	92,51%	1	1	74,75%

Tabela 4.2.24: Relação de quantidade de nós e tempo de execução nas 25 primeiras sub-árvores, processos 8 a 15, instância i640-212

4.3 Testes com Tolerância a Falhas

Nesta seção serão apresentados os resultados alcançados com a introdução do procedimento de Tolerância a Falhas.

Para determinar o intervalo ideal entre gravações de *checkpoints*, foi verificado que a gravação de *checkpoints,* com até 1024 bytes leva menos que 0,001 segundos no ambiente de teste e foi estimado p_j como 0,01. Esta probabilidade de um processador falhar durante a aplicação é significativamente maior do que as estimativas encontradas em alguns trabalhos relacionados, tal como [23]. No entanto, foi considerado como adequado já que as máquinas na *Grid* são tipicamente compartilhadas com muitos usuários, geralmente estudantes e pesquisadores executando aplicações instáveis que podem derrubar o sistema. Aplicando a fórmula apresentada na seção 3.3 desta dissertação, é obtido um tempo recomendado entre *checkpoints* de cerca de 0,45 segundos. Como a solução de um nó nas instâncias testadas geralmente leva pelo menos este tempo, foi decidido gravar o *checkpoint* depois da solução de cada nó não folha, ou seja, a cada *branch* o processo envia a mensagem de *Checkpoint.*

O terceiro experimento, para avaliar o procedimento de tolerância a falhas, foi também executado na *Grid16*. A Tabela 4.3.1 apresenta as medidas de desempenho dos algoritmos em 3 tipos de execução:

(i) na ausência de falhas,

(ii) simulando a falha do processador de número 3 (primeiro cluster) depois deste ter executado cerca de 20% do tempo de relógio, baseado em testes anteriores, e

(iii) simulando a falha no cluster inteiro da PUC (2º cluster) também depois de 20% do tempo de relógio estimado.

Pela comparação dos resultados na Tabela 4.3.1, em relação aos resultados na ausência de falhas, é possível observar que não houve uma degradação significativa do desempenho causada pelas mensagens de *checkpoint* adicionais. Isto pode ser explicado pelo fato de que a maior parte das mensagens são intra-cluster. No caso de falha do processador, foram calculados os valores de *tnat* adicionando o tempo normalizado da CPU gasto pelo processador que tenha apresentado falhas, ao tempo de CPU normalizado dos processadores restantes.

Instância	**Mensagens**	***tnat***	***eff***
I640-211 sem falha	415003	553459	0,98
I640-211 falha de 1 processador	357301	491252	1,12
I640-211 falha de cluster	149704	310115	1,76
I640-212 sem falha	15872	19588	2,22
I640-212 falha de 1 processador	18102	19291	2,25
I640-212 falha de cluster	9810	17957	2,42
I640-213 sem falha	39040	39160	0,98
I640-213 falha de 1 processador	39437	42031	0,95
I640-213 falha de cluster	39883	44730	0,85
I640-214 sem falha	154696	154815	1,82
I640-214 falha de 1 processador	120926	158162	2,49
I640-214 falha de cluster	121619	250508	1,57
I640-215 sem falha	135195	135314	1,09
I640-215 falha de 1 processador	187699	164653	1,01
I640-215 falha de cluster	90758	134509	1,24

Tabela 4.3.1 Comparação dos resultados de execuções do *HierLB* com o procedimento de tolerância a falhas

É observado que o procedimento de tolerância a falhas pode realmente recuperar de uma falha com um mínimo impacto no *tnat*. Desde que nós façamos os *checkpoints* muito freqüentemente, quase nenhum trabalho é perdido por tal falha. Naturalmente, o tempo de relógio da aplicação é provavelmente aumentado, já que existem menos processadores disponíveis na *Grid* depois da falha.

No caso de falha de *cluster*, foi calculado o valor de *tnat* pela adição do tempo normalizado gasto por todos processos no *cluster* que tenha falhado aos tempos dos processadores restantes. Já que os *checkpoints* de *clusters* são menos freqüentes, uma substancial carga de trabalho pode ser perdida por uma falha de *cluster*. No entanto, a Tabela 4.3.1 mostra que alguns valores *tnat* são reduzidos depois que a falha de *cluster* da PUC foi simulada. Isto é facilmente explicado. Depois da falha, o processamento é reduzido para 8 máquinas da UFF, reduzindo-se, assim, o número total de troca de mensagens e evitando-se uma maior sobrecarga de mensagens *inter-cluster*. Naturalmente, como o poder computacional da *Grid* restante é muito menor, o tempo de relógio será provavelmente muito maior.

As Tabelas 4.3.2 até 4.3.6 apresentam detalhes das mensagens enviadas por cada um dos processadores (Veja no apêndice o resumo das descrições das mensagens utilizadas no algoritmo).

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	1	1	1	1	1	1	1	1	60	22	22	22	22	22	22	22	222
PrimalBound	84	14	0	0	0	14	0	7	42	14	0	7	0	0	7	56	245
Subtree	16	23	61	136	243	272	222	15	241	244	491	642	821	1023	800	174	5424
Idle	16	20	21	32	77	120	117	20	198	215	229	415	612	754	735	276	3857
LoadRequest	33	25	41	75	110	251	381	488	666	473	478	477	667	1061	1690	2269	9185
Awake	0	0	0	0	0	0	0	0	13	25	11	16	0	0	0	0	65
Alive	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
CheckStore	21054	22212	21411	22261	22056	22941	25731	23846	21769	21822	22266	23659	16017	17703	41315	43605	389668
ProcessCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckBranch	19	10	21	50	91	201	276	320	217	211	233	314	451	691	1049	1270	5424
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	5	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	18
SubtreCcluster	41	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	41
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	1	0	0	0	0	0	0	0	49	0	0	0	0	0	0	0	50
Notnow	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AliveCluster	354	0	0	0	0	0	0	0	354	0	0	0	0	0	0	0	708
CheckCluster	1	0	0	0	0	0	0	0	41	0	0	0	0	0	0	0	41
ClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckCluster	0	0	0	0	0	0	0	0	41	0	0	0	0	0	0	0	41
LeaderClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	21635	22305	21556	22555	22578	23800	26728	24697	23706	23026	23730	25552	18590	21254	45618	47673	415003

Tabela 4.3.2 : Relação de mensagens enviadas por processo– Técnica *HierLB* com tolerância a falhas – instância i640-211

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	32	24	24	24	24	24	24	24	1	1	1	1	1	1	1	1	208
PrimalBound	63	0	0	7	7	0	0	14	35	7	14	28	7	0	7	21	210
Subtree	78	68	77	74	31	17	5	10	12	31	56	13	26	25	34	0	557
Idle	94	98	121	136	118	127	132	137	7	12	26	32	25	25	21	17	1128
LoadRequest	287	214	177	151	224	94	72	26	31	11	23	65	43	55	58	69	1600
Awake	16	33	18	23	0	0	0	0	0	0	0	0	0	0	0	0	90
Alive	0	0	0	1	0	1	1	2	0	0	0	0	0	0	0	3	8
CheckStore	1725	886	796	1031	839	332	398	204	562	570	538	940	596	647	626	645	11335
ProcessCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckBranch	26	46	54	74	99	36	19	6	10	5	16	50	20	27	30	39	557
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	5	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	12
SubtreCcluster	1	0	0	0	0	0	0	0	27	0	0	0	0	0	0	0	28
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	46	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	47
Notnow	0	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	18
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AliveCluster	7	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	14
CheckCluster	27	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	28
ClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckCluster	27	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	28
LeaderClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	2436	1369	1267	1521	1342	631	651	423	722	637	674	1129	718	780	777	795	15872

Tabela 4.3.3 : : Relação de mensagens enviadas por processo– Técnica *HierLB* com tolerância a falhas – instância i640-212

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	125	47	47	47	47	47	47	47	1	1	1	1	1	1	1	1	462
PrimalBound	56	14	0	14	0	0	0	0	56	14	21	0	0	7	0	0	182
Subtree	120	93	87	81	67	81	93	120	3	11	10	24	30	29	36	45	930
Idle	193	221	256	265	284	300	314	338	7	7	11	12	18	27	28	31	2312
LoadRequest	952	407	250	293	218	188	208	228	115	7	12	17	27	35	51	58	3066
Awake	44	52	33	31	0	0	0	0	0	0	0	0	0	0	0	0	160
Alive	0	1	2	1	2	4	3	4	0	0	0	0	0	0	0	0	17
CheckStore	3682	2271	2201	2963	2676	1250	969	603	1480	1576	1396	3093	1501	1570	1662	1521	30414
ProcessCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckBranch	195	81	63	94	73	71	82	83	69	1	6	9	18	22	29	34	930
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	6	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	13
SubtreCcluster	1	0	0	0	0	0	0	0	85	0	0	0	0	0	0	0	86
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	174	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	175
Notnow	0	0	0	0	0	0	0	0	88	0	0	0	0	0	0	0	88
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AliveCluster	14	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	29
CheckCluster	85	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	86
ClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckCluster	85	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	86
LeaderClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	5734	3187	2939	3789	3367	1941	1716	1423	1931	1617	1457	3156	1595	1691	1807	1690	39040

Tabela 4.3.4 : Relação de mensagens enviadas por processo– Técnica *HierLB* com tolerância a falhas – instância i640-213

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	1	1	1	1	1	1	1	1	38	16	16	16	16	16	16	16	158
PrimalBound	77	0	14	14	0	7	0	7	35	0	0	35	7	14	0	0	210
Subtree	58	102	51	109	105	157	121	98	78	68	87	32	66	68	55	5	1260
Idle	96	64	17	26	37	57	71	116	80	104	114	104	96	98	121	115	1316
LoadRequest	196	189	288	56	117	118	188	126	229	168	154	190	118	126	124	125	2512
Awake	0	0	0	0	0	0	0	0	15	19	8	13	0	0	0	0	55
Alive	0	0	0	0	0	0	0	0	0	0	1	2	2	2	2	3	12
CheckStore	18751	9369	21113	15237	13618	5646	5249	3088	5848	5718	6222	12423	5842	6400	6412	6575	147511
ProcessCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckBranch	118	88	148	42	98	86	144	77	16	46	62	98	44	73	51	69	1260
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	6	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0	15
SubtreCcluster	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	27
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	1	0	0	0	0	0	0	0	96	0	0	0	0	0	0	0	97
Notnow	69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	69
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AliveCluster	68	0	0	0	0	0	0	0	67	0	0	0	0	0	0	0	135
CheckCluster	1	0	0	0	0	0	0	0	27	0	0	0	0	0	0	0	28
ClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckCluster	0	0	0	0	0	0	0	0	27	0	0	0	0	0	0	0	27
LeaderClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	19470	9813	21632	15485	13976	6072	5774	3513	6567	6139	6664	12913	6191	6797	6781	6908	154696

Tabela 4.3.5 : Relação de mensagens enviadas por processo - Técnica *HierLB* com tolerância a falhas – instância i640-214

Mensagens	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	TOTAL
Suspectend	1	1	1	1	1	1	1	1	134	75	75	75	75	75	75	75	667
PrimalBound	63	7	21	14	0	7	0	0	56	7	0	14	0	0	7	0	196
Subtree	87	128	23	6	13	49	98	156	261	162	117	111	111	100	80	17	1519
Idle	97	70	22	9	9	13	39	98	348	396	437	443	458	496	480	459	3874
LoadRequest	222	195	288	80	16	19	31	59	845	624	544	475	471	397	441	372	5079
Awake	0	0	0	0	0	0	0	0	41	78	44	55	0	0	0	0	218
Alive	0	0	0	0	0	0	0	0	0	0	0	1	0	1	2	4	8
CheckStore	17785	8260	18530	13077	13456	5481	4489	2542	3699	4463	4276	8017	4346	4856	3968	4143	121388
ProcessCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckBranch	157	112	175	36	6	10	24	40	42	125	125	140	119	125	148	135	1519
SuspectendCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
PrimalBoundCluster	7	0	0	0	0	0	0	0	6	0	0	0	0	0	0	0	13
SubtreCcluster	106	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	106
IdleCluster	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2
LoadRequestCluster	1	0	0	0	0	0	0	0	201	0	0	0	0	0	0	0	202
Notnow	95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	95
AwakeCluster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AliveCluster	47	0	0	0	0	0	0	0	47	0	0	0	0	0	0	0	94
CheckCluster	1	0	0	0	0	0	0	0	106	0	0	0	0	0	0	0	107
ClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AckCluster	0	0	0	0	0	0	0	0	106	0	0	0	0	0	0	0	106
LeaderClusterCrash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	18671	8773	19060	13223	13501	5580	4682	2896	5894	5930	5618	9331	5580	6050	5201	5205	135195

Tabela 4.3.6 : Relação de mensagens enviadas por processo– Técnica *HierLB* com tolerância a falhas – instância i640-215

4.4 Testes de Escalabilidade

Os testes finais visam verificar a escalabilidade do algoritmo proposto. Foi executada a mesma instância em mais de um ambiente com potencial computacional diferente. A Grid16 corresponde ao ambiente de teste utilizado até então que compreende 8 máquinas (com diferentes velocidades) da UFF e 8 máquinas Pentium IV 1.7 GHz na PUC, com um poder computacional equivalente a 23 máquinas padrão. A Grid24 é composta das 8 máquinas na UFF e as mesmas 16 máquinas na PUC que alcançam um poder computacional de 31 máquinas padrão. E por fim a Grid32 que utiliza 8 máquinas na UFF e 24 máquinas na PUC com cerca de 39 máquinas padrão.

	SEQ	**Grid 16**	**Grid 24**	**Grid 32**	**eff16**	**eff24**	**eff32**
i640-211	545387	576501	361750	368984	0,95	1,50	1,48
i640-212	43479	19588	13152	14169	2,22	3,31	3,07
i640-213	38187	39107	24049	25472	0,98	1,59	1,50
i640-214	393260	216084	152402	154144	1,82	2,58	2,55
i640-215	166188	152079	132713	133298	1,09	1,25	1,25
Média das eficiências					1,41	2,05	1,97

Tabela 4.4.1.Comparação entre resultados de execuções do algoritmo em ambientes de diferente poder computacional.

Os resultados apresentados na Tabela 4.4.1 mostram que não houve perda de desempenho por utilizar mais máquinas, indicando que o algoritmo proposto é escalável.

Na verdade, com o aumento de processadores, mais sub-árvores são distribuídas e maiores são as chances de se encontrar a melhor solução mais rapidamente. Isto pode ser observado quando comparamos os resultados obtidos pelas Grid16 e Grid24. Entretanto, no caso da Grid32 que possui 8 processadores da UFF e 24 da PUC, ocorreu um maior desbalanceamento entre os dois *clusters*, o que resultou na ligeira redução da eficiência, quando comparada à Grid24.

Capítulo 5

Conclusões

Os resultados experimentais obtidos nesta dissertação deixam clara a importância do balanceamento de carga em algoritmos de *branch-and-bound*, bem como a necessidade de um procedimento que permita, em ambientes de *Grids,* que a aplicação termine corretamente, mesmo na presença de falhas, em tempos de execução satisfatórios. Note que falhas são mais comuns em se tratando de equipamentos que terão de funcionar durante grandes períodos de tempo.

O balanceamento proposto neste trabalho é bem adaptável a ambientes de *Grids*, devido à sua estrutura hierárquica, e não mestre-escravo, permitindo que a aplicação seja escalável.

A tolerância a falhas é importante principalmente em problemas que necessitem de um grande poder computacional. O algoritmo apresentado resolve situações comuns de falhas quando utilizamos *Grids*, que geralmente compõem-se de *clusters* dispersos geograficamente. Particularmente, foram apresentadas soluções para dois casos de falhas: em um processador e em um *link* que interconecta diferentes *clusters*. A necessidade de re-execução, em caso de falha de processador, pode ser considerada nula ou, no pior dos casos, ocorre a necessidade de re-execução de um único nó da árvore de *branch-and-bound.* Mesmo com o aumento de mensagens trocadas *intra-clusters*, os

resultados mostram que a diferença deste acréscimo de mensagens não é significativa no desempenho do algoritmo. Em caso de falha de *cluster*, haverá a necessidade de re-execução de parte da sub-árvore já executada no mesmo. Mas esta re-execução poderá ser feita em menor tempo, pois as melhores soluções encontradas nesta sub-árvore já terão sido difundidas através da mensagem de *PrimalBound*, o que fará com que a sub-árvore, na re-execução, seja podada mais rapidamente.

Técnicas desenvolvidas para este *branch-and-bound* distribuído aplicado ao SPG podem ser utilizáveis na solução de diversos outros problemas de otimização combinatória.

As contribuições futuras poderão ter um foco direcionado à melhoria do procedimento de distribuição inicial e procedimentos de balanceamento de carga para obtenção de melhores combinações entre subproblemas e *clusters*. *Clusters* com maior poder computacional receberiam maiores sub-árvores, reduzindo o custo de redistribuição de carga entre eles. Para isto, é necessário que sejam desenvolvidos métodos razoáveis para estimativa de tamanho de árvores.

Finalmente existem, atualmente, cinco instâncias ainda não resolvidas da SteinLib [20]. Esperamos poder aplicar o algoritmo desenvolvido para resolvê-las, assim que tivermos acesso a uma *Grid* com maior poder computacional.

Capítulo 6

Referências

[1] K. Aida, W. Natsume and Y. Futakata, *Distributed computing with hierarchical master-worker paradigm for parallel Branch-and-Bound algorithm*, Proc. CCGrid'03, International Symposium on Cluster Computing and Grid (Tokyo, Japan, 2003) 156-162.

[2] K. Anstreicher, N. Brixius, J.-P. Goux and J. T. Linderoth, *Solving Large Quadratic Assignment Problems on computational Grids*, Mathematical Programming, Series B, 91 (2002) 563-588.

[3] R. Baoukov and T. Soverik, *A Generic Parallel Branch-and-Bound Environment on a Network of Workstations*, Proceedings of Hiper-99, (1999) 474-483.

[4] A. Bruin, G.A.P. Kindervater and H.W.J.M. Trienekens, *Asynchronous parallel Branch-and-Bound and anomalies,* Report EUR-CS-95-05, Department of Computer Science, Erasmus University, Roterdam, 1995.

[5] J. Clausen, *Parallel Branch-and-Bound*, *Principles and Personal Experiments*, Kluwer Academic Publishers (1997) 239-267.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken, LogP: *towards a realistic model of parallel computation*, Proc. PPOPP '93, Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Diego, United States, 1993) 1-12.

[7] Y. Denneulin, B. Le Cun, T. Mautor and J.F. Méhaut, *Distributed Branch-and-Bound Algorithms for Large Quadratic Assignment Problems*, Proceedings of the Fifth Computer Science Technical Section on Computer Science and Operation Research, Dallas-USA, (January 96) 8-10.

[8] D. Z. Du, J.M. Smith and J.H. Rubistein (Eds), *Advances in Steiner Trees*, Combinatorial Optimization Series Vol. 6, Springer, 2000.

[9] C. Duin, Steiner's Problem in graphs, Ph.D. thesis, University of Amsterdam, 1993.

[10] C. W. Duin and A. Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, (1987) 17(2), 353–364.

[11] D. Eager and E. Lazowska, "*A comparison of receiver-initiated and sender-initiated adaptive load sharing*," Performance Evaluation, (1986) vol. 6, 53-68.

[12] M. Elnozahy, L. Alvisi, Y. Wang and D.B. Johnson, *A survey of rollback-recovery protocols in message-passing systems*, Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1996.

[13] I. Foster and C. Kesselman, *The Globus project: a status report*, Future Generation Computer Systems 15 (1999) 607-621.

[14] I. Foster and C. Kesselman, *The Grid: blueprint for a new computing infrastructure* (Morgan-Kaufmann, San Francisco, 1999).

[15] B. Gendron ant T.G. Cranic, *Parallel Branch-and-Bound Algorithms: Survey and Synthesis*, Operation Research 42 (1994) 1042-1066.

[16] J.-P Goux, S. Kulkarni, J.T. Linderoth and M.E. Yoder, *Master-Worker: an enabling framework for applications on the computational Grid*, Cluster Computing 4 (2001) 63-70.

[17] W. Gropp and E. Lusk, *Fault tolerance in message passing interface programs*, The International Journal of High Computing Applications 18 (2004) 363-372.

[18] A. Iamnitchi and I. Foster, *A problem-specific fault-tolerance mechanism for asynchronous, distributed systems*, em Proc. ICPP'00, International Conference on Parallel Processing (Toronto, Canada, 2000) 4-14.

[19] N. Karonis, B. Toonen and I. Foster, MPICH-G2: *A Grid-enabled implementation of the message passing interface*, Journal of Parallel and Distributed Computing 63 (2003) 551-563.

[20] T. Koch, A. Martin and S. Voss, SteinLib: *an updated library on Steiner Problems in graphs,* Konrad-Zuse-Zentrum für Informationstechnik Berlin, ZIB-Report 00-37 (2000) http://elib.zib.de/steinlib.

[21] T.-H. Lai, S. Sahni, *Anomalies in parallel branch-and-bound algorithms*, Communications of the ACM 27 (1984) 594-602.

[22] M. Poggi de Aragão, E. Uchoa and R.F. Werneck, *Dual heuristics on the exact solution of Large Steiner Problems*, Electronic Notes in Discrete Mathematics 7 (2001) 46-51.

[23] X. Qin, H. Jiang, D.R.Swanson, *An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints in Heterogeneous Systems*, in the Proceedings of the 31st International Conference on Parallel Processing *(ICPP 2002),* (Vancouver, Canada, 2002) 360-368.

[24] Ribeiro, Uchoa e Werneck é "*A hybrid GRASP with perturbations for the Steiner problem in graphs"*, INFORMS Journal on Computing, (2002) Vol. 14, 228-246.

[25] A. Robertson, *Linux-HA heartbeat system design*, in: Proc. ALS'00, Annual Linux Showcase and Conference (Atlanta, Georgia, 2000) 305-316.

[26] E. Uchoa, *Algoritmos para problemas de Steiner com aplicações em projeto de circuitos VLSI*, Ph.D. thesis, Catholic University of Rio de Janeiro, 2001.

[27] E. Uchoa, "*Preprocessing Steiner problems from VLSI layout*", Networks, (2002) Vol. 40, 38-50.

[28] J. Weissman, *Fault tolerant computing on the Grid: what are my options*?, em: Proc. HPDC'99, IEEE International Symposium on High Performance Distributed Computing (California, USA, 1999) 26.

[29] J. Weissman, *Fault tolerant Wide-Area parallel computing, in: Proc. IPDPS'00,* Workshop on Fault-Tolerant Parallel and Distributed Systems, International Parallel and Distributed Processing Symposium (Cancun, Mexico, 2000).

[30] R. Werneck, *Problema de Steiner em Grafos: Algoritmos Primais, Duais e Exatos*. Master's Thesis, Department of Informatics, PUC-Rio, 2001.

[31] R. Wong, *A dual ascent approach for Steiner Tree Problems on a directed graph*, Mathematical Programming 28 (1984) 271-287.

APÊNDICE

.

1) Mensagens de distribuição inicial e balanceamento de carga:

Subtree …….. Envio de uma sub-árvore (carga) para o vizinho de *cluster.*
SubTreeCluster……. Envio de uma sub-árvore (carga) para outro *cluster.*
LoadRequest ……... Pedido de sub-árvore (carga) a outro processo.
LoadRequestCluster Pedido de sub-árvore (carga) a outro *cluster.*
Idle…….. Informa que o processo que enviou esta mensagem está ocioso (sem carga)
IdleCluster…….. Informa que o *cluster* que enviou esta mensagem está ocioso (sem carga)
Notnow…….. Informa que o líder do *cluster* está ocioso (sem carga), mas outro(s) processo(s) do *cluster* ainda pode(m) possuir carga.
Awake…….. Enviado aos processos para que iniciem suas atividades pedindo carga a outros processos, pois não houve mais carga a ser distribuída inicialmente.
AwakeCluster……. Enviado aos *clusters* para que iniciem suas atividades pedindo carga a outros *clusters,* pois não houve mais carga a ser distribuída inicialmente.

2) Mensagens de terminação

Suspectend…….. Informa ao líder do *cluster* que o processo que está enviando esta mensagem terminou sua tarefa e não encontrou carga com seus vizinhos de *cluster.*
SuspectendCluster…. Informa ao líder da aplicação que o *cluster* que está enviando esta mensagem terminou sua tarefa e não encontrou carga em outros *clusters.*
Teminate…. Informa aos processos que podem encerrar a execução.

3) Mensagens de difusão do limite *primal*

PrimalBound……... Difusão do limite *primal* encontrado para os vizinhos de *cluster.*
PrimalBoundCluste... Difusão do limite *primal* para os outros *clusters.*

4) Mensagens de tolerância a falhas

Alive..... Indica que o processo que enviou esta mensagem está ativo.
CheckStore..... Contém o *checkpoint* do processo que enviou esta mensagem
ProcessCrash..... Informar aos processos vizinhos que um determinado processo falhou.
AliveCluster..... Indica que o *cluster* que enviou esta mensagem está ativo.
CheckCluster..... Contém o *checkpoint* do cluster que enviou esta mensagem.
ClusterCrash..... Informar aos outros *clusters* que um determinado *cluster* falhou
LeaderClusterCrash. Informar aos líderes dos outros *clusters* da falha e mudança de líder do *cluster* de onde esta mensagem foi enviada.
AckBranch…… Confirma o recebimento da sub-árvore originada do processo que está recebendo esta mensagem
AckCluster…… Confirma o recebimento da sub-árvore originada do *cluster* que está recebendo esta mensagem