Construção Paralela de Árvores de Cortes Utilizando Contrações de Grafo Otimizadas

Charles Maske[1], Jaime Cohen[2], Elias P. Duarte Jr[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19018 – CEP 81531-980 – Curitiba, PR

[2]Departamento de Informática - Universidade Estadual de Ponta Grossa (UEPG)
Av. Gen. Carlos Cavalcanti, 4748 – CEP 84030-900 – Ponta Grossa, PR

Abstract. *A cut tree is a combinatorial structure that represents the edge-connectivity between all pairs of nodes of an undirected graph. Cut trees are used in applications to solve graph connectivity problems, graph partitioning and clustering, routing and in the analysis of complex networks, including social networks, biological networks, among others. This paper presents a parallel version of the classical Gomory-Hu cut tree construction algorithm. The Gomory-Hu algorithm makes multiple calls to a minimum cut algorithm. Minimum cuts are computed in contracted graphs obtained from the input graph. This paper presents an efficient strategy to compute the contracted graphs. The proposed strategy allows processes to take advantage of previously contracted graphs instances. The proposed algorithm was implemented using MPI and experimental results are presented for several families of graphs demonstrating the performance gains of the proposed strategy.*

Resumo. *As árvores de cortes representam, de forma compacta, a aresta conectividade de um grafo. Suas aplicações são diversas, incluindo roteamento, avaliação de conectividade, particionamento e agrupamento em grafos, além da análise de redes complexas, incluindo redes sociais, redes formadas a partir de dados biológicos, entre outras. Neste trabalho é apresentada uma versão paralela de um dos algoritmos clássicos para a construção de árvores de cortes, o algoritmo de Gomory-Hu. Este algoritmo faz múltiplas chamadas a um procedimento que encontra um corte de arestas de capacidade mínima entre dois vértices. Para encontrar os cortes mínimos, o algoritmo faz contrações de vértices do grafo de entrada. A principal contribuição do algoritmo apresentado neste trabalho é a especificação de uma estratégia eficiente que permite que processos aproveitem instâncias de grafos contraídos em passos anteriores. O algoritmo proposto foi implementado em MPI e resultados experimentais são apresentados para diversas famílias de grafos, demonstrando os ganhos de desempenho da estratégia proposta.*

1. Introdução

As árvores de cortes são estruturas combinatórias que representam, de forma compacta, a aresta conectividade entre todos os pares de vértices de um grafo [Nagamochi and Ibaraki 2008]. A aresta conectividade entre um par de vértices s e t é a capacidade de um s-t corte mínimo. As árvores de cortes são utilizadas na solução de

importantes problemas combinatórios incluindo: roteamento [Duarte Jr et al. 2004], particionamento e agrupamento em grafos [Engelberg et al. 2006], entre outros. As árvores de cortes também possuem aplicações na análise de redes complexas, análise de dados biológicos [Tuncbag et al. 2010], análise de redes sociais [Kamath and Caverlee 2011] entre outras.

Existem dois algoritmos sequenciais clássicos para encontrar a árvore de cortes de um grafo com capacidades nas arestas: o algoritmo de Gomory e Hu [Gomory and Hu 1961] e o algoritmo de Gusfield [Gusfield 1990]. Ambos os algoritmos são similares já que fazem chamadas a um procedimento que calcula subproblemas de cortes mínimos entre pares de vértices $n-1$ vezes para um grafo com n vértices. A principal diferença entre os dois é o fato de que o algoritmo de Gusfield descobre os $n-1$ cortes mínimos sobre o grafo de entrada e o algoritmo de Gomory-Hu realiza contrações e calcula os cortes mínimos sobre instâncias de grafos contraídos. Para realizar contrações de vértices, o algoritmo de Gomory-Hu exige a manipulação de estruturas de dados não triviais.

Versões paralelas para os algoritmos de Gusfield e Gomory-Hu foram apresentadas, respectivamente, em [Cohen et al. 2011] e [Cohen et al. 2012]. De maneira geral ambas as versões paralelas dos algoritmos para construção de árvores de cortes utilizam a arquitetura mestre-escravo para paralelizar os cálculos dos *s-t* cortes mínimos. O processo mestre distribui tarefas com pares de vértices s e t para os processos escravos. Estes, por sua vez, resolvem subproblemas de *s-t* cortes mínimos e enviam a resposta ao processo mestre. O processo mestre, ao receber as respostas dos processos escravos, atualiza a árvore e envia novas tarefas. O algoritmo termina com a árvore de cortes construída quando não existirem mais vértices a serem separados pelo procedimento de *s-t* corte mínimo. Experimentos anteriores com versões paralelas do algoritmo de Gomory-Hu mostraram que as construções do grafos contraídos demandam muito tempo, chegando a consumir, em muitos casos, a metade do tempo de execução [Cohen 2013]. Implementar de maneira eficiente este procedimento no algoritmo paralelo de Gomory-Hu não é trivial, uma vez que os escravos não têm acesso a estruturas de dados compartilhadas.

Este trabalho propõe uma estratégia eficiente para realizar as contrações na versão paralela do algoritmo de Gomory-Hu. A ideia principal é fazer com que os processos escravos aproveitem instâncias de grafos contraídos de passos anteriores sem a necessidade de refazer todas as contrações no grafo de entrada na execução de cada tarefa. De forma geral, a implementação dessa otimização requer que o processo mestre tenha controle sobre as tarefas que foram enviadas para cada escravo, armazenando-as para que sejam utilizadas quando as respostas chegarem. Os processos escravos, por sua vez, precisam tomar decisões sobre quando aproveitar, ou não, uma instância de grafo contraído.

Para avaliar a eficiência desta versão foram realizados experimentos em um *cluster* de alto desempenho. Foram realizados testes de *speedup* e comparativos entre as versões paralelas preexistentes. Os experimentos foram feitos em vários conjuntos de instâncias que incluíram grafos sintéticos e grafos reais. Os resultados demonstram *speedups* próximos de lineares para a maioria das instâncias. Nas comparações é possível comprovar os ganhos obtidos com a estratégia proposta.

Este trabalho está organizado da seguinte forma. Na seção 2 são descritas as

versões sequencial e paralela do algoritmo de Gomory-Hu. Na seção 3 é apresentado o algoritmo proposto. Na seção 4 são apresentados os resultados experimentais e na seção 5 seguem as conclusões finais.

2. O Algoritmo de Gomory-Hu

Esta seção inicia com uma descrição do algoritmo de Gomory-Hu sequencial, seguida de uma descrição da versão paralela deste algoritmo.

2.1. O Algoritmo de Gomory-Hu Sequencial

Considere o problema de descobrir a aresta conectividade entre todos os pares de vértices de um grafo capacitado. Seja $G = (V, E, c)$ um grafo tal que V é o conjunto de vértices, E é o conjunto de arestas e c é uma função $c : E \rightarrow \mathbb{R}^+$ chamada de capacidade das arestas. A aresta conectividade entre um par de vértices s e t pode ser calculada aplicando um algoritmo de *fluxo máximo* que encontra um *s-t* corte mínimo. Um *s-t* corte é uma partição $\{S, \overline{S}\}$ do conjunto de vértices V que obrigatoriamente separa os vértices s e t, ou seja, $s \in S$ e $t \in \overline{S}$. A solução trivial é calcular os *s-t* cortes mínimos entre todos os pares de vértices. Gomory e Hu [Gomory and Hu 1961] mostraram que existem exatamente $|V| - 1$ cortes que incluem ao menos um corte mínimo entre cada par de vértices do grafo. Esses $|V| - 1$ cortes são representados sucintamente através de uma árvore de cortes. Em seguida uma descrição geral do algoritmo de Gomory-Hu é apresentada.

A Figura 1 mostra um grafo exemplo e a árvore de cortes correspondente. A árvore de cortes descreve a aresta conectividade entre todos os pares de vértices do grafo. Assim, por exemplo, entre os vértices 3 e 7 do grafo G, temos um *s-t* corte mínimo $\{S, \overline{S}\} = \{\{0, 1, 2, 3, 4\}, \{5, 6, 7\}\}$ de capacidade $c(S, \overline{S}) = 5$. Na árvore T a aresta $\{3, 5\}$ é a de menor capacidade entre os vértices 3 e 7 e a remoção dela induz um *s-t* corte mínimo em G.

Para auxiliar a descrição do Algoritmo 1, o termo *nodo* refere-se aos vértices da árvore de cortes. O algoritmo, incluindo processo de contração apresentado na linha 4, será explicado em seguida.

Algoritmo 1: Algoritmo Sequencial de Gomory-Hu

Entrada: $G = (V, E_G, c_G)$ um grafo não dirigido e com peso nas arestas
Saída: $T = (V, E_T, c_T)$ uma árvore de cortes de G
1 $T \leftarrow (V_T = \{V\}, E_T = \emptyset)$;
2 **enquanto** $\exists X \in V_T$ *tal que* $|X| > 1$ **faça**
3 escolha um nodo $X \in T$;
4 contraia G para formar G';
5 escolha dois vértices $s, t \in X$;
6 calcule o *s-t* corte mínimo em G';
7 atualize T ;
8 **retorne** T

Seja $G = (V_G, E_G, c_G)$ o grafo de entrada com peso nas arestas. O algoritmo inicia a árvore $T = (V_T, E_T, c_T)$ com um nodo representando todos os vértices do grafo de entrada tal que $V_T = \{V_G\}$ e $E_T = \emptyset$. Seja $G' = (V'_G, E'_G, c'_G)$ o grafo contraído iniciado com $G' = G$. Em cada iteração, o algoritmo escolhe um nodo $X \in V_T$ da árvore T, tal que $|X| > 1$, designado de *pivô*. O algoritmo busca todos os vértices das componentes conexas na árvore $T \setminus X$. Os vértices dessas componentes conexas serão

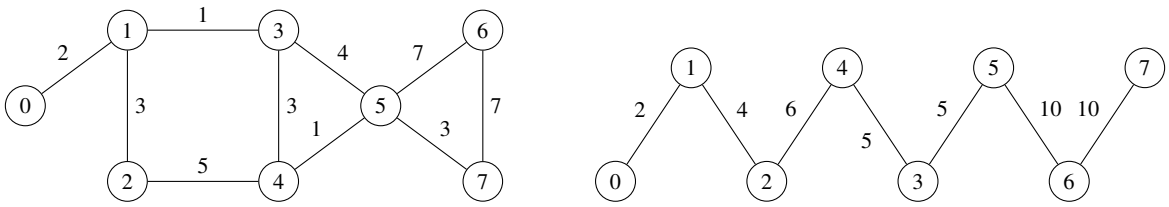

Figura 1. Um exemplo de grafo com peso nas arestas e sua árvore de cortes correspondente.

denotados por $X_1, X_2, \ldots, X_k$, respectivamente. O grafo contraído G' é construído de forma que cada conjunto de vértices X_i é contraído em um único vértice. Tal grafo será denotado por $G' = G \setminus X_1, X_2, \ldots, X_k$. Dois vértices $s, t \in X$ são escolhidos. Calcula-se o corte mínimo entre s, t sobre o grafo contraído G' obtendo a partição $\{S, \overline{S}\}$ e o valor do corte $c(S, \overline{S})$. São criados dois novos nodos X_s, X_t na árvore T tal que $X_s = \{X \cap S\}$ e $X_t = \{X \cap \overline{S}\}$. Uma nova aresta $e = \{X_s, X_t\}$ com $c(e) = c(\{S, \overline{S}\})$ é adicionada a T. Para cada aresta $e' = \{X, Y\} \in E_T$ incidente em X e Y na árvore T, verificamos em qual lado do *s-t* corte mínimo $\{S, \overline{S}\}$ os vértices pertencentes ao nodo Y estão. Caso estejam no lado S então liga-se o nodo X_s a Y formando a aresta $\{X_s, Y\}$. Caso contrário liga-se o nodo X_t a Y formando a aresta $\{X_t, Y\}$. O algoritmo atualiza a árvore de tal forma que o conjunto de vértices seja $V_T = (V_T \setminus \{X\}) \cup \{X_s, X_t\}$ e o conjunto de arestas seja $E_T = E_T \cup \{e\}$.

O algoritmo iterativamente computa cortes para $|V| - 1$ pares de vértices. Estes pares são escolhidos de um conjunto de nodos na árvore T que contêm vértices ainda não separados pelos cortes até então calculados. Os cortes são computados sobre instâncias de grafos contraídos que são são construídos a partir da estrutura parcial da árvore e do grafo de entrada.

2.2. O Algoritmo de Gomory-Hu Paralelo

Em [Cohen et al. 2012] os autores apresentam uma versão paralela do algoritmo de Gomory-Hu. Os autores fizeram a sua implementação utilizando a biblioteca MPI seguindo a arquitetura mestre-escravo. O processo mestre é responsável por realizar as atualizações na árvore em construção e gerar instâncias do problema do *s-t* corte mínimo para os processos escravos. Esses constroem instâncias de grafo contraído e calculam o *s-t* corte mínimo retornando ao mestre o resultado. A estratégia, segundo os autores, é a do menor esforço, uma vez que os processos escravos calculam os *s-t* cortes mínimos sem sincronismo, apesar da possibilidade de alguns cortes invalidarem outros.

Seja p o número total de processos e $proc_0,\ proc_1,\ \ldots,\ proc_{p-1}$ os processos que executam o algoritmo paralelo de Gomory-Hu. Cada processo, ao iniciar, mantém uma cópia do grafo de entrada $G = (V_G, E_G, c_G)$. O processo mestre, identificado como $proc_0$, inicializa a árvore $T = (V_T, E_T, c_T)$, escolhe um nodo pivô $X \in V_T$ tal que $|X| \geq 2$. Em seguida é criada uma *tarefa*, composta por um par de vértices $s, t \in X$ e uma *partição* que associa os vértices do grafo G aos vértices do grafo contraído. Os vértices s, t são escolhidos e a *partição* é criada da seguinte forma: cada vértice do nodo pivô recebe um número sequencial único; esses são os vértices que não sofrerão contração. Os demais

vértices serão numerados de acordo com a sua componente conexa. O processo mestre envia para cada processo escravo $proc_s$, $s > 0$, uma *tarefa* e fica aguardando as respostas.

Um processo escravo $proc_s$, por sua vez, recebe a tarefa do processo mestre e constrói um grafo contraído G'. A construção do grafo contraído consiste em percorrer a lista de arestas do grafo de entrada e determinar, para cada uma delas, se existe uma aresta correspondente no grafo contraído G'. Em seguida, com o par de vértices *s* e *t*, o processo $proc_s$ resolve um problema de *s-t* corte mínimo e devolve para o processo mestre uma *resposta* composta pelos vértices *s* e *t* recebidos na tarefa e pelo *s-t* corte mínimo. O processo mestre recebe de $proc_s$ a *resposta* e verifica se o par de vértices *s* e *t* estão contidos no mesmo nodo da árvore. Caso estejam, a árvore T será atualizada. No passo seguinte, o processo mestre escolhe um novo nodo *pivô* e envia uma nova *tarefa* para $proc_s$ e processa outras *respostas* ou fica aguardando outro processo escravo se comunicar.

A atualização da árvore deve ser feita de forma serial pois não são permitidas atualizações concorrentes. Essa característica faz com que o mestre processe uma resposta de cada vez. Algumas respostas poderão ser descartadas, já que os vértices *s* e *t* podem já ter sido separados por outra atualização de T resultante de outro *s-t* corte mínimo encontrado por outro processo.

A descrição do Algoritmo 2 mostra a versão paralela do algoritmo de Gomory-Hu em alto nível, em que as operações *envie* e *receba* correspondem às funções de troca de mensagens da biblioteca MPI. Na seção seguinte será apresentado o algoritmo com as otimizações e com um nível maior de detalhamento.

Algoritmo 2: Algoritmo Paralelo De Gomory-Hu

```
   Entrada: G = (V, E_G, c_G) , proc_j, 0 ≤ j < p processos
   Saída: T = (V, E_T, c_T) uma árvore de cortes de G
1  se proc_j = 0 então
       // processo mestre
2      T ← {{V_G}, ∅} ;
3      distribua tarefas para todos os processos;
4      enquanto |V_T| < |V_G| faça
5          receba de proc_j resposta (s,t,S), onde {S, S̄} é um s-t corte mínimo de G ;
6          se os vértices s, t não estão separados então
7              atualize a árvore ;
8          se existirem nodos com mais de dois vértices em T então
9              envie nova tarefa para proc_j ;
10     retorne T ;
11 senão
       // processo escravo
12     enquanto receber tarefas faça
13         receba tarefa (s,t,partição);
14         se tarefa = fim então
15             termine;
16         construa grafo contraído ;
17         calcule s-t corte mínimo entre s e t ;
18         envie reposta (s,t,S) para proc_0;
```

3. Algoritmo de Gomory-Hu Paralelo Utilizando Contrações Otimizadas

Através dos experimentos realizados com a versão paralela do algoritmo de Gomory-Hu (apresentado no seção 2.2), foi constatado que a construção do grafo contraído é um dos pontos que consomem uma grande quantidade de tempo [Cohen 2013]. Portanto melhorar a eficiência deste procedimento é um caminho para trazer ganhos no tempo de execução.

Implementar de forma eficiente as operações de contração no algoritmo de Gomory-Hu não é trivial.

Na versão paralela do algoritmo de Gomory-Hu, as contrações são realizadas sobre o grafo de entrada em todas tarefas executadas pelos processos escravos. O objetivo principal deste trabalho é especificar uma forma de realizar as contrações de forma eficiente, possibilitando aos processos escravos aproveitar instâncias de grafos contraídos para que novas contrações sejam realizadas sobre esses, e não sobre o grafo de entrada, sempre que possível. Como já vimos anteriormente na seção 2.2, a construção do grafo contraído depende da partição de vértices construída a partir do pivô escolhido. Para melhor compreensão da otimização considere as seguintes definições:

Definição 1 *Dado um grafo $G = (V, E)$, dizemos que $G' = (V', E')$ é um grafo contraído a partir de G se:*

- *O conjunto de vértices V' de G' é uma partição de V*
- *Para cada $\{u_1, u_2\} \in E$ existe uma aresta $\{U_1, U_2\} \in E'$ se $u_1 \in U_1$ e $u_2 \in U_2$*

Definição 2 *Dadas partições P e Q de um conjunto X, dizemos que P é um refinamento de Q se todo elemento de P é subconjunto de algum elemento de Q.*

Por exemplo, a partição $\{\{1,3\},\{2,4,7\},\{5\},\{6,8\}\}$ de $\{1,2,3,4,5,6,7,8\}$ é um refinamento da partição $\{\{1,3,2,4,7\},\{5,6,8\}\}$.

Definição 3 *Dizemos que o grafo contraído $G'' = (V'', E'')$ é um refinamento de $G' = (V', E')$ se V'' é um refinamento de V'.*

No contexto do algoritmo de Gomory-Hu paralelo, as operações de contração de grafos podem ser otimizadas desde que o grafo contraído a ser construído seja um refinamento do grafo contraído utilizado no passo anterior. Além disso, é necessário um teste eficiente para determinar quais tarefas definem grafos contraídos que são refinamentos de outros. O lema abaixo indica como esse teste pode ser implementado. Utilizaremos a seguinte notação: $G'(X, T, G)$ é grafo contraído induzido pela escolha do pivô X na árvore de cortes parcial T de G.

Lema 1 *O grafo contraído $G'(X, T_1, G)$ é um refinamento do grafo contraído $G'(Y, T_2, G)$ se, e somente se, o nodo pivô X está contido em Y.*

O algoritmo então faz a escolha de um nodo pivô que esteja contido no pivô utilizado para produzir a tarefa anterior submetida a um processo escravo. Essa estratégia é suficiente para garantir que o novo grafo contraído seja um refinamento daquele utilizado na iteração anterior pelo mesmo processo. Assim, o grafo contraído é construído a partir do grafo da iteração anterior ao invés de ser construído a partir do grafo de entrada.

Para que o processo mestre possa fazer a escolha do pivô com base no lema 1, ele necessita armazenar os últimos pivôs escolhidos nas tarefas enviadas aos processos escravos. O nodo escolhido para ser o novo pivô será aquele com o maior número de vértices dentre aqueles contidos no pivô da iteração anterior. Essa escolha aumenta a chance de futuros refinamentos serem encontrados.

Outra estratégia de otimização implementada consiste em não executar contrações que não produzam um grafo muito menor. Dessa forma, se a operação de contração não

reduzir o número de vértices de uma constante k, a contração não é efetuada e o grafo da iteração anterior é utilizado no cômputo do corte mínimo. Nos experimentos relatados adiante, usa-se uma constante igual a 10.

O Algoritmo 3 abaixo corresponde à especificação do algoritmo paralelo proposto.

Algoritmo 3: Algoritmo Paralelo De Gomory-Hu Otimizado

```
   Entrada: G = (V, E_G, c_G) , proc_j, 0 ≤ j < p processos
   Saída: T = (V, E_T, c_T) uma árvore de cortes de G
1  se proc_j = 0 // processo mestre
2  então
3      processos_estados[j] ← V para todo j, 1 ≤ j < p ;
4      T ← {{V_G}, ∅} ;
5      distribua tarefas para todos os processos;
6      enquanto |V_T| < |V_G| faça
7          receba de proc_j resposta (s,t,S), onde {S, S̄} é um s-t-corte mínimo de G ;
8          se s e t pertencem ao mesmo nodo X de V_T // atualização da árvore
9          então
10             X_s ← X ∩ S;
11             X_t ← X ∩ S̄;
12             e ← {X_s, X_t};
13             c(e) ← c(S, S̄);
14             para cada aresta e' = {X, Y} ∈ E_T incidente em X na árvore T faça
15                 se Y ⊆ S então
16                     E_T ← E_T ∪ {{X_s, Y}} \ {{X, Y}};
17                 senão
18                     E_T ← E_T ∪ {{X_t, Y}} \ {{X, Y}};
19             V_T ← (V_T \ {X}) ∪ {X_s, X_t};
20             E_T ← E_T ∪ {e};
21         se |V_T| = |V_G| então
22             envie mensagem de finalização ao processo proc_j;
23         senão
24             (X, refinar) ← escolhe_pivô(T, processos_estados[proc_s]);
25             partição ← constroi_partição(X, T) ;
26             (s, t) ← escolhe_par_st(X);
27             processos_estados[proc_j] ← X;
28             envie tarefa (s,t,partição,refinar) para processo proc_s;
29     retorne T ;
30 senão
       // processo escravo
31     enquanto receber tarefas faça
32         receba tarefa (s,t,partição,refinar);
33         se tarefa = fim então
34             termine;
35         se contrações reduzem o núm. de vértices de G' então
36             se refinar então
37                 G' ← refinar_grafo_contraído(G', partição) ;
38             senão
39                 G' ← constrói_grafo_contraído(G, partição) ;
40         S ← corte_mínimo(G', s, t) ;
41         envie reposta (s,t,S) para proc_0;
```

O Algoritmo 3 recebe como entrada um grafo capacitado $G = (V, E_G, c_G)$ e devolve uma árvore de cortes $T = (V, E_T, c_T)$. Seja p o número total de processos, o processo $proc_0$ corresponde ao processo mestre e os processos $proc_j$, $1 \leq j \leq p$, os processos escravos. Na linha 3 o algoritmo inicializa o vetor *processos_estados* com o conjunto de vértices do grafo de entrada. Em seguida, o processo mestre envia *tarefas* contendo pares (s, t) de vértices para todos os processos escravos. No laço de repetição da linha 6 o processo mestre aguarda as respostas dos escravos. Ao receber uma resposta contento um par (s, t) e um *s-t* corte mínimo $S, \overline{S}$, o mestre verifica se os vértices s e t encontram-se no mesmo nodo da árvore e, caso afirmativo, a árvore é atualizada. A

atualização se dá da seguinte forma: toma-se o nodo $X \in T$ tal que $s, t \in X$ são criados os nodos X_s e X_t na árvore tal que X_s receberá os vértices de $X \cap S$ e X_t receberá os vértices de $X \cap \overline{S}$. Uma nova aresta $e = \{X_s, X_t\}$ em E_T com capacidade $c(e) = c(S, \overline{S})$ é adicionada à árvore T. Em seguida, entre as linhas 14 e 18, itera-se sobre todos as arestas $e' = \{X, Y\}$ atualizando-as conforme o lado do corte $\{S, \overline{S}\}$ em que os vértices de Y se encontram. Se, após a atualização, a condição $V_T = V_G$ for satisfeita a árvore de cortes foi construída e o algoritmo aguarda as respostas restantes para enviar mensagem de finalização para todos os processos. Caso contrário, o processo mestre escolhe um novo nodo pivô X com procedimento *escolhe_pivô*, na linha 24, para o processo $proc_j$ verificando o vértice do nodo armazenado em $processos\_estados[j]$ que está contido em um nodo de maior cardinalidade. Se for possível encontrar um nodo pivô desta forma, o processo mestre enviará comando ao escravo para que este *refine* o grafo contraído, caso contrário, o processo escravo fará as contrações a partir do grafo de entrada. Após escolhido o novo nodo pivô, o vetor $processos\_estados[j]$ é atualizado e uma nova tarefa é enviada ao processo $proc_j$.

Na linha 32 um processo escravo $proc_j$ recebe uma tarefa do processo mestre. Caso a tarefa recebida contenha uma informação indicando *fim* então o processo escravo finaliza. Em seguida, o processo escravo faz a construção do grafo contraído G' a partir do grafo contraído anterior através do procedimento *refinar_grafo_contraído*, se foi enviado na tarefa o comando para $refinar$. Caso contrário, o processo de contração será realizado sobre o grafo de entrada através do procedimento *constrói_grafo_contraído*. Após o processo de contração, o processo escravo $proc_j$ calcula o *s-t* corte mínimo $\{S, \overline{S}\}$ sobre o grafo G' e envia uma resposta (s, t, S) ao mestre.

4. Resultados Experimentais

Para execução dos experimentos foram utilizadas instâncias variadas que estão descritas na Tabela 1. As instâncias BLOG, PGRI, ROME e GEO são instâncias de grafos com dados reais: uma rede de blogs [Adamic and Glance 2005], uma rede de distribuição de energia [Watts and Strogatz 1998], uma rede representando as ruas de Roma [Storchi et al. 1999] e uma rede colaborativa científica [Batagelj and Mrvar]. Duas instâncias foram geradas utilizando os modelos Erdős–Rényi [Bollobás 2001] e Barabási–Albert [Albert and Barabási 2001]. As demais instâncias são grafos sintéticos que foram utilizadas em *benchmarks* para algoritmos de corte mínimo e árvores de cortes [Nagamochi et al. 1994, Chekuri et al. 1997, Goldberg and Tsioutsiouliklis 1999]. As instâncias do tipo *noi* são grafos aleatórios com conjuntos de vértices divididos em grupos de forma que as arestas internas ao grupo tendem a ter capacidades maiores do que as arestas com vértices de grupos distintos. As instâncias do tipo *path* são grafos aleatórios que possuem um caminho de tamanho k com arestas com grandes capacidades. Cada vértice fora desse caminho é conectado aleatoriamente ao caminho por uma aresta também com capacidade grande e as demais arestas são aleatórias e têm capacidades pequenas. As instâncias do tipo *tree* são semelhantes às instâncias do tipo *path*, porém constrói uma árvore aleatória com arestas com capacidades grandes conectando os primeiros k vértices.

O ambiente de execução foi um *cluster* homogêneo com 18 servidores biprocessados interligados por uma rede Ethernet de 1Gbps[1]. Os processadores são do tipo Intel®

[1] *Cluster* do Laboratório Central de Processamento de Alto Desempenho (LCPAD) da UFPR que é

Tabela 1. Características das instâncias utilizadas nos experimentos.

Instância	$\|V\|$	$\|E\|$
BA	2000	9995
DCYC	1024	2048
ER	2000	10079
GEO	3621	9461
NOI	1000	99900
PATH	2000	21990
BLOG	1222	16714
PGRI	4941	6594
ROME	3353	8870
TREE	2000	21990

Xeon® Processor E5-2670[2] com *clock* de 2.60GHz com 8 núcleos e 16 *threads* cada. Cada servidor possui 128GB de memória RAM e 20MB de memória *cache*. Em todos os experimentos foram utilizados um processo por servidor.

O código foi escrito em C/C++ e compilado com `gcc` utilizando nível máximo de otimização do compilador (`-O3`). Os *speedups* foram calculados como $S = T_S/T_P$, tal que T_S corresponde ao tempo da implementação do algoritmo sequencial e T_P corresponde ao tempo de execução da implementação do algoritmo paralelo com P processos. Todos os experimentos foram executados 10 vezes cada e os resultados reportados são baseados nas médias dos tempos de execução. A versão *GH-Opt* é a implementação do algoritmo proposto neste trabalho e o algoritmo *GH* é a implementação sem otimização.

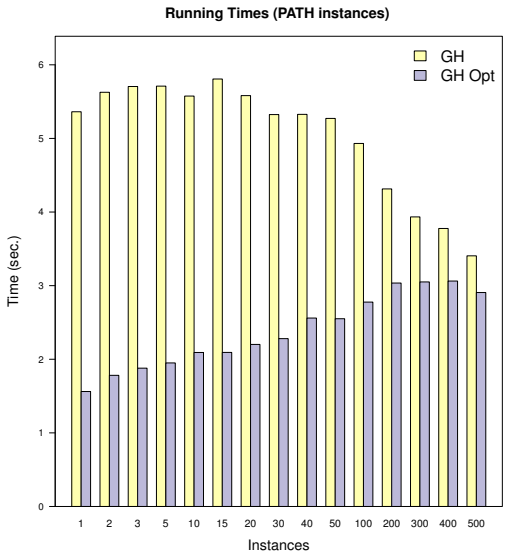

Figura 2. Tempos totais de execução para os algoritmos *GH* e *GH-Opt* para instâncias do tipo *path* variadas.

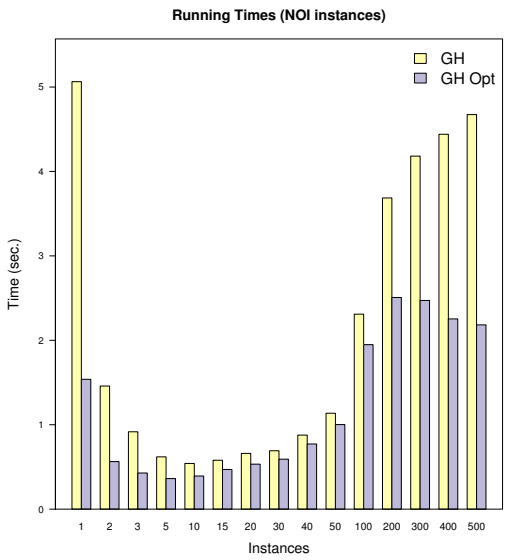

Figura 3. Tempos totais de execução para os algoritmos *GH* e *GH-Opt* para instâncias do tipo *noi* variadas.

financiado pela FINEP através dos projetos CT-INFRA/UFPR

[2]Intel® Xeon® Processor E5-2670 `http://ark.intel.com/pt-br/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI`

As figuras 2, 3 e 4 mostram os tempos totais de execução para as duas versões do algoritmo, com e sem as contrações de vértices otimizadas, utilizando instâncias do tipo *path*, *noi* e *tree*, respectivamente, com 1000 vértices e densidade de 20% variando o parâmetro k. O algoritmo proposto neste trabalho, descrito como *GH-Opt*, obteve os melhores resultados em todas as instâncias.

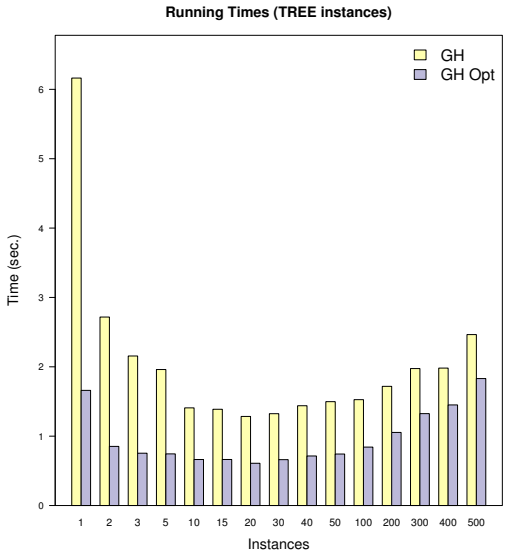

Figura 4. Tempos totais de execução para os algoritmos *GH* e *GH-Opt* para instâncias do tipo *tree* variadas.

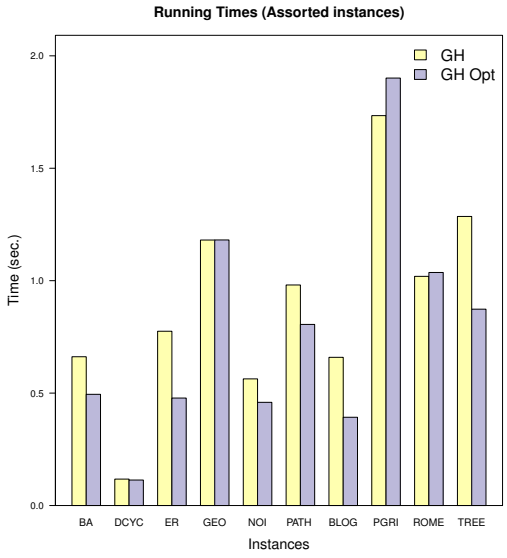

Figura 5. Tempos totais de execução para para os algoritmos *GH* e *GH-Opt* com as instâncias da Tabela 1.

A Figura 5 mostra os tempos totais de execução para as duas versões do algoritmo utilizando as instâncias da Tabela 1. Para a maioria das instâncias o algoritmo GH-Opt obteve melhores resultados. Apenas nas instâncias PGRI e ROME o algoritmo GH-Opt finalizou com tempos de execução maiores do que a versão GH. Essas instâncias, em particular, não contém cortes balanceados e o algoritmo de GH-Opt não tem muitas oportunidades de otimizar as contrações. Nas instâncias GEO e DCYC os tempos foram iguais para ambas as implementações.

Na Figura 6 é representado o *speedup* do algoritmo Gomory-Hu com contrações otimizadas utilizando instâncias do tipo *noi* com 2000 vértices e densidade de 20% variando o parâmetro k. Para o limite de 18 máquinas utilizadas na execução dos experimentos os *speedups* foram lineares.

Na Figura 7 são representados os tempos médios de contração. É possível constatar os ganhos que a estratégia proposta concede à implementação através da avaliação dos tempos totais do algoritmo.

5. Conclusão

Nesse trabalho foi proposta uma estratégia eficiente para realizar as contrações na versão paralela do algoritmo de Gomory-Hu. A ideia principal é fazer com que os processos escravos aproveitem instâncias de grafos contraídos de passos anteriores sem a necessidade de refazer todas as contrações no grafo de entrada durante a execução de cada tarefa.

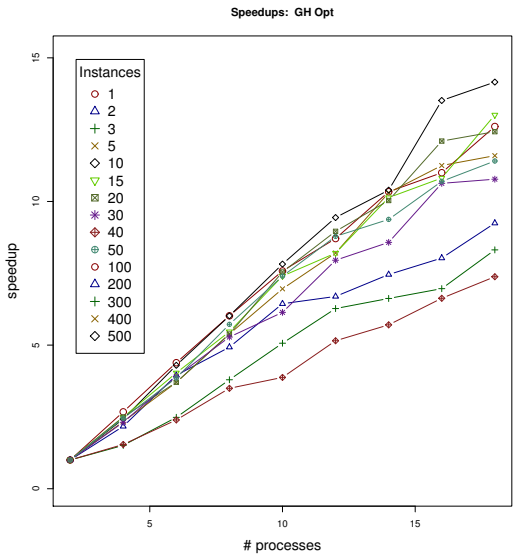

Figura 6. *Speedup* do algoritmo *GH-Opt* utilizando instâncias do tipo *noi* com 2000 vértices.

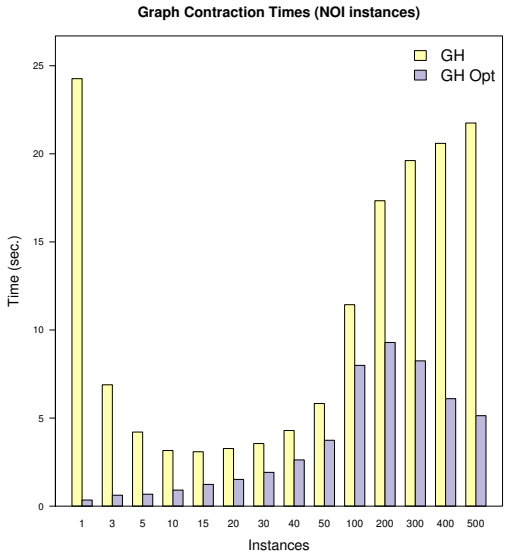

Figura 7. Tempos médios de contração para os algoritmos *GH* e *GH-Opt*.

De forma geral, a implementação dessa otimização requer que o processo mestre tenha controle sobre as tarefas que foram enviadas para cada escravo, armazenando-as para que sejam utilizadas quando as respostas chegarem. Os processos escravos, por sua vez, precisam tomar decisões sobre quando aproveitar, ou não, uma instância de grafo contraído.

Para avaliar a eficiência desta versão foram realizados experimentos em um *cluster* de alto desempenho. Foram realizados testes de *speedup* e comparativos entre as versões paralelas. Os experimentos foram feitos em vários conjuntos de instâncias que incluíram grafos sintéticos e grafos reais. Os resultados demonstraram *speedups* próximos de lineares para a maioria das instâncias. Nas comparações foi possível comprovar os ganhos obtidos com a estratégia proposta.

Entre os trabalhos futuros estão a paralelização do processo mestre, já que esse pode tornar-se um "gargalo"a partir de um determinado número de processos escravos. Outro trabalho futuro é paralelizar a construção do grafo contraído utilizando a biblioteca OpenMP, dividindo o esforço em várias *threads*.

Referências

[Adamic and Glance 2005] Adamic, L. A. and Glance, N. (2005). The political blogosphere and the 2004 u.s. election: Divided they blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, pages 36–43. ACM.

[Albert and Barabási 2001] Albert, R. and Barabási, A.-l. (2001). Statistical mechanics of complex networks. *Reviews of Modern Physics*, page 54.

[Batagelj and Mrvar] Batagelj, V. and Mrvar, A. Pajek datasets. Disponível para download online em `http://vlado.fmf.uni-lj.si/pub/networks/data`.

[Bollobás 2001] Bollobás, B. (2001). *Random Graphs*. Cambridge University Press, second edition. Cambridge Books Online.

[Chekuri et al. 1997] Chekuri, C. S., Goldberg, A. V., Karger, D. R., Levine, M. S., and Stein, C. (1997). Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 324–333, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Cohen 2013] Cohen, J. (2013). *Algoritmos Paralelos Para Árvores De Cortes E Medidas De Centralidade Em Grafos*. PhD thesis, Universidade Federal do Paraná - UFPR.

[Cohen et al. 2012] Cohen, J., Rodrigues, L. A., and Duarte Jr., E. P. (2012). A parallel implementation of gomory-hu's cut tree algorithm. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 124–131, Washington, DC, USA. IEEE Computer Society.

[Cohen et al. 2011] Cohen, J., Rodrigues, L. A., Silva, F., Carmo, R., Guedes, A. L. P., and Duarte, E. P. (2011). Parallel implementations of gusfield's cut tree algorithm. In *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ICA3PP'11, pages 258–269, Berlin, Heidelberg. Springer-Verlag.

[Duarte Jr et al. 2004] Duarte Jr, E., Santini, R., and Cohen, J. (2004). Delivering packets during the routing convergence latency interval through highly connected detours. In *Dependable Systems and Networks, 2004 International Conference on*, pages 495–504.

[Engelberg et al. 2006] Engelberg, R., Könemann, J., Leonardi, S., and Naor, J. (2006). Cut problems in graphs with a budget constraint. In *LATIN 2006: Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 435–446. Springer Berlin Heidelberg.

[Goldberg and Tsioutsiouliklis 1999] Goldberg, A. V. and Tsioutsiouliklis, K. (1999). Cut tree algorithms. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 376–385, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Gomory and Hu 1961] Gomory, R. E. and Hu, T. C. (1961). Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570.

[Gusfield 1990] Gusfield, D. (1990). Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155.

[Kamath and Caverlee 2011] Kamath, K. Y. and Caverlee, J. (2011). Transient crowd discovery on the real-time social web. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 585–594. ACM.

[Nagamochi and Ibaraki 2008] Nagamochi, H. and Ibaraki, T. (2008). *Algorithmic Aspects of Graph Connectivity*. Algorithmic Aspects of Graph Connectivity. Cambridge University Press.

[Nagamochi et al. 1994] Nagamochi, H., Ono, T., and Ibaraki, T. (1994). Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(1-3):325–341.

[Storchi et al. 1999] Storchi, G., Dell'Olmo, P., and Gentili, M. (1999). Road network of the city of rome.

[Tuncbag et al. 2010] Tuncbag, N., Salman, F. S., Keskin, O., and Gursoy, A. (2010). Analysis and network representation of hotspots in protein interfaces using minimum cut trees. *Proteins: Structure, Function, and Bioinformatics*, 78(10):2283–2294.

[Watts and Strogatz 1998] Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. 393(June):440–442.