Improved Parallel Implementations of Gusfield's Cut Tree Algorithm

Jaime Cohen
Universidade Estadual de Ponta Grossa
Departamento de Informática
jaimecohen@gmail.com

Luiz A. Rodrigues
Universidade Estadual do Oeste do Paraná
Colegiado de Ciência da Computação
luiz.rodrigues@unioeste.br

Elias P. Duarte Jr.
Universidade Federal do Paraná
Departamento de Informática
elias@inf.ufpr.br

Abstract

This work presents parallel versions of Gusfield's cut tree algorithm and the proposal of two heuristics aimed at improving their performance. Cut trees are a compact representation of the edge-connectivity between every pair of vertices of an undirected graph. Cut trees have a vast number of applications in combinatorial optimization and in the analysis of networks originated in many applied fields. However, surprisingly few works have been published on the practical performance of cut tree algorithms. This paper describes two parallel versions of Gusfield's cut tree algorithm and presents extensive experimental results which show a significant speedup on most real and synthetic graphs in our dataset.

Resumo

Este artigo apresenta duas implementações paralelas do algoritmo de Gusfield para a construção de árvores de cortes de um grafo e a proposta de duas heurísticas para melhorar seus desempenhos. Árvores de cortes são uma representação compacta da aresta-conectividade entre todos os pares de vértices de um grafo não direcionado. As árvores de cortes têm inúmeras aplicações em otimização combinatória e na análise de redes complexas. Entretanto, poucos trabalhos sobre o desempenho prático dos algoritmos para árvores de cortes foram publicados. Este artigo descreve duas versões paralelas do algoritmo de Gusfield e heurísticas para melhorar os seus desempenhos. Resultados experimentais mostram que as implementações atingem speedups altos, tanto em instâncias sintéticas como em grafos obtidos de redes reais.

1 Introdução

Este trabalho apresenta a descrição de duas implementações paralelas do algoritmo de Gusfield para árvores de cortes e resultados experimentais que mostram um bom desempenho dessas implementações em um conjunto variado de instâncias do problema. Heurísticas são propostas para melhorar a eficiência das implementações.

Duas implementações para arquitetura MIMD (*Multiple Instruction stream, Multiple Data stream*) foram elaboradas. A primeira implementação foi desenvolvida para ser executada em *clusters* computacionais e utilizou a biblioteca MPI (*Message Passing Interface*) [1]. A outra implementação é voltada para a arquitetura de multiprocessamento simétrico ou SMP (*Symmetric Multiprocessing*) e utiliza a biblioteca OpenMP [2].

Árvores de cortes são uma importante estrutura combinatória que representa a aresta-conectividade entre todos os pares de vértices de um multigrafo capacitado. As árvores de cortes têm inúmeras aplicações diretas, por exemplo [3, 4, 5, 6, 7, 8], e também são utilizadas na solução de inúmeros outros problemas combinatórios em áreas como particionamento de grafos, conectividade e roteamento, por exemplo [9, 10, 11, 12, 13, 14, 15].

Apesar da importância das árvores de cortes devido à sua extensa lista de aplicações, nenhuma implementação paralela de algoritmos para árvores de cortes está publicamente disponível e não temos conhecimento de que algum estudo experimental acerca do assunto tenha sido publicado.

Uma análise superficial do algoritmo de Gusfield pode levar à conclusão de que o seu laço principal não pode ser paralelizado com eficiência devido à dependência entre as iterações. Entretanto, os resultados apresentados adiante mostram que o algoritmo de Gusfield pode ser paralelizado

com relativa facilidade e de forma a produzir *speedups*[1] significativos.

Dado um grafo não direcionado com n vértices, o algoritmo sequencial de Gusfield executa $n-1$ chamadas a um algoritmo de fluxo-máximo. Cada execução encontra um corte mínimo no grafo de entrada que separa um vértice, chamado de *origem*, de seu vizinho na árvore em construção. A estratégia usada para paralelizar o algoritmo foi a de executar essas $n-1$ iterações em paralelo e de forma otimista. À medida que as execuções do fluxo máximo terminam, o algoritmo faz tentativas de modificar a árvore de cortes em construção. Essa tentativa será bem sucedida somente se os vértices separados pelo corte mínimo encontrado ainda forem vizinhos na árvore. Caso outra execução paralela tenha modificado a árvore de forma que os dois vértices sendo separados não sejam mais vizinhos na árvore, o corte mínimo encontrado é rejeitado e um novo corte mínimo separando a origem de seu novo vizinho na árvore deve ser calculado.

O desempenho das implementações paralelas do algoritmo de Gusfield depende da estrutura do grafo. Por isso, executamos experimentos em um conjunto de grafos que cobre diversas classes de grafos sintéticos bem como algumas redes complexas reais.

O restante do artigo está assim organizado. Na Seção 2 descrevemos o algoritmo de Gusfield para computar uma árvore de cortes. A Seção 3 trata da paralelização de algoritmos para árvores de cortes. Nela são apresentadas justificativas para a escolha de paralelizar o algoritmo de Gusfield e são descritas as implementações com MPI e OpenMP. A Seção 4 apresenta os resultados experimentais e a Seção 5 traz as conclusões.

2 Algoritmos para Árvores de Cortes

Iniciamos com algumas definições. Seja um grafo não direcionado $G = (V, E)$ cujas arestas estão associadas a capacidades por uma função $c : E \to \mathbb{Z}_+$. Um *corte* de G é uma bipartição de V. O conjunto de arestas que *cruzam* o corte $\{X, Y\}$ é $E_G(X, Y) = \{\{u, v\} \in E(G) \mid u \in X \text{ e } v \in Y\}$. A *capacidade* de um corte é o somatório das capacidades das arestas que cruzam o corte.

Sejam s e t dois vértices de V. Um s-t-corte de G é um corte $\{X, V - X\}$ tal que $s \in X$ e $t \in V - X$. Um s-t-corte mínimo é um s-t-corte de capacidade mínima. A *aresta-conectividade local* entre s e t, usualmente denotada por $\lambda_G(s, t)$, é a capacidade de um s-t-corte mínimo.

Considere o problema de calcular a aresta-conectividade local entre todos os pares de vértices de um grafo não dirigido. Isto é, desejamos calcular a capacidade de um corte mínimo entre cada par de vértices do grafo. A solução ingênua consiste em executar $\binom{n}{2}$ algoritmos de corte mínimo (ou fluxo máximo), um para cada par de vértices. Em 1961, R. E. Gomory e T. C. Hu [16] mostraram que o cômputo de apenas $n-1$ fluxos máximos é suficiente. A solução consiste na construção de uma árvore capacitada sobre o conjunto de vértices do grafo que representa os valores das conectividades locais para todos os pares de vértices. Abaixo definimos esta árvore.

Uma *árvore de fluxo equivalente* de um grafo G é uma árvore capacitada T sobre o conjunto de vértices V tal que para todos os pares de vértices $u, v \in V$, a menor capacidade de uma aresta no caminho entre u e v em T é igual à conectividade local entre u e v em G, isto é, $\lambda_G(u, v) = \lambda_T(u, v)$, para todos $u, v \in V$.

Uma *árvore de cortes* de um grafo G é uma árvore de fluxo equivalente T tal que o corte induzido pela remoção da aresta de capacidade mínima do caminho entre u e v em T é um u-v-corte mínimo de G para todos $u, v \in V$. As árvores de cortes também são conhecidas como *árvores de Gomory-Hu* [17].

A Figura 1(b) mostra uma árvore de cortes do grafo da Figura 1(a). A Figura 1(a) mostra um corte mínimo entre os vértices A e F induzido pela remoção da aresta $\{C, D\}$ da árvore de cortes.

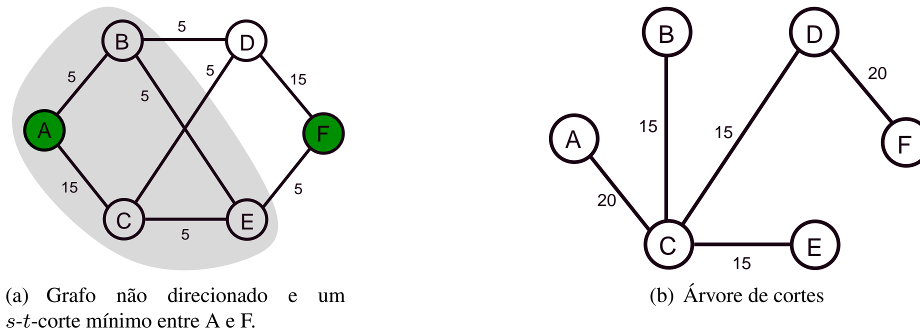
O algoritmo de Gusfield para encontrar uma árvore de fluxo equivalente de um multigrafo capacitado consiste de $n-1$ iterações. Cada uma delas calcula um s-t-corte mínimo. O pseudo-código pode ser visto no Algoritmo 1. O conjunto de vértices é representado por $\{1, 2, \ldots, |V|\}$. O algoritmo inicia com uma árvore com topologia de estrela em que o nodo 1 é o centro (linhas 1-2). A cada iteração (linhas 3-6), o algoritmo escolhe um vértice diferente, s, $s \geq 2$, como origem. Essa escolha determina o vértice de destino, t, como o vizinho atual de s na árvore. Um s-t-corte mínimo é computado e a árvore em construção é modificada: cada nodo t' vizinho de t com $t' > s$ que esteja no lado de s do s-t-corte é desconectado de t e reconectado a s. O algoritmo termina quando cada nodo de 2 até $|V|$ tiver sido a origem de um s-t-corte de uma iteração.

A implementação do algoritmo de Gusfield é simples, pois pode utilizar qualquer algoritmo de fluxo máximo sem nenhuma modificação. A versão descrita no Algoritmo 1 calcula uma árvore de fluxo equivalente. Uma pequena modificação o faz encontrar uma árvore de cortes: na linha 8, permita que qualquer vizinho de t que esteja em X seja reconectado a s.

3 A Paralelização de Algoritmos para Árvores de Corte

A paralelização de algoritmos para árvores de corte envolve pelo menos duas escolhas:

[1] O *speedup* é a razão entre o tempo de execução sequencial e o tempo de execução paralelo.



(a) Grafo não direcionado e um s-t-corte mínimo entre A e F.

(b) Árvore de cortes

Figura 1. Uma árvore de cortes e um s-t-corte mínimo.

Algoritmo 1 Algoritmo de Gusfield - Versão Sequencial

Entrada $G = (V_G, E_G, c)$, um grafo não-orientado
Saída $T = (V_T, E_T, f)$, uma árvore de fluxo equivalente de G

1: **para** $i = 1$ até $|V_G|$ **faça**
2: $tree_i \leftarrow 1$
// $|V_G| - 1$ iterações (fluxos máximos)
3: **para** $s \leftarrow 2$ até $|V_G|$ **faça**
4: $t \leftarrow tree_s$
5: $flow_s \leftarrow$ fluxo-máximo(s, t)
6: $\{X, \overline{X}\} \leftarrow s$-$t$-corte mínimo
// atualiza a árvore
7: **para** $u \in V_G,\ u > s$ **faça**
8: **se** $tree_u = t$ **e** $u \in X$ **então**
9: $tree_u \leftarrow s$
// Devolve uma árvore de fluxo equivalente
10: $V_T \leftarrow V_G$
11: $E_T \leftarrow \emptyset$
12: **para** $s \leftarrow 2$ até $|V_G|$ **faça**
13: $E_T \leftarrow E_T \cup \{s, tree_s\}$
14: $f(\{s, tree_s\}) \leftarrow flow_s$
15: **devolva** $T = (V_T, E_T, f)$

- a escolha do algoritmo: algoritmo de Gomory-Hu ou algoritmo de Gusfield;
- a escolha do nível de paralelismo a ser implementado.

Assim como o algoritmo de Gusfield, o algoritmo de Gomory-Hu também faz $n - 1$ chamadas a um algoritmo de s-t-cortes mínimos. Entretanto, o algoritmo de Gomory-Hu contrai cada lado do corte encontrado e usa recursão nos dois grafos obtidos.

Descrevemos a seguir várias razões para escolher paralelizar o algoritmo de Gusfield em oposição ao algoritmo de Gomory-Hu.

A existência de s-t-cortes mínimos balanceados favorece o algoritmo de Gomory-Hu porque o tamanho do grafo é reduzido a cada iteração que utilize um corte balanceado. Entretanto, a maioria dos grafos reais e grafos gerados por modelos aleatórios como o Modelo de Erdös-Rényi (ER) e o Modelo de Barabási-Albert (BA) raramente possuem cortes balanceados. Isso implica que os tamanhos dos subproblemas não se tornam significativamente menores. Ainda mais importante, a inexistência de cortes balanceados sugere que seja difícil garantir um balanceamento de carga entre os processos em implementações do algoritmo de Gomory-Hu.

O algoritmo de Gusfield calcula fluxos máximos sempre no mesmo grafo de entrada. Assim, cada processo pode manter uma cópia do grafo de entrada de forma que não seja necessária a transmissão de grafos ao longo da execução paralela. Entretanto, uma implementação paralela do algoritmo de Gomory-Hu necessita transferir mais dados entre os processos uma vez que o grafo sofre transformações.

Trataremos agora do grau de paralelismo a ser explorado e justificaremos a opção por paralelizar o laço principal do algoritmo de Gusfield. A alternativa dessa escolha é utilizar um algoritmo paralelo para o problema do fluxo máximo. Entretanto, o problema do fluxo máximo é difícil de ser paralelizado. Apesar de extensa pesquisa sobre fluxo em redes em paralelo, os trabalhos experimentais do assunto reportam *speedups* pequenos (ex. [18, 19]) devido ao sincronismo necessário para a execução dos algoritmos. Por outro lado, a implementação paralela do algoritmo de Gusfield pode resolver um fluxo máximo por *linha de execução* ou processo, sem sincronização, e alcançar *speedups* elevados, como veremos na Seção 4.

As nossas implementações são baseadas no algoritmo *Push-Relabel* para o problema do fluxo máximo [20] e utilizam o código HIPR[2], desenvolvido por B.V. Cherkassky e A.V. Goldberg [21]. Reportamos resultados da versão do algoritmo de Gusfield que encontra uma árvore de fluxos equivalentes.

[2]De propriedade de IG Systems, Inc. Copyright 1995-2004. Disponível livremente para fins de pesquisa.

3.1 Implementação com MPI

Message Passing Interface (MPI) é uma biblioteca de rotinas para o gerenciamento de processos e para a troca de mensagens em arquiteturas paralelas com memória distribuída [2].

A implementação do algoritmo de Gusfield com MPI utiliza o modelo mestre-escravo. O processo mestre é responsável por enviar os subproblemas aos escravos e também por manipular a árvore de cortes em construção.

Os processos são numerados de 0 até $p-1$. O processo mestre é o $proc_0$. Cada processo mantém uma cópia do grafo de entrada. O processo mestre cria tarefas e as envia para os processos escravos. Uma tarefa é definida por um par de nodos (s,t) que definem a origem e o destino de uma instância do problema do fluxo máximo.

Quando um processo escravo conclui a execução do fluxo máximo, ele envia ao processo mestre o valor do fluxo máximo e uma lista dos nodos que definem o corte mínimo encontrado. Ao receber a resposta, o processo mestre pode atualizar a árvore de cortes em construção desde que o nodo s ainda seja vizinho do nodo t. A atualização da árvore de cortes é semelhante à que é feita no algoritmo sequencial. Se s e t não forem mais vizinhos no momento desse processamento, então dizemos que a tarefa associada ***falhou*** e outra tarefa cuja origem é s e o destino é o novo vizinho de s na árvore é produzida.

A estrutura do grafo influencia no número de tarefas falhas. Quando o conjunto X que define o s-t-corte $\{X, V-X\}$ é pequeno, a árvore sofre poucas alterações e, portanto, poucas tarefas podem falhar. O *speedup* da execução paralela depende do número de tarefas falhas ao longo de sua execução.

3.2 Implementação com OpenMP

OpenMP (*Open Multi-Processing*) é uma interface de programação de aplicação (API) projetada para facilitar a programação paralela em arquiteturas de memória compartilhada SMP (*Symmetric Multiprocessing*). Essa API fornece diretivas que estendem as linguagens Fortran, C e C++, que definem como o processamento é compartilhado por *threads* a serem executadas em diferentes processadores ou núcleos e também como os dados da memória compartilhada são acessados pelas *threads* [2].

A adaptação do algoritmo de Gusfield em OpenMP foi feita pela paralelização do laço principal que faz as $n-1$ chamadas da rotina que calcula o fluxo máximo.

Seja k o número máximo pré-definido de *threads*. O algoritmo utiliza uma estratégia otimista e encontra k s-t-cortes mínimos em paralelo. Cada um desses cortes é usado em uma tentativa de modificar a árvore de cortes em construção. Cada *thread*, após encontrar um s-t-corte, verifica se o vértice de destino, t, ainda é o vizinho de s na árvore em construção. Isso não ocorre quando um corte encontrado por outra *thread*, que foi bem sucedido em modificar a árvore, separou s de t. Nesse caso, dizemos que a *thread* ***falhou*** e um outro s-t-corte é calculado para separar s do seu novo vizinho na árvore. Caso a *thread* seja bem sucedida em separar s de t, ela atualiza a árvore em construção. Os testes e as modificações da árvore são feitos em exclusão mútua, dentro de uma região crítica, para garantir a correção do algoritmo.

4 Resultados Experimentais

A seguir apresentamos a descrição dos experimentos e os resultados obtidos com a paralelização do algoritmo de Gusfield.

Ambiente Computacional Os experimentos com MPI foram executados em um *cluster* composto de 14 computadores com processadores Intel Core 2 Quad 2.4 GHz, com 2 GB de memória e 4MB de memória *cache* interconectados por uma rede Gigabit Ethernet. Os experimentos com OpenMP foram executados em um computador com processador Quad-Core AMD Opteron com 2.8 GHz de *clock* de 8 núcleos, 8 GB de memória e 512 KB de memória *cache*. As implementações foram escritas em linguagem C e compiladas com o *gcc* com nível de otimização -O3.

Descrição do Conjunto de Instâncias O conjunto de dados de testes é composto por 10 grafos representando diferentes classes de grafos reais e sintéticos, como mostrado na Tabela 1. Os 4 primeiros grafos foram obtidos a partir de redes reais: 2 redes de colaboradores em trabalhos científicos [22, 23], uma rede de transmissão elétrica [24] e uma rede *peer-to-peer* [22]. Duas redes foram geradas por modelos aleatórios: o modelo binomial de Erdös-Rényi (ER) [25] e o modelo de conexões preferenciais de Barabási-Albert (BA) [26]. As outras 4 instâncias são grafos sintéticos de diferentes tipos utilizados em outros trabalhos experimentais sobre cortes mínimos e árvores de cortes [27, 28].

Medidas de Desempenho Paralelo O *speedup*, denotado por S, é uma medida frequentemente utilizada para quantificar a melhora do tempo de execução paralelo em relação ao sequencial e é definido como $S = T_S/T_P$, onde T_S é o tempo da execução sequencial e T_P é o tempo da execução paralela utilizando P processadores. A *eficiência*, E, é o *speedup* normalizado e é calculada como $E = S/P$.

Resultados Experimentais Execuções preliminares foram feitas para definir as melhores formas de escalonamento de tarefas. Nas execuções com MPI, os melhores

Tabela 1. Lista de grafos utilizados nos experimentos.

Grafo	$\|V\|$	$\|E\|$
TREE	1500	563625
NOI	1500	562125
CA-HepPh	11204	235238
BA	10000	49995
ER	10000	49841
P2P-Gnutella	10876	39994
PATH	2000	21990
GeoComp	3621	9461
PowerGrid	4941	6594
DBLCYC	1024	2048

resultados ocorreram com a execução do processo mestre e um escravo em uma mesma máquina e os demais escravos em máquinas exclusivas. Essa distribuição dos processos se justifica pois o trabalho do mestre é significativamente menor do que o dos escravos, e, assim, pelo menos um escravo se beneficia ao se comunicar diretamente com o mestre que está na mesma máquina. A implementação com OpenMP teve melhor desempenho com a estratégia dinâmica de escalonamento do OpenMP, que distribui as tarefas em tempo de execução para *threads* ociosas de forma a proporcionar algum balanceamento de carga.

Os resultados apresentados nas próximas duas seções abaixo se baseiam em implementações que não utilizam heurísticas para evitar laços falhos, que, contudo, já produziram bons *speedups*. Entretanto, mostraremos adiante que os resultados podem ser melhorados com a utilização de duas heurísticas relativamente simples.

Resultados com MPI Os resultados dos experimentos com a implementação em MPI são mostrados na Tabela 2 e na Figura 2. Os dados apresentados correspondem a médias de 10 execuções para cada situação. A eficiência das execuções paralelas foi alta, em particular para as instâncias P2P-Gnutella, BA, ER, NOI e TREE. Nas instâncias DBLCY, PATH e CA-HepPH a eficiência foi em torno de 0.60. Somente nas instâncias Geocomp e Powergrid a eficiência foi inferior a 0.50.

A implementação do algoritmo de Gusfield com MPI produziu bons *speedups* sem a necessidade de heurísticas para evitar laços falhos. Entretanto, nos casos de instâncias que produzem um maior número de falhas, algumas heurísticas podem ser consideradas. Uma delas consiste em escolher o s-t-corte mínimo $\{X, \overline{X}\}$, $s \in X$, que minimize $|X|$ na tentativa de reduzir o número de modificações da árvore em construção. Outra heurística de consequência menos direta consiste em escolher os vértices de origem em alguma ordem particular, por exemplo, em ordem não crescente do grau do vértice.

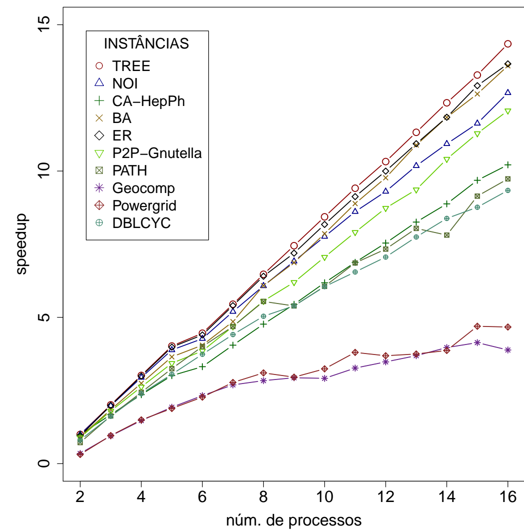


Figura 2. Speedups das execuções com MPI.

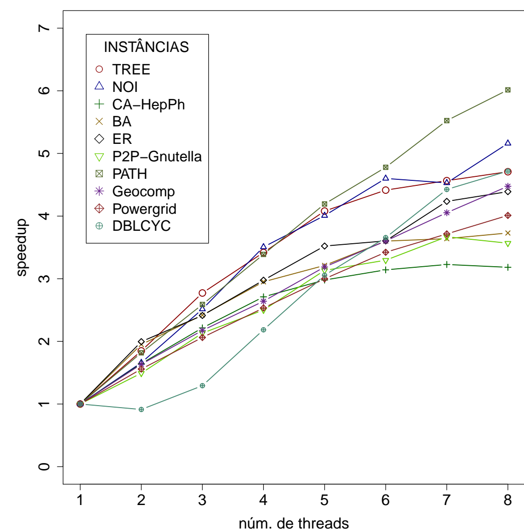


Figura 3. Speedups das execuções com OpenMP.

Resultados com OpenMP Os resultados obtidos com a implementação baseada em OpenMP do algoritmo de Gusfield executados em uma máquina com 8 núcleos são mostrados na Tabela 3 e na Figura 3. A média dos valores de eficiência foi acima de 0.50 na maioria das instâncias.

O melhor *speedup* obtido com OpenMP em grafos reais

Tabela 2. Resultados da implementação do algoritmo de Gusfield com MPI. Médias dos tempos de execução, speedups (S) e eficiência (E) são mostrados para números pares de processos.

Number of	CA-HepPh			Geocomp			Powergrid			P2P-Gnutella			BA		
procs.	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
sequential	475.01	-	-	2.25	-	-	3.85	-	-	65.23	-	-	87.71	-	-
2	479.93	0.99	0.49	6.45	0.35	0.17	12.31	0.31	0.16	73.24	0.89	0.45	95.49	0.92	0.46
4	201.56	2.36	0.59	1.53	1.47	0.37	2.57	1.50	0.37	24.88	2.62	0.66	31.94	2.75	0.69
6	143.35	3.31	0.55	0.97	2.32	0.39	1.69	2.27	0.38	16.94	3.85	0.64	21.66	4.05	0.67
8	99.64	4.77	0.60	0.79	2.84	0.35	1.24	3.10	0.39	11.73	5.56	0.69	14.42	6.08	0.76
10	76.84	6.18	0.62	0.77	2.91	0.29	1.19	3.24	0.32	9.24	7.06	0.71	11.16	7.86	0.79
12	63.01	7.54	0.63	0.65	3.48	0.29	1.04	3.69	0.31	7.47	8.73	0.73	8.98	9.77	0.81
14	53.50	8.88	0.63	0.57	3.97	0.28	0.99	3.87	0.28	6.27	10.41	0.74	7.41	11.83	0.85
16	46.52	10.21	0.64	0.58	3.88	0.24	0.82	4.67	0.29	5.41	12.06	0.75	6.45	13.60	0.85
Number of	**DBLCYC**			**ER**			**NOI**			**PATH**			**TREE**		
procs.	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
sequential	11.13	-	-	104.23	-	-	384.84	-	-	5.42	-	-	236.78	-	-
2	13.83	0.81	0.40	109.90	0.95	0.47	385.61	1.00	0.50	7.52	0.72	0.36	237.10	1.00	0.50
4	4.66	2.39	0.60	34.82	2.99	0.75	130.75	2.94	0.74	2.22	2.44	0.61	78.53	3.02	0.75
6	2.98	3.74	0.62	23.70	4.40	0.73	90.11	4.27	0.71	1.35	4.00	0.67	53.11	4.46	0.74
8	2.21	5.04	0.63	16.27	6.40	0.80	63.36	6.07	0.76	0.98	5.54	0.69	36.59	6.47	0.81
10	1.84	6.05	0.60	12.75	8.17	0.82	49.59	7.76	0.78	0.89	6.06	0.61	28.07	8.43	0.84
12	1.58	7.06	0.59	10.43	10.00	0.83	41.38	9.30	0.77	0.74	7.33	0.61	22.94	10.32	0.86
14	1.33	8.38	0.60	8.80	11.84	0.85	35.21	10.93	0.78	0.69	7.81	0.56	19.20	12.33	0.88
16	1.19	9.34	0.58	7.63	13.66	0.85	30.37	12.67	0.79	0.56	9.73	0.61	16.50	14.35	0.90

Tabela 3. Resultados da implementação do algoritmo de Gusfield com OpenMP. A tabela mostra as médias dos tempos de execução, dos speedups (S) e da eficiência (E) para cada instância.

Number of	CA-HepPh			Geocomp			Powergrid			P2P-Gnutella			BA		
threads	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
1	517.07	-	-	2.10	-	-	3.03	-	-	65.79	-	-	87.66	-	-
2	314.32	1.65	0.82	1.29	1.63	0.82	1.94	1.56	0.78	44.10	1.49	0.75	44.94	1.95	0.98
3	233.42	2.22	0.74	0.97	2.17	0.72	1.47	2.06	0.69	30.87	2.13	0.71	36.24	2.42	0.81
4	190.77	2.71	0.68	0.80	2.64	0.66	1.20	2.53	0.63	26.31	2.50	0.63	29.71	2.95	0.74
5	173.56	2.98	0.60	0.66	3.19	0.64	1.01	3.00	0.60	21.00	3.13	0.63	27.29	3.21	0.64
6	164.59	3.14	0.52	0.58	3.60	0.60	0.88	3.42	0.57	19.96	3.30	0.55	24.35	3.60	0.60
7	160.22	3.23	0.46	0.52	4.05	0.58	0.82	3.72	0.53	17.93	3.67	0.52	24.08	3.64	0.52
8	162.48	3.18	0.40	0.47	4.48	0.56	0.76	4.01	0.50	18.44	3.57	0.45	23.50	3.73	0.47
Number of	**DBLCYC**			**ER**			**NOI**			**PATH**			**TREE**		
threads	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
1	10.12	-	-	115.16	-	-	585.37	-	-	5.60	-	-	315.09	-	-
2	11.09	0.91	0.46	57.71	2.00	1.00	352.84	1.66	0.83	3.08	1.82	0.91	170.51	1.85	0.92
3	7.83	1.29	0.43	47.71	2.41	0.80	232.72	2.52	0.84	2.16	2.59	0.86	113.66	2.77	0.92
4	4.63	2.18	0.55	38.69	2.98	0.74	166.95	3.51	0.88	1.65	3.39	0.85	92.18	3.42	0.85
5	3.31	3.06	0.61	32.70	3.52	0.70	146.06	4.01	0.80	1.33	4.19	0.84	77.23	4.08	0.82
6	2.77	3.66	0.61	31.96	3.60	0.60	127.23	4.60	0.77	1.17	4.78	0.80	71.38	4.41	0.74
7	2.29	4.43	0.63	27.19	4.23	0.60	129.23	4.53	0.65	1.01	5.52	0.79	69.02	4.57	0.65
8	2.14	4.72	0.59	26.23	4.39	0.55	113.44	5.16	0.65	0.93	6.01	0.75	66.93	4.71	0.59

foi de 4.48 e em grafos sintéticos foi de 6.01. Os piores *speedups* em redes reais e em redes sintéticas foram de 3.18 and 3.73, respectivamente.

Outros bons resultados foram obtidos em experimentos em um computador com 16 núcleos. O melhor *speedup* foi de 9.4 utilizando 16 *threads* no grafo NOI. Um experimento com o grafo ER com 3000 nodos e grau médio 5 alcançou um *speedup* de 9.2 utilizando 10 *threads*.

A implementação com OpenMP em arquitetura com memória compartilhada não é tão escalável quanto a implementação com MPI em *clusters*. A memória compartilhada é um gargalo na execução com OpenMP, pois a utilização de memória RAM se torna excessiva nos grafos maiores e um número elevado de *threads*, já que cada *thread*

necessita uma cópia do grafo para a execução de algoritmos de fluxo máximo em diferentes pares de vértices.

Assim como ocorre na implementação com MPI, heurísticas podem reduzir o número de laços falhos e melhorar o desempenho do algoritmo em alguns casos.

Heurísticas para Redução do Número de Laços Falhos Duas heurísticas foram implementadas na tentativa de reduzir o número de laços falhos. A primeira consiste em verificar se o corte trivial formado pelo vértice de origem é um corte mínimo. A escolha desse corte implica que a separação do vértice e de seu vizinho não produz modificações na árvore em construção e, portanto, não interfere com o trabalho paralelo dos outros processos.

A segunda heurística consiste em ordenar os vértices de forma a maximizar as chances dos cortes encontrados terem margens pequenas do lado da origem. A ordenação escolhida foi a não crescente de graus dos vértices pois ela implica que se o s-t-corte mínimo é trivial, então $\{\{s\}, V - \{s\}\}$ é um s-t-corte mínimo cuja utilização não modifica a árvore em construção.

A Tabela 4 mostra a redução do número de falhas nas execuções da implementação usando OpenMP com 8 *threads* e da implementação usando MPI em 16 processos em todas as 10 instâncias do conjunto de testes. Pode-se verificar que as duas heurísticas apresentadas foram suficientes para reduzir o número de laços falhos a valores muito baixos. A Tabela 5 mostra os *speedups* das execuções com e sem as heurísticas da implementação em MPI com 16 processos.

5 Conclusão

As árvores de cortes são estruturas combinatórias amplamente utilizadas. Dois algoritmos sequenciais para construção das árvores de cortes são bem conhecidos, porém nenhum experimento com implementações paralelas deles foi publicado. Apresentamos nesse artigo os resultados experimentais de duas implementações do algoritmo de Gusfield utilizando OpenMP e MPI, respectivamente. Os resultados mostram que as implementações paralelas alcançam bons *speedups*. As heurísticas propostas reduzem o número de falhas. A solução paralela é relativamente simples e requer poucas mudanças no código sequencial, particularmente no caso da implementação baseada em OpenMP. Enquanto a implementação com OpenMP permite um maior controle sobre os elementos de processamento (*threads*), a versão com MPI provê melhor escalabilidade. As duas implementações são complementares pois podem tirar proveito de computadores com múltiplos processadores bem como de *clusters* computacionais.

Trabalhos futuros incluem novos experimentos com as heurísticas para melhorar a escalabilidade das implementações e uma implementação paralela do algoritmo de Gomory-Hu.

Tabela 5. Speedups das heurísticas da implementação em MPI executada em 15 máquinas (16 processos).

Instância	sem heurística	heurística 1	heurísticas 1 & 2
powergrid	2.1	4.0	6.1
CA-HepPh	9.9	10.5	12.1
geocomp2	3.2	4.3	5.6
p2p-Gnutella04	9.3	9.7	11.2
dblcyc.1024	9.9	12.0	12.8
er.10000.5.1	9.6	9.8	9.8
noi5.1500.38	5.8	5.9	8.5
path.2000.101	6.8	7.1	7.9
tree.1500.100	7.0	7.1	7.1
ba.10000.5.1	15.2	15.5	15.3

Agradecimentos

Esse trabalho fez uso do *cluster* computacional do LCPAD-UFPR que tem financiamento do FINEP através do projeto CT-INFRA/UFPR. Jaime Cohen está de licença da UEPG para concluir o doutorado na UFPR e é bolsista da Fundação Araucária/SETI (Edital No. 16/2008). Luiz A. Rodrigues é aluno de doutorado da UFPR e bolsista da Fundação Araucária/SETI (Núm. 19836). Este projeto foi parcialmente apoiado pelo projeto núm. 304013/2009-9 do CNPq.

Referências

[1] P. S. Pacheco. *A User's Guide to MPI*. University of San Francisco, 1998.

[2] B. Chapman, G. Jost, and R. Van der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT Press, 2008.

[3] G. Rao, H. Stone, and T. Hu. Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers*, 28:291–299, 1979.

[4] C. Kim. Cut-tree construction for facility layout. *Computers & Industrial Engineering*, 28(4):721–730, 1995.

[5] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 181–190, New York, NY, USA, 2007. ACM.

[6] B. Saha and P. Mitra. Dynamic algorithm for graph clustering using minimum cut tree. In *Proceedings of the Sixth IEEE International Conference on Data Mining - Workshops*, pages 667–671, Washington, DC, USA, 2006. IEEE Computer Society.

Tabela 4. Número de laços falhos com e sem as heurísticas da implementação em MPI executada em 15 máquinas e da implementação com OpenMP executada com 8 threads.

	MPI		OpenMP	
Instância	sem heurística	com heurísticas	sem heurística	com heurísticas
powergrid	3887.0	306.4	3296.3	3.3
CA-HepPh	2907.2	29.0	3909.1	32.7
geocomp2	1997.6	44.8	1995.0	14.9
p2p-Gnutella04	1279.2	0.0	1277.8	3.6
dblcyc.1024	317.4	14.0	160.8	15.0
er.10000.5.1	182.4	0.0	114.3	0.0
noi5.1500.38	170.2	38.8	88.3	12.6
path.2000.101	112.2	42.4	108.4	26.2
tree.1500.100	1.2	0.6	0.6	2.3
ba.10000.5.1	0.0	0.0	0.0	0.0

[7] A. Mitrofanova, M. Farach-Colton, and B. Mishra. Efficient and robust prediction algorithms for protein complexes using gomory-hu trees. In R. B. Altman, A. K. Dunker, L. Hunter, T. Murray, and T. E. Klein, editors, *Pacific Symposium on Biocomputing*, pages 215–226, 2009.

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

[9] L. Wu and P. K. Varshney. On survivability measures for military networks. *Military Communications Conference*, pages 1120–1124, 1990.

[10] C. L. M. M. Grötschel and M. Stoer. Polyhedral and computational investigations for designing communication networks with high survivability requirements. *Operations Research*, 1995.

[11] H. Saran and V. V. Vazirani. Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108, 1995.

[12] G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4), 2003.

[13] A. Letchford, G. Reinelt, and D. Theis. Odd minimum cut-sets and b-matchings revisited. *SIAM Journal on Discrete Mathematics*, 22(4), 2008.

[14] R. Görke, T. Hartmann, and D. Wagner. Dynamic graph clustering using Minimum-Cut trees. In F. Dehne, M. Gavrilova, J.-R. Sack, and C. Tóth, editors, *Algorithms and Data Structures*, volume 5664 of *Lecture Notes in Computer Science*, chapter 30, pages 339–350. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009.

[15] K. Y. Kamath and J. Caverlee. Transient crowd discovery on the real-time social web. In *Proceedings of the 4th ACM Web Search and Data Mining Conference (WSDM)*, 2011.

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

[17] H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, New York, NY, USA, 2008.

[18] D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In M. J. Oudshoorn and S. Rajasekaran, editors, *ISCA PDCS*, pages 41–48. ISCA, 2005.

[19] B. Hong and Z. He. An asynchronous multi-threaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[20] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35:921–940, 1988.

[21] B. V. Cherkassky and A. V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.

[22] J. K. J. Leskovec and C. Faloutsos. Graph evolution: Densification and shrinking diameters. In *ACM Transactions on Knowledge Discovery from Data (ACM TKDD)*, 2007.

[23] V. Batagelj and A. Mrvar. Pajek datasets. *Disponível em http://vlado.fmf.uni-lj.si/pub/networks/data/*, Acessado em abril de 2011.

[24] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[25] B. Bollobás. *Random Graphs*. Cambridge University Press, 2 edition, 2001.

[26] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(1):47–97, 2002.

[27] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 324–333, PA, USA, 1997. SIAM.

[28] A. V. Goldberg and K. Tsioutsiouliklis. Cut tree algorithms. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 376–385, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.