Algoritmo Paralelo para Árvore Geradora usando GPU

Jucele F. A. Vasconcellos[1], **Edson N. Cáceres**[1], **Henrique Mongelli**[1], **Siang W. Song**[2]

[1]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul (UFMS)
Campo Grande – MS – Brazil

[2]Instituto de Matemática e Estatística – Universidade de São Paulo (USP)
São Paulo – SP – Brazil

{jucele,edson,mongelli}@facom.ufms.br, song@ime.usp.br

Resumo. *Neste trabalho, usando o modelo BSP/CGM, propomos um algoritmo paralelo, com uma implementação em CUDA, para obter uma árvore geradora de um grafo. Trabalhos anteriores para este problema são baseados na solução do problema de list ranking que, embora eficiente na teoria, não produz bons ganhos na prática. Num trabalho posterior, baseado na ideia do cálculo de uma estrutura chamada esteio, Cáceres et al. propuseram um algoritmo paralelo no modelo BSP/CGM para obter uma árvore geradora sem a utilização de list ranking. O cálculo do esteio é obtido com a utilização de um grafo bipartido auxiliar, com o uso de ordenação inteira. Neste artigo melhoramos aquele trabalho em vários aspectos. No algoritmo proposto, para implementação em GPGPU, não é mais necessário calcular o grafo bipartido, e a construção do esteio não necessita do algoritmo de ordenação. A eficiência e escalabilidade do algoritmo proposto são verificadas por experimentos.*

1. Introdução

A computação da árvore geradora de um grafo é um dos principais problemas da área de Teoria dos Grafos, e com grande quantidade de aplicações na área de Computação. Vários algoritmos usam a computação de uma árvore geradora como um procedimento intermediário, o que motiva a busca por algoritmos eficientes para a determinação da árvore geradora de um dado grafo. Existem algoritmos sequenciais ótimos, baseados em busca em profundidade e em busca em largura [Karger et al. 1995], para computar a árvore geradora de um dado grafo. Considerando que os grafos de vários problemas reais têm um tamanho muito grande, apesar do fato de os algoritmos sequenciais para a computação da árvore geradora terem complexidade linear com relação a entrada, torna-se imperioso a busca por algoritmos paralelos eficientes para esse problema.

Um dos desafios para a obtenção de um algoritmo paralelo eficiente é o fato que os algoritmos de busca em largura e busca em profundidade não possuem uma versão paralela equivalente a sequencial, o que dificulta a sua utilização. Os principais algoritmos paralelos para esse problema são baseados na solução proposta por Hirschberg et al. [Hirschberg et al. 1979], onde os vértices do grafo são sucessivamente combinados em super vértices maiores, ou no algoritmo de Borůvka [Borůvka 1926]. Usando essas abordagens, vários algoritmos paralelos para o problema da árvore geradora foram propostos [Chin et al. 1982, Johnson and Metaxas 1995].

No início dos anos 2000, o acesso mais fácil a máquinas paralelas de memória compartilhada (Beowulf´s) possibilitou que os algoritmos paralelos teóricos fossem im-

plementados. Esses algoritmos quando implementados em máquinas paralelas reais não obtiveram bons *ganhos* ou *speed-ups*. Nesse período um esforço considerável foi dispendido na obtenção de algoritmos paralelos eficientes, não só do ponto de vista teórico, mas que também obtivessem bons *speed-ups*. Utilizando modelos de computação realista BSP/CGM [Valiant 1990, Dehne et al. 1996], Dehne et al. propuseram um algoritmo BSP/CGM para computar uma árvore geradora e os componentes conexos de um grafo [Dehne et al. 2002], que utiliza $O(\log p)$ rodadas de comunicação, onde p é o número de processadores. Esse algoritmo, como os baseados nos algoritmos de Hirschberg et al. e no de Borůvka , utilizam o algoritmo de *list ranking* [Dehne and Song 1997] em vários de seus passos. Esse fato acaba por impactar o *speed-up* desses algoritmos.

Baseado na ideia do cálculo de um esteio [Cáceres et al. 1993], Cáceres et al. propuseram uma nova abordagem para o cálculo de uma árvore geradora sem a utilização de *list ranking* [Cáceres et al. 2004]. O algoritmo utiliza $O(\log p)$ rodadas de comunicação, mas tem a vantagem prática de evitar a computação do *list ranking*. O cálculo do esteio é obtido com a utilização de um grafo bipartido auxiliar, onde as arestas do esteio são selecionadas com a utilização de um algoritmo de ordenação inteira, que pode ser implementada de forma eficiente no modelo BSP/CGM [Chan and Dehne 1999]. Um dos limitantes para o *speed-up* desse algoritmo é a necessidade de computar um grafo bipartido do grafo de entrada. O grafo bipartido é obtido com a inserção de um vértice em cada uma das arestas do grafo original.

Com o aumento da capacidade das placas gráficas *General Purpose Graphics Processing Units* (GPGPU´s), surge a oportunidade de analisar o comportamento dos algoritmos BSP/CGM nessa nova arquitetura. Um dos principais limitantes dos *speed-ups* dos algoritmos BSP/CGM, quando implementados em máquinas de memória distribuída é o tempo gasto com a comunicação. Para problemas irregulares como o da computação da árvore geradora de um grafo, são necessárias $O(\log p)$ rodadas de comunicação para a obtenção da árvore geradora do grafo de entrada.

A abordagem proposta por Lima et al. [Lima et al. 2016] estabelece que os superpassos do modelo BSP/CGM sejam representados pelas invocações sequenciais cada *kernel* do CUDA. A execução paralela de cada *kernel* pelas diversas *threads* criadas pelo CUDA representa uma rodada de computação, que pode ser intercalada por comunicação entre as *threads* através da memória da GPU e comunicação entre a GPU e a CPU. Considerando esta abordagem, podemos prever que o comportamento teórico do nosso algoritmo paralelo terá um desempenho compatível quando implementado numa GPGPU. No caso do problema de árvore geradora mínima, existem diversos trabalhos que propõem soluções paralelas usando GPGPU, como os apresentados por [Vineet et al. 2009, Nobari et al. 2012, Li and Becchi 2013, Nasre et al. 2013].

Neste trabalho, utilizando o modelo BSP/CGM, propomos um algoritmo BSP/CGM para a computação da árvore geradora de um dado grafo. O algoritmo proposto é eficiente no modelo BSP/CGM e utiliza $O(\log p)$ rodadas de computação. O algoritmo é baseado no algoritmo proposto por Cáceres et al. [Cáceres et al. 2004] e no algoritmo de Borůvka [Graham and Hell 1985]. Nosso algoritmo não precisa calcular o grafo bipartido auxiliar, e a construção do esteio não necessita de algoritmos de ordenação. Esses dois passos utilizam um tempo considerável na execução total do algoritmo. Para demonstrar que o algoritmo também tem um bom desempenho na prática, a implementação foi

executada com a GPGPU Nvidia Quadro M4000 (que possui 1.664 núcleos e oito GB de memória). Os resultados de *speed-up* obtidos pelo algoritmo são competitivos, demostrando que o modelo BSP/CGM é adequado para o projeto de algoritmos paralelos realistas.

2. Algoritmo paralelo para o problema da árvore geradora

Nosso algoritmo paralelo foi projetado usando o modelo BSP/CGM [Valiant 1990, Dehne et al. 1996]. Resumidamente, esse modelo consiste de um conjunto de p processadores, cada um tendo uma memória local de tamanho $O(n/p)$.

Um algoritmo nesse modelo executa um conjunto de rodadas (superpassos) de computação local alternadas com fases de comunicação global, separadas por uma barreira de sincronização. O custo da comunicação considera o número de rodadas necessárias para a execução do algoritmo.

O modelo BSP/CGM é adequado para o projeto e análise de algoritmos paralelos onde há muita comunicação entre os processos. Essa é uma característica de problemas irregulares, ou seja a entrada em cada rodada do programa muda e os processadores necessitam das informações que foram computadas nos diversos processadores para a próxima rodada. O problema da árvore geradora se enquadra nessa classe, o que motiva a utilização do modelo para prever o comportamento e a complexidade do algoritmo.

Como estamos interessados em analisar não só os aspectos teóricos do nosso algoritmo, temos que mapear os passos do algoritmo BSP/CGM na arquitetura da GPGPU. As rodadas (superpassos) do modelo BSP/CGM são representados pelas chamadas de cada *kernel* do CUDA. Além disso, associamos o conjunto de processadores (p) do modelo BSP/CGM ao conjunto de *streaming multiprocessors* (SM´s) da GPGPU.

Vamos agora descrever os conceitos básicos necessários que serão utilizados em nosso algoritmo. Seja $G = (V, E)$ um **grafo**, onde $V = \{v_1, v_2, \ldots, v_n\}$ é um conjunto de n **vértices** e E é um conjunto de m **arestas** (v_i, v_j), sendo v_i e v_j vértices de V. Um **caminho** em G é uma sequência de arestas $(v_1, v_2), (v_2, v_3), (v_3, v_4) \ldots, (v_{n-1}, v_n)$ conectando vértices distintos $v_1, \ldots, v_n$ de G. Um **ciclo** é um caminho conectando vértices distintos $v_1, v_2, \ldots, v_k$ tal que $v_1 = v_k$. Um grafo é considerado **conexo** se existe um caminho para todo par de vértices v_i, v_j, $1 \leq i \neq j \leq n$ em V. Uma **árvore** $T = (V, E)$ é um grafo conexo sem ciclos. Uma **floresta** é um conjunto de árvores. Uma **árvore geradora** de $G = (V, E)$ é uma árvore $T = (V, E')$ que inclui todos os vértices de G e é um subgrafo de G, ou seja todas as arestas de T pertencem a G, $E' \subset E$. Uma árvore geradora $T = (V, E')$ de $G = (V, E)$ também pode ser definida como o conjunto maximal de arestas de G, $E' \subset E$ e $|E'| = |V| - 1$, que não contém nenhum ciclo.

No algoritmo é utilizado o conceito denominado esteio, mas diferente da definição apresentada em [Cáceres et al. 1993]. No contexto deste artigo, um **esteio**, representado por S, é definido como uma floresta geradora de G, tal que cada vértice $v_i \in V$ é incidente em S com pelo menos uma aresta (v_i, v_j) tal que v_j é o menor vértice ligado a v_i. Uma aresta (v_i, v_j) do esteio é considerada uma **aresta zero-diferença** se ela for escolhida para os dois vértices, tanto para v_i quanto para v_j. A Figura 1 apresenta um grafo G com cinco vértices e oito arestas. O esteio para este grafo é composto por quatro arestas, onde uma delas é zero-diferença. No caso desse exemplo o esteio gerado na primeira iteração do algoritmo é uma árvore geradora de G.

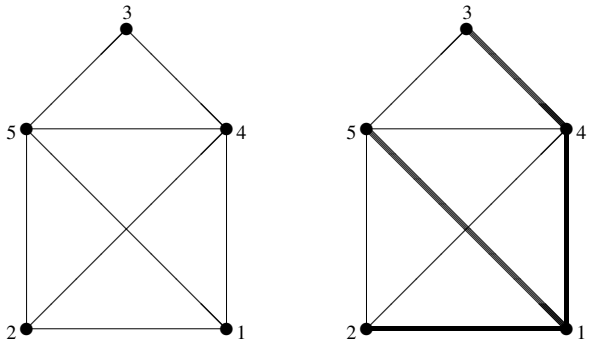

Figura 1. À esquerda é apresentado o grafo $G = (V, E)$, sendo $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente de G, formado pela aresta $(1, 2)$ (escolhida pelos vértices 1 e 2), aresta $(1, 4)$ (escolhida pelo vértice 4), aresta $(1, 5)$ (escolhida pelo vértice 5) e aresta $(3, 4)$ (escolhida pelo vértice 3), sendo $(1, 2)$ a única aresta zero-diferença.

Algoritmo 1 apresenta a ideia de funcionamento da solução proposta. Este consiste basicamente em encontrar o esteio do grafo e se necessário, caso a árvore geradora não esteja completa, compactá-lo para nova iteracão do algoritmo. A solução consiste de um algoritmo paralelo, ou seja, os passos são executados por diversos processadores encontrando a solução de forma colaborativa.

Algoritmo 1: Algoritmo para árvore geradora

Entrada: Um grafo conexo $G = (V, E)$
Saída: Uma árvore geradora de G cujas arestas estão em $Solucao$.

```
1  início
2      Solucao := vazia
3      repita
4          escolher as arestas do esteio
5          adicionar as arestas do esteio ao conjunto Solucao
6          verificar o número de arestas zero-diferença
7          se número de arestas zero-diferença != 1 então
8              compactar o grafo
9          fim
10     até número de arestas zero-diferença = 1;
11 fim
```

O número de iterações do algoritmo e a árvore geradora produzida depende da rotulação dos vértices. A Figura 2 apresenta o mesmo grafo da Figura 1 mas a rotulação dos vértices é diferente. Neste exemplo, o esteio gerado na primeira iteração do algoritmo, formado pelas arestas $(1, 4)$, $(1, 5)$ e $(2, 3)$, apresenta duas arestas zero-diferença $(1, 4)$ e $(2, 3)$, implicando em outra iteração do algoritmo.

Para o exemplo da Figura 2 é necessário fazer a compactação do grafo G, visto que o número de arestas zero-diferença é diferente de um. Essa compactação inicia com o cálculo das componentes conexas a partir das arestas do esteio e a eliminação das arestas que interligam vértices de uma mesma componente. O grafo resultante da compactação terá dois vértices (1 e 2) e quatro arestas.

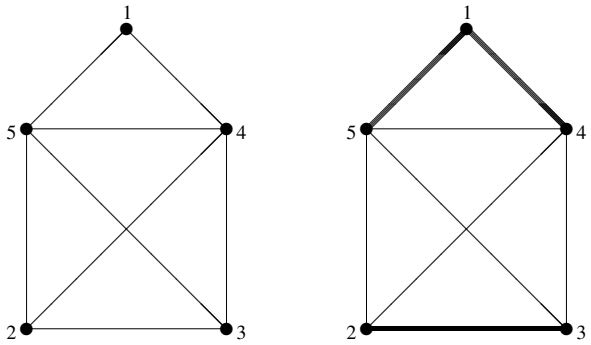

Figura 2. À esquerda é apresentado o grafo $G = (V, E)$, sendo $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente de G, formado pela aresta $(1, 4)$ (escolhida pelos vértices 1 e 4), aresta $(1, 5)$ (escolhida pelo vértice 5) e aresta $(2, 3)$ (escolhida pelos vértices 2 e 3), sendo $(1, 4)$ e $(2, 3)$ arestas zero-diferença.

3. Correção, Complexidade e Detalhes da Implementação

Nesta seção, abordamos a correção do algoritmo proposto.

Lema 1. *Considere um grafo conexo $G = (V, E)$. Seja G' o grafo obtido pela adição das arestas escolhidas para compor o esteio S (passo 5 do Algoritmo 1). Então G' é acíclico. Mais ainda, se S contém exatamente uma aresta zero-diferença então G' é uma árvore geradora de G.*

Demonstração. Considerando a forma como as arestas são selecionadas para construir o esteio S, onde para cada vértice v_i, é selecionada a aresta com o menor v_j, o conjunto de arestas selecionadas (v_i, v_j), tal que $v_i > v_j$ não formam um ciclo. Se a aresta (v_k, v_l) tal que $v_k < v_l$, ou a aresta (v_l, v_k) foi selecionada ou todas arestas (v_k, v_s) adjacentes a v_k tem que $v_s > v_l$, como $v_k < v_l$, temos que as arestas adjacentes a v_s só se conectarão a v_l usando a aresta v_k, não formando um ciclo. Se apenas uma das arestas for selecionada duas vezes, temos que o esteio S é uma árvore geradora de G. □

Teorema 1. *A cada iteração o número de arestas zero-diferença é no mínimo divido por 2.*

Demonstração. Considerando que cada vértice v_i seleciona uma aresta adjacente (v_i, v_j), onde v_j é o menor vértice adjacente a v_i, o número de arestas selecionadas duas vezes é no máximo $\lceil |V| \rceil / 2$. □

Teorema 2. *O algoritmo para a computação da árvore geradora utiliza $\log p$ rodadas de comunicação com computação $O(n/p)$ computação local.*

Demonstração. Pelo teorema anterior, temos que o grafo compactado após a execução do algoritmo tem no máximo $\lceil |V| \rceil / 2$ vértices, assim, após a $\log p$ rodadas, o grafo compactado terá no máximo 1 aresta zero-diferença e a árvore geradora será obtida. A computação local dos passos de cada rodada pode ser feita em tempo $O(n/p)$. □

Considerando que os trabalhos mais recentes sobre árvores geradoras tais como [Vineet et al. 2009, Nobari et al. 2012, Li and Becchi 2013, Nasre et al. 2013], resolvem problemas mais gerais, além de usarem em sua implementação recursos computacionais muito diferentes, nosso objetivo nesse trabalho foi o de verificar se o algoritmo

proposto também tinha um bom desempenho em máquinas paralelas reais de memória compartilhada, e qual o ganho obtido pelo algoritmo com relação a sua versão sequencial. Para efetuar essa comparação, uma versão sequencial do algoritmo foi desenvolvida usando ANSI C. A versão paralela foi implementada para GPGPU usando CUDA (*Compute Unified Device Architecture*). Ambas as implementações estão disponíveis para download em https://github.com/jucele/ArvoreGeradora.

A implementação CUDA implementa os passos do algoritmo utilizando nove funções do tipo *kernel*. Logo após a leitura dos dados do grafo de entrada estes dados são copiados para a memória global da GPGPU. Duas funções do tipo *kernel* são utilizadas para escolher a aresta do esteio para cada um dos vértices. Após a seleção de uma aresta, utilizamos uma função para marcá-la como uma aresta do esteio. Também usamos uma função para calcular o número de arestas zero-diferença, critério de parada do algoritmo, e copiar as arestas do esteio para o vetor Solução. Outras cinco funções são utilizadas para a compactação do grafo, incluindo o cálculo dos componentes conexos, onde empregamos a proposta apresentada em [Hawick et al. 2010]. Também usamos funções atômicas disponíveis na biblioteca CUDA na implementação de alguns *kernels*.

A implementação do algoritmo não utilizou nenhuma técnica especial de programação em CUDA, pois o objetivo principal do nosso trabalho foi o de apresentar um algoritmo eficiente no modelo BSP/CGM que pudesse ser facilmente implementado numa máquina paralela real.

4. Resultados Experimentais

Como descrevemos anteriormente, o principal objetivo da implementação é o de demonstrar a funcionalidade do algoritmo proposto numa máquina paralela real.

Para isso usamos uma estação de trabalho Intel® Xeon® E5-1620 v3, 3.50GHz, com 8 cores, 10 MB de cache, 32 GB de memória e uma GPGPU Nvidia Quadro M4000 (com 1.664 núcleos e oito GB de memória). Para analisar a eficiência e escalabilidade do algoritmo utilizamos como entrada dois conjuntos de grafos. O primeiro conjunto composto de grafos construídos artificialmente por meio de um gerador de grafos aleatórios. E o segundo composto pelos grafos das redes rodoviárias dos Estados Unidos, disponibilizados no Nono Desafio de Implementação DIMACS.

Para o primeiro conjunto de grafos de entrada utilizamos o gerador aleatório [Johnsonbaugh and Kalin 1991] disponível em http://condor.depaul.edu/rjohnson/source/graph ge.c. Esse gerador possibilitou gerar grafos conexos com um conjunto bem variado de vértices e arestas. Foram gerados 28 grafos conexos com 10.000, 15.000, 20.000, 25.000 e 30.000 vértices. A Tabela 1 apresenta as principais características dos grafos, onde n é o número de vértices e m o número de arestas.

O Nono Desafio de Implementação DIMACS, apresentado no sítio http://www.dis.uniroma1.it/challenge9/, disponibiliza doze grafos de redes rodoviárias dos Estados Unidos. Visto que nossa implementação trabalha com grafos não dirigidos, e como os arquivos dos grafos disponibilizados apresentam as arestas duplicadas (uma para representar o arco entre o vértice a e b e outra para representar a ligação entre b e a), reduzimos o número de arestas dos grafos pela metade. A Tabela 2 mostra as informações desse conjunto de grafos.

Tabela 1. Características básicas dos grafos de entrada gerados.

Grafo de entrada	n	m	densidade	m/n
graph10a	10.000	1.000.000	0,020	100
graph10b	10.000	2.500.000	0,050	250
graph10c	10.000	5.000.000	0,100	500
graph10d	10.000	7.500.000	0,150	750
graph10e	10.000	10.000.000	0,200	1.000
graph15a	15.000	2.500.000	0,022	166,7
graph15b	15.000	5.500.000	0,049	366,7
graph15c	15.000	11.500.000	0,102	766,7
graph15d	15.000	17.000.000	0,151	1.133,3
graph15e	15.000	22.500.000	0,200	1.500
graph15f	15.000	56.300.000	0,500	3.753,3
graph15g	15.000	84.350.000	0,750	5.623,3
graph15h	15.000	112.492.500	1,000	7.499,5
graph20a	20.000	4.000.000	0,020	200
graph20b	20.000	10.000.000	0,050	500
graph20c	20.000	20.000.000	0,100	1.000
graph20d	20.000	30.000.000	0,150	1.500
graph20e	20.000	40.000.000	0,200	2.000
graph25a	25.000	6.200.000	0,020	248
graph25b	25.000	15.500.000	0,050	620
graph25c	25.000	32.000.000	0,100	1.280
graph25d	25.000	47.000.000	0,150	1.880
graph25e	25.000	62.500.000	0,200	2.500
graph30a	30.000	9.000.000	0,020	300
graph30b	30.000	22.500.000	0,050	750
graph30c	30.000	45.000.000	0,100	1.500
graph30d	30.000	67.500.000	0,150	2.250
graph30e	30.000	90.000.000	0,200	3.000

Tabela 2. Características básicas dos grafos do nono desafio DIMACS, considerando os grafos não dirigidos e eliminando as arestas duplicadas.

Grafo de entrada	n	m	densidade	m/n
USA-road-d.NY	264.346	366.648	0,0000105	1,4
USA-road-d.BAY	321.270	399.652	0,0000077	1,2
USA-road-d.COL	435.666	527.767	0,0000056	1,2
USA-road-d.FLA	1.070.376	1.354.681	0,0000024	1,3
USA-road-d.NW	1.207.945	1.417.704	0,0000019	1,2
USA-road-d.NE	1.524.453	1.946.326	0,0000017	1,3
USA-road-d.CAL	1.890.815	2.325.452	0,0000013	1,2
USA-road-d.LKS	2.758.119	3.438.289	0,0000009	1,2
USA-road-d.E	3.598.623	4.382.787	0,0000007	1,2
USA-road-d.W	6.262.104	7.609.574	0,0000004	1,2
USA-road-d.CTR	14.081.816	17.120.937	0,0000002	1,2
USA-road-d.USA	23.947.347	29.120.580	0,0000001	1,2

Para cada um dos grafos de entrada, as implementações sequencial e CUDA foram executadas 20 vezes, sendo usada a média do tempo de execução para analisar o comportamento experimental do algoritmo. A Tabela 3 apresenta os resultados obtidos dos testes para o primeiro conjuntos de grafos de entrada (Tabela 1), onde cada linha

mostra o número de iterações do algoritmo, o tempo de execução da implementação sequencial e o tempo de execução da implementação CUDA e o *speed-up* da implementação em CUDA relativa a implementação sequencial. Vale salientar que o número de iterações do algoritmo necessárias para encontrar a árvore geradora para esse conjunto de testes não passou de dois.

Tabela 3. Resultados dos testes para os grafos gerados artificialmente.

Grafo de entrada	Número de Iterações	Tempo Sequencial (s)	Tempo CUDA (s)	speed-up
graph10a	2	0,031	0,005	6,862
graph10b	2	0,082	0,025	3,286
graph10c	2	0,165	0,018	9,309
graph10d	2	0,259	0,025	10,294
graph10e	2	0,361	0,032	11,389
graph15a	2	0,077	0,009	8,200
graph15b	2	0,170	0,020	8,729
graph15c	2	0,395	0,038	10,500
graph15d	2	0,551	0,052	10,653
graph15e	1	0,278	0,051	5,445
graph15f	1	0,693	0,096	7,183
graph15g	1	1,694	0,132	12,858
graph15h	1	2,858	0,167	17,075
graph20a	2	0,120	0,014	8,598
graph20b	2	0,303	0,034	9,013
graph20c	2	0,668	0,062	10,824
graph20d	2	0,977	0,085	11,490
graph20e	2	1,445	0,106	13,653
graph25a	2	0,186	0,021	8,702
graph25b	2	0,503	0,051	9,870
graph25c	2	1,001	0,094	10,688
graph25d	1	0,584	0,100	5,859
graph25e	2	1,962	0,156	12,613
graph30a	2	0,273	0,031	8,892
graph30b	2	0,695	0,072	9,606
graph30c	2	1,535	0,127	12,053
graph30d	2	2,363	0,173	13,653
graph30e	2	3,852	0,214	18,013

A Tabela 4 mostra os resultados dos testes para o conjunto de grafos do nono desafio DIMACS (Tabela 2). Para essas entradas o *speed-up* da implementação em CUDA em relação à sequencial variou de 3,248 a 19,689. Como o número de vértices desse conjunto de grafos de entrada é bem maior do que o anterior são necessárias de 7 a 10 iterações do algoritmo para que a árvore geradora seja encontrada.

O gráfico da Figura 3 ilustra como o tempo da implementação sequencial é pior do que o tempo da implementação paralela utizando CUDA. A Figura 4 mostra o crescente *speed-up* da implementação CUDA a medida que o tamanho da entrada aumenta em termos de arestas para grafos com 30.000 vértices.

Os resultados mostram a funcionalidade do algoritmo numa máquina paralela real. Os *speed-ups* obtidos também mostram a eficiência do algoritmo, uma vez que os tempos

Tabela 4. Resultados dos testes para os grafos do conjunto nono desafio DIMACS.

Grafo de entrada	Número de Iterações	Tempo Sequencial (s)	Tempo CUDA (s)	speed-up
USA-road-d.NY	7	0,056	0,017	3,248
USA-road-d.BAY	7	0,064	0,018	3,464
USA-road-d.COL	8	0,088	0,020	4,425
USA-road-d.FLA	9	0,263	0,052	5,083
USA-road-d.NW	8	0,283	0,051	5,590
USA-road-d.NE	9	0,475	0,060	7,903
USA-road-d.CAL	9	0,560	0,061	9,245
USA-road-d.LKS	9	0,880	0,084	10,426
USA-road-d.E	10	1,356	0,127	10,665
USA-road-d.W	10	2,299	0,214	10,746
USA-road-d.CTR	10	13,277	0,674	19,689
USA-road-d.USA	10	9,302	0,776	11,985

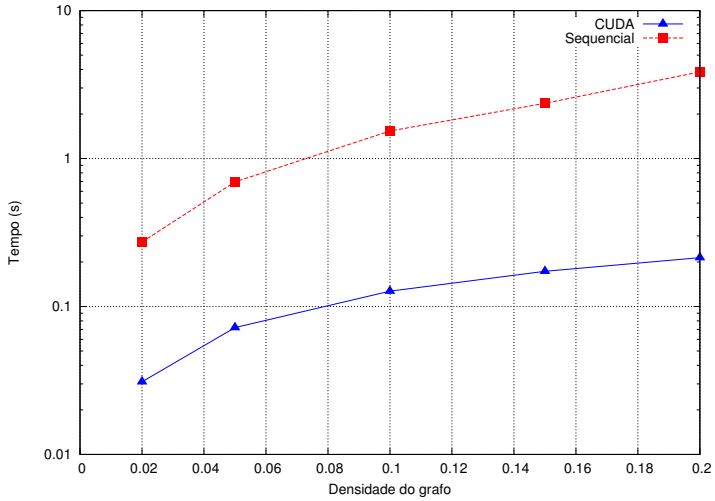

Figura 3. Tempo de Execução para grafos com 30.000 vértices.

das execuções paralelas são bem melhores que das sequenciais. Relembramos que o problema tratado tem uma solução sequencial ótima que é linear para o tamanho da entrada, e que essa a solução não possui uma implementação paralela direta.

Uma das dificuldades da implementação desse algoritmo numa arquitetura de memória distribuída é a quantidade de mensagens trocadas entre os processadores durante a fase de computação, pois em várias computações, há a necessidade de percorrer a lista das arestas e essa lista está distribuída entre os diversos processadores. Na arquitetura de memória compartilhada esse problema é minimizado, além do que existem melhorias implementadas com relação ao algoritmo anterior [Cáceres et al. 2004], não sendo mais necessárias a construção do grafo bipartido auxiliar nem a ordenação das arestas.

Como destacamos anteriormente, a comparação dos tempos de execução com algoritmos para árvore geradora utilizando CUDA que foram publicados recentemente não foi possível, pois os algoritmos são para problemas mais específicos e utilizam recursos computacionais bem diferentes.

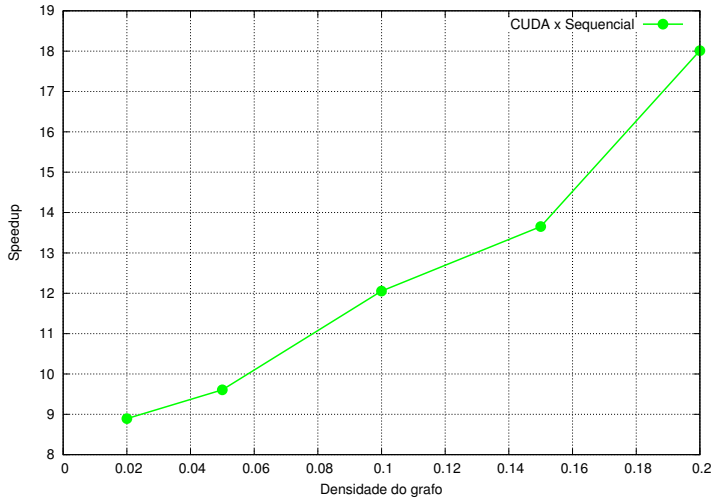

Figura 4. Speedups obtidos pela implementação CUDA em relação a implementação sequencial para grafos com 30.000 vértices.

5. Conclusões e trabalhos futuros

Neste trabalho, utilizando o modelo BSP/CGM, apresentamos um algoritmo paralelo para o cálculo de árvore geradora. O modelo BSP/CGM tem se mostrado bem adequado para o projeto de algoritmo paralelos, principalmente os que utilizam muita comunicação entre os processadores. Além disso, os algoritmos projetados nesse modelo têm obtido bons *speed-ups* quando implementado em máquinas reais.

Considerando que os principais algoritmos sequenciais para o problema da árvore geradora utilizam estratégias de busca em profundidade ou busca em largura em um grafo, e que essas estratégias não possuem uma paralelização eficiente, os algoritmos paralelos propostos para esse problema usam outras abordagens. Várias das soluções fazem uso do algoritmo de *list ranking*, o que prejudica o desempenho final dos algoritmos.

O algoritmo proposto é baseado no trabalho de Cáceres et al. [Cáceres et al. 2004], que não utiliza *list ranking* e utiliza um grafo bipartido auxiliar e ordenações do conjunto de arestas. O algoritmo proposto neste trabalho não faz uso de um grafo bipartido, nem necessita ordenar o conjunto de arestas, e utiliza $\log p$ rodadas de comunicação com $O(n/p)$ computação local em cada unidade de processamento.

Para demonstrar a eficiência do algoritmo proposto, uma implementação CUDA foi desenvolvida e testada em uma GPGPU. Utilizamos um gerador de grafos para analisar a escalabilidade da implementação. Também realizamos testes utilizando como entrada os grafos disponibilizados pelo novo desafio DIMACS. Os tempos e *speed-ups* obtidos são competitivos, e mostram que o modelo BSP/CGM é apropriado para o projeto e desenvolvimento de algoritmos paralelos para máquinas paralelas reais.

Como trabalhos futuros, pretendemos desenvolver uma abordagem de poda para reduzir o conjunto de arestas do grafo de entrada, que pode melhorar os resultados do algoritmo e permitir trabalhar com grafos de entrada maiores. Pretendemos avaliar também o algoritmo com outros conjuntos de dados e analisar o seu desempenho em equipamentos computacionais com maior poder de processamento, tais como múltiplas GPUs.

Agradecimentos

Esta pesquisa foi parcialmente financiada pelo CNPq Proc. No. 482736/2012-7, 30.2620/2014-1 e 465446/2014-0, e FAPESP Proc.2014/50937-1.

Referências

Borůvka, O. (1926). On a minimal problem. *Prace Moravské Pridovedecké Spodecnosti*, 3:37–58.

Cáceres, E. N., Dehne, F., Mongelli, H., Song, S. W., and Szwarcfiter, J. (2004). A coarse-grained parallel algorithm for spanning tree and connected components. In *Euro-Par 2004. Lecture Notes in Computer Science*, volume 3149, p. 828–831. Springer-Verlag.

Cáceres, E. N., Deo, N., Sastry, S., and Szwarcfiter, J. L. (1993). On finding Euler tours in parallel. *Parallel Processing Letters*, 3(3):223–231.

Chan, A. and Dehne, F. (1999). A note on coarse grained parallel integer sorting. *Parallel Processing Letters*, 9(4):533–538.

Chin, F. Y., Lam, J., and Chen, I.-N. (1982). Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665.

Dehne, F., Fabri, A., and Rau-Chaplin, A. (1996). Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. *International Journal on Computational Geometry & Applications*, 6(3):298–307.

Dehne, F., Ferreira, A., Cáceres, E., Song, S. W., and Roncato, A. (2002). Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica*, 33(2):183–200.

Dehne, F. and Song, S. W. (1997). Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, 25(1):1–16.

Graham, R. L. and Hell, P. (1985). On the history of of the minimum spanning tree problem. In *Annals of the History of Computing*, volume 7, p. 43–57.

Hawick, K. A., Leist, A., and Playne, D. (2010). Parallel graph component labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655–678.

Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. (1979). Computing connected components on parallel computers. *Comm. ACM*, 22(8):461–464.

Johnson, D. and Metaxas, P. (1995). A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19(3):383 – 401.

Johnsonbaugh, R. and Kalin, M. (1991). A graph generation software package. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education*, volume 23, p. 151–154.

Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328.

Li, D. and Becchi, M. (2013). Deploying graph algorithms on gpus: an adaptive solution. In *Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium*, IPDPS 2013, p. 1013–1024.

Lima, A. C. d., Branco, R. G., Ferraz, S., Cáceres, E. N., Gaioso, R. R. A., Martins, W. S., and Song, S. W. (2016). Solving the maximum subsequence sum and related problems using BSP/CGM model and multi-GPU CUDA. *Journal of The Brazilian Computer Society (Online)*, 22:1–13.

Nasre, R., Burtscher, M., and Pingali, K. (2013). Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, p. 147–156.

Nobari, S., Cao, T.-T., Karras, P., and Bressan, S. (2012). Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, p. 205–214.

Valiant, L. G. (1990). A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111.

Vineet, V., Harish, P., Patidar, S., and Narayanan, P. J. (2009). Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, p. 167–171.