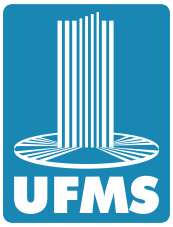
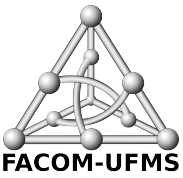


Universidade Federal de Mato Grosso do Sul
Faculdade de Computação



Jucele França de Alencar Vasconcellos

Algoritmos Paralelos para Alinhamento de Sequências e Árvores Geradoras

CAMPO GRANDE - MS
2018

Jucele França de Alencar Vasconcellos

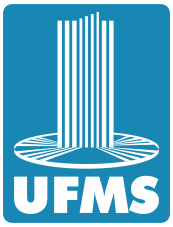
Algoritmos Paralelos para Alinhamento de Sequências e Árvores Geradoras

Tese apresentada a Faculdade de Computação da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para a obtenção do título de Doutora em Ciência da Computação.

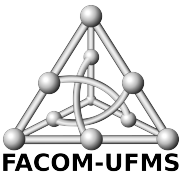
Orientador: Prof. Dr. Edson Norberto Cáceres

Este exemplar corresponde à versão final da Tese defendida por Jucele França de Alencar Vasconcellos e orientada pelo Prof. Dr. Edson Norberto Cáceres.

CAMPO GRANDE - MS
2018



Universidade Federal de Mato Grosso do Sul
Faculdade de Computação



Jucele França de Alencar Vasconcellos

Algoritmos Paralelos para Alinhamento de Sequências e Árvores Geradoras

Banca Examinadora:

- Prof. Dr. Edson Norberto Cáceres
  Faculdade de Computação, UFMS
- Profa. Dra. Lúcia Maria de Assumpção Drummond
  Instituto de Computação, UFF
- Prof. Dr. Wellington Santos Martins
  Instituto de Informática, UFG
- Prof. Dr. Marcelo Henriques de Carvalho
  Faculdade de Computação, UFMS
- Profa. Dra. Edna Ayako Hoshino
  Faculdade de Computação, UFMS

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campo Grande, 19 de outubro de 2018

Dedicatória

A meu querido pai Jurandy (em memória) e minha filha Leda.

As grandes motivações para conseguir finalizar este trabalho.

Depois de terdes sofrido um pouco,
o Deus de toda a graça,
que vos chamou para a sua glória eterna,
no Cristo Jesus, vos restabelecerá
e vos tornará firmes, fortes e seguros.
(1Pd 5,10)

Agradecimentos

Agradeço a meu orientador, Prof. Edson Cáceres, por sua experiente, habilidosa e eficiente orientação, além do incentivo constante e da crença em minha capacidade.

Muito obrigada a meu marido Francisco por sua paciência, carinho, apoio e companheirismo de sempre.

A meus pais pelas orações, confiança e encorajamento.

Aos Profs. Siang, Mongelli e Nalvo e a amiga Christiane pela profícua parceria.

Aos Profs. Nalvo e Mongelli, enquanto diretores da FACOM, pelo apoio institucional recebido.

Aos servidores da Secretaria de Pós-Graduação da FACOM que, no decorrer destes seis anos, ofereceram seu apoio em diversas situações, indo muitas vezes além do esperado para a função.

Aos amigos Anderson, Bianca, Christiane, Rodrigo e Samuel, pela colaboração técnica e amizade.

Ao Prof. Wellington e aos colegas Evandro e Roussian pelo apoio na utilização das máquinas do Instituto de Informática da Universidade Federal de Goiás (INF/UFG).

Ao Laboratório Virtual de Computação de Alto Desempenho da Universidade de Carleton, através do Prof. Frank Dehne, pela disponibilidade de uso do *cluster*.

A Profa. Alba Melo por disponibilizar acesso a seu laboratório na Universidade de Brasília (UnB) e ao colega Daniel Sundfeld pelo apoio durante os testes.

A todos os amigos e colegas da FACOM pela torcida e incentivo.

Aos membros da banca examinadora pela oportunidade de receber seus comentários e críticas e, consequentemente, melhorar meu trabalho.

A minha Mãe do Céu, sob o título de Mãe do Perpétuo Socorro, por sua constante intercessão.

E, o mais importante, obrigada a meu bom Deus. Sem Ele nada seria possível.

Resumo

Os avanços tecnológicos tem gerado grande volume de dados a serem analisados nas mais diversas áreas. Dentre os problemas tratados nesta tese estão a busca da árvore geradora (ST) e da árvore geradora mínima (MST) de um grafo. Estes problemas apresentam um grande conjunto de possíveis aplicações, dentre elas projeto de redes de computadores e de transporte, redes de abastecimento de água, redes de telecomunicações, dentre outros. Existem algoritmos sequências eficientes mas de difícil paralelização pelas características intrínsecas ao método de construção da solução. Durante o desenvolvimento desta pesquisa foram projetados e implementados algoritmos paralelos eficientes para calcular tanto a ST quanto a MST de um grafo. Para o projeto das soluções desenvolvidas foi empregado o modelo BSP/CGM e técnicas de computação de alto desempenho. A melhor solução apresentada alcançou um *speedup* de até 21 quando comparada com outra solução eficiente recentemente publicada. Outro problema também estudado tem relação com a biologia molecular computacional, mais especificamente o alinhamento de sequências. Nesta área, grande parte dos dados a serem explorados são organizados em sequências de caracteres e sua análise busca definir correlações entre sequências, com objetivo de descobrir informações funcionais, estruturais e evolutivas. Existe uma vasta gama de problemas que são resolvidos com algoritmos de manipulação de sequências de caracteres. Um dos resultados alcançados neste trabalho foi o projeto e implementação de soluções eficientes para os problemas de alinhamento de sequências. *multicore* e *manycore*.

Abstract

The technological advances have generated a large volume of data to be analyzed in the most diverse areas. The spanning tree (ST) and the minimum spanning tree (MST) of a graph are two problems addressed in this thesis. These problems present a large set of possible applications, among them the design of computer networks and transport, water supply networks, telecommunications networks, among others. There are some efficient algorithms but they are difficult to parallelize due to the intrinsic characteristics of the solution construction method. During the development of this research, efficient parallel algorithms were designed and implemented to calculate both the ST and the MST of a graph. For the design of the developed solutions, we employed the BSP/CGM model and high-performance computing techniques. Our best solution reached a speedup of up to 21 when compared to another efficient recently published solution. We also studied some problems related to computational molecular biology, more specifically sequence alignment. In this area, much of the data to be explored is organized into sequences of characters and its analysis seeks to define correlations between sequences, to discover functional, structural and evolutionary information. There is a wide range of problems that are solved with algorithms of manipulation of sequences of characters. One of the results achieved in this work was the design and implementation of efficient solutions for sequence alignment problems.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviações e Siglas

9DIMACS	Nono Desafio de Implementação DIMACS
AMS	Alinhamento Múltiplo de Sequências
BSP	*Bulk Synchronous Parallel*
CGM	*Coarse Grained Multicomputer*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DIMACS	*Center for Discrete Mathematics and Theoretical Computer Science*
DNA	*deoxyribonucleic acid ou ácido desoxirribonucleico*
GPGPU	*General-Purpose Computing on Graphics Processing Units*
GPU	*Graphics Processing Unit*
MIMD	*Multiple Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
MPI	*Message Passing Interface*
MST	*Minimum Spanning Tree*
SIMD	*Single Instruction Multiple Data*
SIMD	*Single Program Multiple Data*
ST	*Spanning Tree*

Capítulo 1

Introdução

O cenário tecnológico atual tem trazido um desafio adicional à computação: processar uma quantidade enorme de informações de diversas áreas, captadas das mais diferentes fontes. Porém, a capacidade computacional de um único computador para continuar tratando este crescente número de informações apresenta restrições, apesar da grande evolução das últimas décadas. Assim, a computação de alto desempenho também faz uso de processadores *multicore* e *manycore*, organizados ou não em *cluster*, aliados a técnicas de programação paralela [51, 50, 26], para agilizar o processamento.

A computação paralela permite que vários cálculos sejam realizados ao mesmo tempo, através da divisão de um problema grande em partes menores, que podem ser executadas de forma concorrente (em paralelo). Ao projetar uma solução paralela pode-se optar por diferentes formas de computação paralela. Dentre as mais comuns temos o paralelismo de dados, onde distribui-se o dado por diferentes processadores para serem computados em paralelo (SIMD - *Single Instruction Multiple Data*) e o paralelismo de tarefas em que diferentes cálculos são realizados no mesmo ou em diferentes conjuntos de dados (MISD - *Multiple Instruction Single Data* ou MIMD - *Multiple Instruction Multiple Data*).

Uma importante área da teoria da computação, que engloba diversos problemas com aplicações nas mais variadas áreas, é a Teoria dos Grafos. Tem-se vários problemas com soluções sequenciais eficientes e conhecidas. No entanto, com o aumento das instâncias dos problemas a serem tratados, é necessária a busca por soluções mais rápidas, fazendo uso das tecnologias paralelas para permitir a resoluções de instâncias maiores em menor tempo.

Outra área com bastante campo a ser explorado na otimização de soluções é a biologia molecular, visto que sua demanda por pesquisas e soluções computacionais que auxiliem a análise das informações biológicas aumentou, compondo a área de pesquisa de Bioinformática ou Biologia Computacional. Essa demanda da biologia se intensifica à medida que o desenvolvimento tecnológico atual permite que grande volume de dados biológicos sejam produzidos com maior rapidez

Nos diferentes problemas a serem abordados, a eficiência desejada para a solução paralela a ser construída pode ser alcançada com a proposição de melhorias em abordagens sequenciais ou projeto de novas soluções paralelas, utilizando, por exemplo, os modelos BSP/CGM (*Bulk Synchronous Parallel - Coarse Grained Multicomputer*) [60, 10] e Multi-BSP [61]. Para as soluções teóricas propostas deve-se analisar seu comportamento em

ambientes paralelos reais, a fim de comprovar experimentalmente o desempenho esperado.

Neste contexto, durante o desenvolvimento do presente trabalho foi possível avaliar novas abordagens e aliar o uso de tecnologias de processamento paralelo, desenvolvendo soluções que apresentem melhorias no desempenho de algoritmos aplicados a bioinformática, mais especificamente para os problemas de alinhamento *pairwise* e múltiplo de sequências de DNA, e a computação da árvore geradora e árvore geradora mínima de um grafo.

1.1 Motivação e Breve Descrição dos Problemas

Durante o período de doutorado foi possível trabalhar pesquisando soluções paralelas para dois problemas distintos: computação de árvores geradoras de um grafo e alinhamento de sequências biológicas. O contexto de aplicação das duas soluções envolve o crescimento das entradas e a consequente necessidade de soluções mais eficientes, que possam tirar proveito dos ambientes de computação paralela. As propostas desenvolvidas para esses dois problemas foram realizadas em parceria com outros pesquisadores das áreas de computação paralela, grafos e bioinformática, o que acabou por enriquecer o trabalho.

O primeiro tipo de problema envolve a descoberta da árvore geradora e da árvore geradora mínima de um dado grafo. A solução desses problemas pode ser empregada em diferentes áreas como busca de eficiência em projeto de redes de computadores e de transporte, redes de abastecimento de água, redes de telecomunicações e circuitos eletrônicos. Mas, estes problemas pertencem a uma classe que apresenta uma relação muito forte entre os dados gerados durante a computação da solução. Consequentemente, a paralelização de soluções sequenciais ou o projeto de soluções paralelas não é uma tarefa trivial. As soluções propostas se mostraram eficientes, de simples entendimento e fácil paralelização.

O segundo problema, o alinhamento de sequências tanto duas a duas (alinhamento *pairwise*) quanto em conjunto de mais de duas (alinhamento múltiplo) visa a comparação entre as sequências extraídas de material genético, a fim de verificar semelhanças e com isso descobrir informações funcionais, estruturais e evolutivas. As soluções para esse tipo de problema envolve uma grande dependência entre os dados gerados em etapas intermediárias da computação, tornando as soluções mais difíceis de serem paralelizadas.

1.2 Contribuições do trabalho

Dentre as principais contribuições resultantes do trabalho desenvolvido durante o doutorado podemos destacar:

- Dois algoritmos paralelos e eficientes para árvore geradora;
- Dois algoritmos paralelos e eficientes para árvore geradora mínima;
- Implementações paralelas do algoritmo estrela [17] para alinhamento múltiplo de sequências, usando MPI (*Message-Passing Interface*) e CUDA (*Compute Unified DeviceArchitecture*);

- Implementação paralela do algoritmo para alinhamento *pairwise* de sequências; e
- Proposta de solução tolerante a falhas para o alinhamento *pairwise* de sequências.

Parte dessas contribuições está presente em cinco artigos publicados durante o período de doutoramento: [66, 63, 62, 64, 65].

1.3 Estrutura do trabalho

O Capítulo 2 apresenta alguns conceitos básicos sobre computação paralela e a descrição dos ambientes de teste utilizados para avaliar as soluções propostas. No Capítulo 3 são descritos os algoritmos para árvore geradora e árvore geradora mínima que fazem uso das propriedades do grafo bipartido correspondente ao grafo de entrada. Os algoritmos para árvore geradora e árvore geradora mínima, que não utilizam o grafo bipartido, são descritos, respectivamente, nos capítulos 4 e 5. O Capítulo 6 descreve os problemas de alinhamento de sequências e as soluções desenvolvidas para este contexto. Por fim, o Capítulo 7 apresenta as conclusões do trabalho e as possibilidades de trabalhos futuros.

Capítulo 2

Conceitos e Informações Preliminares

2.1 Computação Paralela

É natural pensarmos na utilização de computação paralela para problemas que necessitem de intenso processamento computacional ou trabalhem com grande volume de dados. Para isso, muitas vezes, é necessário reformularmos uma proposta inicialmente sequencial para obtermos uma versão que seja mais adequada para a abordagem paralela. Considerando a característica de problemas atuais que trabalham com entradas de tamanho muito grande, percebe-se o interesse no desenvolvimento de algoritmos paralelos que sejam implementáveis em máquinas paralelas disponíveis e que os tempos de execução obtidos nas implementações sejam compatíveis com o previsto no modelo de computação utilizado, independentemente de arquitetura de interconexão da máquina paralela que estiver sendo utilizada.

Uma opção é o desenvolvimento paralelo de soluções utilizando processadores *multicore* ou GPGPU (*General-Purpose Computing on Graphics Processing Units*). A existência de várias unidades de processamento nessas arquiteturas permite criar diferentes instâncias de um programa para resolução de um problema em menos tempo (SPMD - *Single Program Multiple Data*). Para facilitar o desenvolvimento de um programa paralelo existem bibliotecas que auxiliam a troca de mensagem e o acesso à memória.

2.1.1 Modelo BSP-CGM

No início dos anos 90, Valiant [60] introduziu um modelo simples que fornece uma previsão razoável do desempenho da implementação dos algoritmos projetados. Esse modelo de "granularidade grossa", denominado *Bulk Synchronous Parallel* (BSP), além de ser um dos modelos realísticos mais importantes, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros.

O modelo *Coarse Grained Multicomputer* (CGM), proposto por Dehne et al. [10], tem o propósito de ser um modelo próximo às máquinas paralelas com memória distribuída existentes. Devido a similaridade dos modelos utiliza-se o termo BSP/CGM, que emprega apenas dois parâmetros: o número de processadores p e o tamanho da entrada N. O modelo BSP/CGM é formado por um conjunto de p processadores, cada um com uma

memória local e conectados por um *switch* para o envio de mensagens.

Um algoritmo BSP/CGM consiste em uma sequência de superpassos separados por barreiras de sincronização (Figura 2.1). Cada superpasso é composto de uma combinação de passos de computação local e de comunicação global executados pelos processadores. Essas operações independentes, executadas em cada processador, utilizam dados disponibilizados localmente no início do superpasso, para realizar a computação, e instruções de envio e recebimento de mensagens, para a comunicação.

Em cada rodada de comunicação, cada processador troca, no máximo, $O(\frac{N}{p})$ dados com outros processadores, onde N é o tamanho da entrada e p é o número de processadores, e o custo de comunicação é modelado pelo número de rodadas de comunicação utilizadas. Para estes algoritmos, o maior objetivo é minimizar o número de superpassos e a quantidade de computação local. É esperado que em $O(p)$ rodadas de comunicação, a solução seja encontrada, ou a partir desse momento apenas um dos processadores seja capaz de realizar a computação restante para finalização da solução.

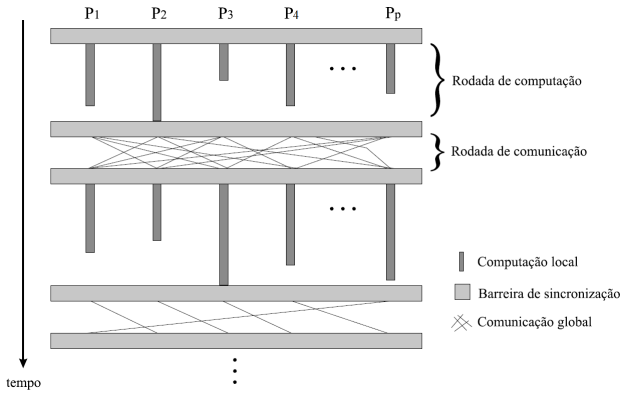


Figura 2.1: Algoritmo BSP/CGM.

O modelo BSP/CGM é adequado para o projeto e análise de algoritmos paralelos onde há muita comunicação entre os processos, visto que separa a comunicação necessária em passos que devem ser intercalados com rodadas de computação. 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 de alinhamento de sequências e da árvore geradora se enquadram nessa classe, o que motiva a utilização do modelo para prever o comportamento e a complexidade do algoritmo.

Deve-se analisar não só os aspectos teóricos do algoritmo, mas também mapear os passos do mesmo para a arquitetura paralela a ser utilizada. No caso de ambientes com GPGPU, as rodadas (superpassos) do modelo BSP/CGM são representados pelas chamadas de cada função *kernel* do CUDA, como apresentado em [31]. Além disso, associamos o conjunto de processadores (p) do modelo BSP/CGM ao conjunto de *streaming multiprocessors* (SM's) da GPGPU. Os *kernels* CUDA são funções desenvolvidas utilizando a linguagem C, estendida para CUDA, que são executadas em paralelo por diferentes *threads* CUDA nos SM's.

Em 2011 Valiant [61] apresentou um modelo hierárquico, com um número arbitrário de níveis, que estende o BSP. Esse modelo, denominado multi-BSP, atende tanto às ar-

quiteturas mono como multi-chip, que apresentam vários níveis de memória e de cache. O objetivo é modelar de forma conjunta todos os níveis de uma arquitetura. Para cada nível são definidos parâmetros explícitos para o número de processadores, tamanho de cache/memória, custos de comunicação e sincronização.

Nossos algoritmos paralelos foram projetados utilizando o modelo BSP/CGM [60, 10]. Este modelo considera um conjunto de p processadores, cada um com uma memória local de tamanho $O(n/p)$, onde n é o tamanho da entrada. Um algoritmo neste modelo executa um conjunto de etapas de computação local (*super passos*) alternando com as fases de comunicação global, separadas por uma barreira de sincronização. O custo da comunicação considera o número super passos necessários para executar o algoritmo.

O modelo BSP/CGM é apropriado para o projeto e análise de algoritmos paralelos, onde há muita comunicação entre os processos. Esta é uma característica de problemas irregulares, isto é, a entrada em cada rodada do programa muda e os processadores precisam da informação que diferentes processadores computaram na última rodada. Os problemas da árvore geradora e árvore geradora mínima estão nessa classe, o que motiva o uso do modelo para prever o comportamento e a complexidade do algoritmo.

O mapeamento de um algoritmo BSP/CGM para um ambiente de memória distribuída é simples. Os super passos consistem em rodadas de computação e comunicação, onde a computação é realizada nos nós e a comunicação é feita através de uma rede. Ao usar o ambiente GPGPU, temos um ambiente de memória compartilhada, e podemos ver as invocações de cada função kernel do CUDA como um super passo do modelo BSP/CGM como afirma Lima et al. [31]. A execução paralela de cada *kernel* pelas várias *threads* criadas pelo CUDA constitui uma rodada de computação, que pode ser alternada pela comunicação entre as *threads* através da memória e a comunicação entre a GPU e a CPU. Considerando essa abordagem, pode-se prever que o algoritmo proposto terá um desempenho compatível a seu comportamento teórico, quando implementado em uma GPGPU.

2.1.2 Algumas Tecnologias Utilizadas

Dentre as tecnologias disponíveis para facilitar o desenvolvimento de soluções que fazem uso de algum tipo de paralelismo foram utilizadas durante deste trabalho MPI e CUDA. Ao projetar e implementar as soluções paralelas aqui apresentadas usamos tanto o modelo BSP-CGM, como as tecnologias MPI e CUDA.

MPI (*Message-Passing Interface*) é uma biblioteca para programas C/C++ e Fortran que oferece funções para facilitar a implementação do envio e recebimento de mensagens em sistemas de memória distribuída [50]. Nesse tipo de sistema cada processador tem sua memória local e se comunica com os demais através da rede de conexão.

CUDA (*Compute Unified Device Architecture*) é uma plataforma de computação paralela e um modelo de programação, criado pela NVIDIA [48], que permite a realização de computação de propósito geral utilizando aceleradores gráficos, GPUs (*Graphics Processing Unit*) fabricadas pela NVIDIA. É comum nomearmos este tipo de uso da GPU para processamento de aplicações tradicionais de GPGPU (*General-Purpose Computing on Graphics Processing Units*). Na arquitetura CUDA os programas tem partes executa-

das na CPU, chamada de *host* e partes executadas na GPU, chamada de *device*. O *host* faz chamada a funções (*kernels* CUDA) para que o *device* execute alguma computação.

Um conceito utilizado quando se trabalha com soluções de múltiplas *threads* é o de *warp* que representa a unidade de escalonamento de *threads* do multiprocessador. Outro conceito importante quando se tem preocupação em aumentar o desempenho, é o de coalescência de memória. O acesso coalescente à memória consiste em acessar dados em endereços contíguos de memória. O acesso não é contíguo à memória deteriora o desempenho da aplicação.

2.2 Ambientes de Teste

Os algoritmos propostos nestes trabalho e descritos nos capítulos seguintes foram testados e analisados utilizando alguns dos seguintes ambientes:

- Ambiente 1 (*cluster* Carleton):
  - *cluster bewolf* com 64 nós de processamento, pertencente ao Laboratório Virtual de Computação de Alto Desempenho (*High Performance Computing Virtual Laboratory* - HPCVL) da Universidade de Carleton;
  - cada nó com 4 processadores Dual-Core AMD Opteron 2,2 GHz, lançado no ano de 2006, com 1024 KB de cache e 8 GB de memória RAM;
  - interconexão entre os nós usando *gigabit ethernet*; e
  - Biblioteca MPI.
- Ambiente 2 (GTX680 - UnB):
  - um computador desktop da Universidade de Brasília;
  - com processador Intel Core i7-3770, lançado no ano de 2012, 8 MB de cache e 8 GB de memória RAM;
  - placa de vídeo Nvidia GeForce GTX 680, com arquitetura Kepler, lançada no ano de 2012, com 1.536 núcleos de processamento e 2 GB de memória; e
  - Sistema operacional Ubuntu 14.04.
- Ambiente 3 (GTX460 - UFMS):
  - um computador desktop da FACOM/UFMS;
  - com processador Intel Core 2 Quad, lançado no ano de 2010, com 6144 KB de cache e 4 GB de memória RAM; e
  - placa de vídeo Nvidia GeForce GTX 460, com arquitetura Ferni, lançada no ano de 2010, com 336 núcleos de processamento e 1024 MB de memória.
  - Sistema operacional Ubuntu 12.04; e
  - CUDA toolkit versão 4.2.
- Ambiente 4 (GTX745 - UFMS):

  - um computador desktop da FACOM/UFMS;
  - com 8 processadores Intel Core i7-4790S, lançado no ano de 2014, com 8 MB de cache e 15 GB de memória RAM;
  - placa de vídeo Nvidia GeForce GTX 745, com arquitetura Kepler, lançada no ano de 2014, com 384 núcleos de processamento e 4 GB de memória;
  - Sistema operacional Ubuntu 16.04; e
  - CUDA toolkit versão 8.0.

- Ambiente 5 (M4000 - UFMS):
  - um computador desktop da FACOM/UFMS;
  - com 8 processadores Intel Xeon E5-1620 v3, lançado no ano de 2016, com 10 MB de cache e 32 GB de memória RAM;
  - placa de vídeo Nvidia Quadro M4000, com arquitetura Maxwell, lançada no ano de 2015, com 1664 núcleos de processamento e 8 GB de memória;
  - Sistema operacional Ubuntu 16.04; e
  - CUDA toolkit versão 8.0.

- Ambiente 6 (K40M - UFG):
  - um servidor do INF/UFG;
  - com 40 processadores Intel Xeon E5-2650 v3, lançado no ano de 2014, com 25600 KB de cache e 126 GB de memória RAM;
  - placa de vídeo Nvidia Tesla K40M, com arquitetura Kepler, lançada no ano de 2013, com 2880 núcleos de processamento e 12 GB de memória;
  - Sistema operacional Debian GNU/Linux 8; e
  - CUDA toolkit versão 7.5.

Capítulo 3

Algoritmos Paralelos para Árvores Geradoras Usando Grafo Bipartido

A computação da árvore geradora (*Spanning Tree* - ST), árvore geradora mínima (*Minimum Spanning Tree* - MST) e componentes conexos de um grafo são problemas fundamentais na Teoria dos Grafos 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. Graham e Hell [15] apontam a importância do problema da árvore geradora mínima no projeto de redes de computadores e de transporte, redes de abastecimento de água, redes de telecomunicações e circuitos eletrônicos. Uma pesquisa sobre as muitas facetas do problema da árvore geradora mínima pode ser encontrada no artigo [36].

Este capítulo está organizado da seguinte maneira. Uma introdução ao assunto e uma revisão da literatura serão abordadas na Seção 3.1. O funcionamento dos algoritmos para a árvore geradora e árvore geradora mínima são apresentados na Seção 3.2. Na Seção 3.3, é apresentado um algoritmo paralelo para árvore geradora e na Seção 3.4 as diferenças para encontrar a árvore geradora mínima. Discute-se a correção dos algoritmos na Seção 3.5. Os resultados experimentais são mostrados na Seção 3.6 e, finalmente, as conclusões e contribuições integram a última Seção 3.7.

3.1 Introdução e Conceitos Básicos

Existem algoritmos sequenciais ótimos, baseados em busca em profundidade e em busca em largura [28], 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. No entanto, as soluções paralelas não usam esses métodos de busca por não serem fáceis de paralelizar [53].

Os principais algoritmos paralelos para esse problema são baseados na solução proposta por Hirschberg et al. [19], onde os vértices do grafo são sucessivamente combinados em

super vértices maiores, ou no algoritmo de Borůvka [3]. Usando essas abordagens, vários algoritmos paralelos para o problema da árvore geradora foram propostos [9, 20].

Com o aumento da capacidade das placas gráficas, surge a oportunidade de analisar o comportamento dos algoritmos BSP/CGM nessa arquitetura. Um dos principais limitantes dos *speedups* 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, onde p é o número de processadores, para a obtenção da árvore geradora do grafo de entrada, sendo que o uso das GPGPU tem a vantagem de estabelecer a comunicação através da memória compartilhada.

Alguns trabalhos propondo soluções paralelas para o problema da árvore geradora mínima (MST) usando a GPGPU podem ser encontrados na literatura, como os apresentados em [67, 44, 41, 35]. O algoritmo proposto em [44] é inspirado no algoritmo de Prim [52]. Os artigos [67], [41] e [35] apresentaram soluções paralelas baseadas no algoritmo de Borůvka [3].

O algoritmo para árvore geradora apresentado em [4] usa como entrada um grafo bipartido, pois um grafo geral pode ser transformado em um grafo bipartido correspondente. No entanto, os autores usam uma decomposição do grafo que não parece adequada para encontrar as arestas mais leves da árvore geradora e, consequentemente, computar a árvore geradora mínima do grafo de entrada.

Neste capítulo é apresentada uma abordagem para obter uma árvore geradora e uma árvore geradora mínima de um dado grafo que não precisa resolver o caminho euleriano nem o problema de *list ranking*. A proposta usa alguns conceitos apresentados em [4], mas inicia com a criação de um grafo bipartido correspondente ao grafo de entrada, ou seja, é aplicável a qualquer grafo conexo. Os algoritmos paralelos descritos neste capítulo foram projetados usando o modelo BSP/CGM, com $O(\log p)$ rodadas de comunicação e tempo de computação local $O(\frac{n+m}{p})$, onde p é o número de processadores, n o número de vértices e m o número de arestas do grafo. Implementando e testando os algoritmos propostos em máquinas paralelas reais demostrou-se que estes apresentam bom desempenho. Os testes foram realizados utilizando diferentes modelos de GPUs. Os resultados obtidos mostraram que os algoritmos são escaláveis e têm *speedups* competitivos.

Devido à disponibilidade do código-fonte e por apresentar um desempenho superior para grafos não muito esparsos, comparou-se os resultados da implementação do algoritmo proposto para árvore geradora mínima com um algoritmo sequencial eficiente publicado recentemente [33], chamado de *Edge Pruned Minimum Spanning Tree* (EPMST). Este algoritmo apresenta melhor desempenho em comparação com a solução *Filter-Kruskal* [49]. Comparou-se os resultados do EPMST com os resultados alcançados pela versão paralela e pela versão para CPU do algoritmo proposto neste capítulo.

3.1.1 Conceitos básicos e Notações

Agora descreve-se alguns conceitos básicos usados nos algoritmos propostos. Considere $G = (V, E)$ um **grafo** simples e não orientado, 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, w_{ij}, v_j), onde v_i e v_j são vértices de

V e w_{ij} é o peso da aresta. Para simplificar, pode-se representar uma aresta apenas como (v_i, v_j), estando o peso w_{ij} implícito. Para o problema da árvore geradora o peso das arestas é ignorado.

Um **caminho** em G é uma sequência de arestas $(v_1, v_2), (v_2, v_3), (v_3, v_4) \ldots, (v_{k-1}, v_k)$ conectando vértices distintos $v_1, \ldots, v_k$ de G. Um **ciclo** é um caminho que conecta diferentes vértices $v_1, v_2, \ldots, v_k$, tal que $v_1 = v_k$. Um **grafo conexo** tem pelo menos um caminho para cada 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. Um subgrafo de um grafo G é um grafo cujo conjunto de vértices é um subconjunto do conjunto de vértices G e o conjunto de arestas é um subconjunto do conjunto de arestas de G. Uma **árvore geradora** (ST) de $G = (V, E)$ é uma árvore $T = (V, E')$ que inclui todos os vértices de G e é um subgrafo de G, isto é, todas as arestas de T pertencem a G, $E' \subset E$. Uma árvore geradora de G também pode ser definida como o conjunto maximal de arestas de G tal que $|E'| = |V| - 1$ e que não contém nenhum ciclo. Uma **árvore geradora mínima** (MS) é uma árvore geradora cuja soma dos pesos de suas arestas seja o mínimo possível.

Um **grafo bipartido** é um grafo cujos vértices podem ser divididos em dois conjuntos disjuntos V_1 e V_2, de forma que cada aresta do grafo conecta um vértice em V_1 e um vértice em V_2. Os algoritmos propostos leem o grafo de entrada $G = (V, E)$ e criam um grafo bipartido correspondente $H = (V \cup U, E')$ baseado em G, onde V e U formam os conjuntos de vértices. Os detalhes para obter o grafo bipartido correspondente ao grafo de entrada original serão mostrados a seguir.

O conceito denominado **esteio** ou **strut** é usado nos algoritmos, mas difere da definição apresentada em [5]. No contexto desta tese, um **esteio**, representado por S, é definido como uma floresta de H, tal que cada aresta de S incide em exatamente um vértice de V. Os detalhes da escolha das arestas do esteio dependem se está sendo considerado o problema da árvore geradora ou o problema da árvore geradora mínima e serão fornecidos posteriormente. O **grau** de um vértice de um grafo é o número de arestas incidentes ao vértice. Um vértice u_j é considerado um vértice **zero-diferença** em S se o grau do mesmo em S for igual ao grau do mesmo em H, ou seja $d_H(u_j) - d_S(u_j) = 0$, onde $d_H(u_j)$ indica o grau de u_j em H e $d_S(u_j)$ o grau de u_j em S.

3.2 Ideia geral dos algoritmos propostos para árvore geradora (ST) e uma árvore geradora mínima (MST)

Inicia-se esta seção expondo os conceitos básicos utilizados nos algoritmos. Na sequência o algoritmo básico é apresentado e depois suas etapas são detalhadas.

3.2.1 O algoritmo paralelo básico

O algoritmo 1 apresenta as principais ideias dos algoritmos paralelos propostos apresentados neste capítulo. Ele usa o paradigma SIMD, ou seja, as mesmas etapas são executadas por vários processadores que encontram a solução colaborativamente. Os algoritmos para árvore geradora e árvore geradora mínima são muito semelhantes, ambos seguindo o Al-

goritmo 1. A principal diferença consiste na maneira como o esteio é construído. Como no modelo BSP/CGM são esperadas $O(\log p)$ rodadas, estabeleceu-se que, após $\log p$ rodadas, se a árvore geradora ou a árvore geradora mínima não for encontrada, o algoritmo continua o processamento localmente na CPU.

3.2.2 Criando um grafo bipartido correspondente ao grafo de entrada (Algoritmo 1 - linha 3 a linha 9)

Para encontrar uma árvore geradora (ou árvore geradora mínima) de um determinado grafo, primeiro cria-se um grafo bipartido correspondente ao grafo de entrada. Este passo é executado em paralelo e pode ser feito adicionando um novo vértice em cada aresta (linha 4) do grafo original, subdividindo assim cada aresta original em duas novas arestas (linha 5). Se os vértices do grafo original forem considerados como pertencentes a uma partição e os vértices recém-adicionados como pertencentes a uma segunda partição, então tem-se um grafo bipartido resultante.

De maneira mais formal, considere um grafo conexo $G = (V, E)$, onde V é um conjunto de n vértices $\{v_1, v_2, \ldots, v_n\}$ e E é um conjunto de m arestas (v_i, v_j), onde v_i e v_j são vértices de V. Cada aresta (v_i, v_j) tem um peso denotado por w_{ij}. Pode-se criar um grafo bipartido correspondente H adicionando um conjunto U de m novos vértices $(u_1, u_2, \ldots, u_m)$ e substituindo cada aresta (v_i, v_j) de E por duas arestas (v_i, u_k) e (v_j, u_k), ambas com peso igual a w_{ij}. Seja E' o conjunto de arestas (v_i, u_k) para todos os $v_i \in V$ e $u_k \in U$. O grafo $H = (V \cup U, E')$, no algoritmo representado por $H = (V_H \cup U_H, E_H)$, assim obtido, é bipartido. Lembre-se de que os pesos são usados apenas para encontrar uma árvore geradora mínima. No algoritmo para calcular uma árvore geradora, ignora-se o peso das arestas.

A Figura 3.1 mostra um exemplo desta etapa. À esquerda, tem-se o grafo original $G = (V, E)$ com $V = \{1, 2, \ldots, 5\}$. No meio, é representado o grafo bipartido correspondente $H = (V_H \cup U_H, E_H)$ com $U_H = \{\bar{1}, \bar{2}, \ldots, \bar{8}\}$. E à direita, o mesmo grafo bipartido é mostrado em outra visão, onde os dois conjuntos de vértices V_H e U_H são ilustrados separadamente. Observe que qualquer vértice $u_k \in U_H$ que, por construção, foi criado a partir de uma aresta (v_i, v_j) do grafo original G, sempre tem grau dois e ambas as arestas incidentes em u_k têm o mesmo peso. Há uma correspondência um-para-um entre uma aresta (v_i, v_j) do grafo original G e um vértice adicionado u_k. Será usada a notação $aresta\_original(u_k)$ para indicar a aresta original (v_i, v_j). Diz-se também que a aresta (v_i, v_j) é *associada* a u_k.

3.2.3 Obtendo o esteio ao computar a árvore geradora (Algoritmo 1 - linha 15 a linha 24)

Considere o grafo bipartido correspondente $H = (V_H \cup U_H, E_H)$ com conjuntos de vértices $V_H = \{v_1, v_2, \ldots, v_n\}$ e $U_H = \{u_1, u_2, \ldots, u_m\}$ e conjunto de arestas E_H onde cada aresta liga um vértice de V_H a um vértice de U_H. Por simplificação, considere que cada vértice v_i de V_H seja representado por i, ou seja, $V_H = \{1, 2, \ldots, n\}$. Da mesma forma, cada vértice u_j de U_H será representado por $\bar{j}$, ou seja, $U_H = \{\bar{1}, \bar{2}, \ldots, \bar{m}\}$. Para computar

Algoritmo 1 Algoritmo básico para árvores geradoras usando grafo bipartido

Entrada: Um grafo conexo $G = (V, E)$, onde $V = \{v_1, v_2, \ldots, v_n\}$ é um conjunto com n vértices e E é um conjunto com m arestas (v_i, v_j), onde v_i e v_j são vértices de V. Cada aresta (v_i, v_j) tem um peso correspondente representado por w_{ij}.
Saída: Uma árvore geradora (ou uma árvore geradora mínima) de G cujas arestas estão em *ConjuntoSolucao*.

1: *ConjuntoSolucao* := vazio.
2: // Criação do grafo bipartido $H = (V_H \cup U_H, E_H)$ correspondente ao grafo de entrada G.
3: **for each** $(e_i(v_a, v_b) \in E)$ **in parallel do**
4: Adicione o vértice u_i a U_H // u_i é um novo vértice associado a aresta e_i
5: Adicione as arestas (v_a, w_{ab}, u_i) e (v_b, w_{ab}, u_i) a E_H
6: **end for**
7: **for each** $(v_i \in V)$ **in parallel do**
8: Adicione o vértice v_i to V_H
9: **end for**
10: // Encontrando a solução para ST ou MST.
11: *terminou* := false
12: $r := 0$
13: **while** $((r < \log p)$ **AND** (*terminou* = false)) **do** // p é o número de processadores
14: // Obtendo o esteio.
15: **for each** $(u_i \in U_H)$ **in parallel do**
16: $d_S(u_i) = 0$
17: **end for**
18: **for each** $(v_i \in V_H)$ **in parallel do**
19: Encontre a menor aresta entre todas as arestas incidentes a v_i
20: **end for**
21: **for each** $(v_i \in V_H)$ **in parallel do**
22: // considerando que (v_i, u_j) é a menor aresta entre todas as arestas incidentes a v_i
23: $d_S(u_j) = d_S(u_j) + 1$ // função atômica
24: **end for**
25: // Adicionando arestas a *ConjuntoSolucao*
26: **for each** $(u_j \in U_H)$ **in parallel do**
27: // u_j é o vértice que corresponde a aresta e_j do grafo original, criado na linha 4
28: **if** $(d_S(u_j) \geq 1)$ **then**
29: Adicione a aresta e_j, onde e_j corresponde a $aresta\_original(u_j)$, a *ConjuntoSolucao*
30: **end if**
31: **end for**
32: // Calculando o número de vértices *zero-diferença*
33: *numzerodif* = 0 // *numzerodif* é o número de vértices *zero-diferença* de U
34: **for each** $(u_j \in U_H)$ **in parallel do**
35: **if** $(d_S(u_j) == 2)$ **then**
36: *numzerodif* = *numzerodif* +1 // função atômica
37: **end if**
38: **end for**
39: **if** (*numzerodif* == 1) **then**
40: *terminou* := true
41: **else** // Compactação do grafo H
42: **for each** (vértice u_t, com arestas (v_i, u_t) e (v_j, u_t) em E_H) **in parallel do**
43: **if** $(d_S(u_t) \geq 1)$ **then**
44: Unir os vértices adjacentes no esteio em um super-vértice v_i, sendo v_i o menor rótulo entre eles
45: Atualizar para v_i as arestas de E_H incidentes aos vértices unidos
46: Eliminar os demais vértices unidos de V_H
47: **end if**
48: **end for**
49: **for each** (vértice u_t, com arestas (v_a, u_t) e (v_b, u_t) em E_H) **in parallel do**
50: **if** $(v_a == v_b)$ **then** // v_a e v_b foram unidos no "for" anterior
51: Eliminar u_t de U_H
52: Eliminar (v_a, u_t) e (v_b, u_t) de E_H
53: **end if**
54: **end for**
55: **end if**
56: $r := r + 1$
57: **end while**

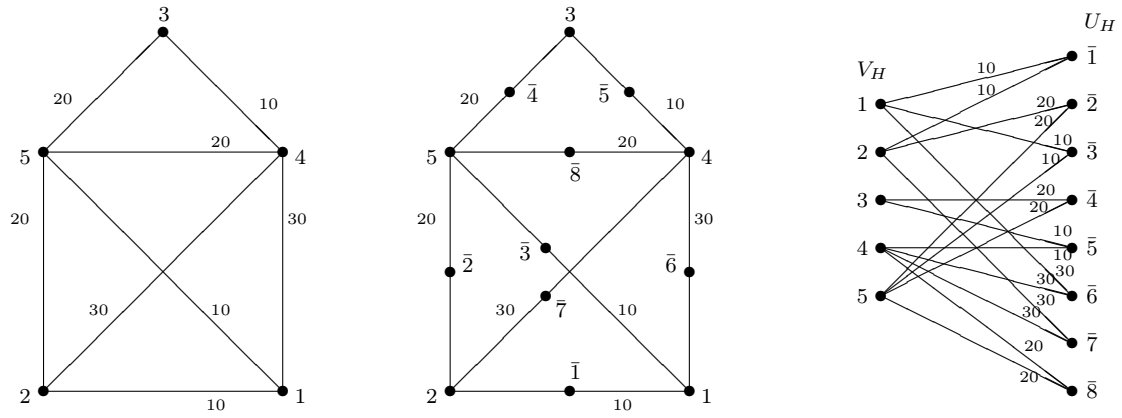


Figura 3.1: À esquerda, o grafo original $G = (V, E)$, no meio, o grafo bipartido correspondente $H = (V_H \cup U_H, E_H)$ e, à direita, o mesmo grafo bipartido separando V_H e U_H.

uma árvore geradora, o esteio é obtido da seguinte maneira. Entre todas as arestas (i, j) incidentes a i em H, selecione a aresta (i, k) com o menor valor k.

Como exemplo, considere o grafo bipartido correspondente $H = (V_H \cup U_H, E_H)$ da Figura 3.1, em que $V_H = \{v_1, v_2, \ldots, v_5\} = \{1, 2, \ldots, 5\}$ e $U_H = \{u_1, u_2, \ldots, u_8\} = \{\bar{1}, \bar{2}, \ldots, \bar{8}\}$. Para cada vértice v_i de V_H, considere todas as arestas (v_i, u_j) incidentes a v_i. A Tabela 3.1 ilustra cinco grupos de arestas (v_i, u_j), um grupo para cada $v_i = 1, 2, \ldots, 5$.

Tabela 3.1: Arestas do grafo H representado na Figura 3.1. As arestas com o menor u_k escolhidas para compor o esteio estão marcadas com um asterisco (“ * ”).

$(v_i$	$u_j)$	$(v_i$	$u_j)$	$(v_i$	$u_j)$	$(v_i$	$u_j)$	$(v_i$	$u_j)$
(1	$\bar{6}$)	* (2	$\bar{1}$)	* (3	$\bar{4}$)	(4	$\bar{6}$)	* (5	$\bar{2}$)
(1	$\bar{3}$)	(2	$\bar{7}$)	(3	$\bar{5}$)	(4	$\bar{7}$)	(5	$\bar{3}$)
* (1	$\bar{1}$)	(2	$\bar{2}$)			(4	$\bar{8}$)	(5	$\bar{8}$)
						* (4	$\bar{5}$)	(5	$\bar{4}$)

Por definição, para encontrar uma árvore geradora, o esteio S é formado pelas arestas com o menor u_k para cada vértice v_i (na Tabela 3.1 marcadas “ * ”) a saber, (1 $\bar{1}$), (2 $\bar{1}$), (3 $\bar{4}$), (4 $\bar{5}$) e (5 $\bar{2}$). Na Figura 3.2, o esteio S obtido é ilustrado pelas linhas pontilhadas. A propósito de notação, uma aresta do esteio pode ser referenciada com uma *aresta-esteio*.

3.2.4 Obtendo o esteio ao computar a árvore geradora mínima (Algoritmo 1 - linha 15 a linha 24)

Considerando o cálculo de uma árvore geradora mínima, agora o esteio é obtido da seguinte maneira. Entre todas as arestas (v_i, u_j) incidentes a v_i em H, escolha a aresta com menor peso e, se houver várias arestas (v_i, u_k) com o mesmo menor peso, selecione a aresta (v_i, u_k) com o menor u_k.

Como exemplo de obtenção do esteio, considere o grafo bipartido correspondente $H = (V_H \cup U_H, E_H)$ da figure 3.1. Para cada vértice v_i de V, considere todas as arestas (v_i, u_j) incidentes a v_i apresentadas na Tabela 3.2. Para melhor ilustrar o exemplo, estendeu-se a

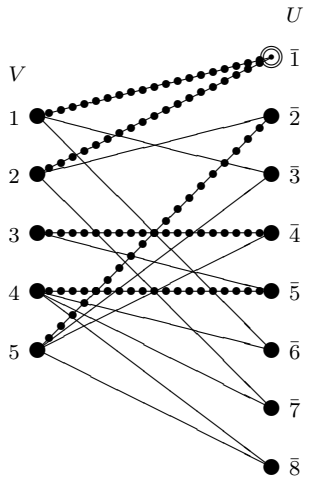


Figura 3.2: O esteio S representado por linhas pontilhadas no caso da árvore geradora. Os vértices $\bar{1}$, $\bar{2}$, $\bar{4}$ e $\bar{5}$ têm pelo menos uma *aresta-esteio* incidente. O vértice $\bar{1}$ tem duas *arestas-esteio* incidentes e, por isso, é um vértice *zero-diferença*.

notação de aresta (v_i, u_j) adicionando o peso w_{ij} no meio, resultando em $(v_i\ w_{ij}\ u_j)$. Na Tabela 3.2 são representados os cinco grupos de arestas $(v_i\ w_{ij}\ u_j)$, um grupo para cada vértice $v_i = 1, 2, \ldots, 5$.

Pela definição usada no cálculo de uma árvore geradora mínima, o esteio S é composto das arestas mais leves, considerando o peso e o rótulo do vértice u. A Tabela 3.2 mostra as arestas escolhidas para cada vértice marcadas com " * ", a saber, $(1\ \ 10\ \ \bar{1})$, $(2\ \ 10\ \ \bar{1})$, $(3\ \ 10\ \ \bar{5})$, $(4\ \ 10\ \ \bar{5})$ e $(5\ \ 10\ \ \bar{3})$. Na Figura 3.3, o esteio S obtido é ilustrado pelas linhas pontilhadas.

Tabela 3.2: Arestas do grafo H representado na Figura 3.1 para o caso da árvore geradora mínima. As arestas mais leves, escolhidas para compor o esteio, estão marcadas com um asterisco (" * ").

$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$
(1	30	$\bar{6}$)	* (2	10	$\bar{1}$)	(3	20	$\bar{4}$)	(4	30	$\bar{6}$)	(5	20	$\bar{2}$)
(1	10	$\bar{3}$)	(2	30	$\bar{7}$)	* (3	10	$\bar{5}$)	(4	30	$\bar{7}$)	* (5	10	$\bar{3}$)
* (1	10	$\bar{1}$)	(2	20	$\bar{2}$)				(4	20	$\bar{8}$)	(5	20	$\bar{8}$)
									* (4	10	$\bar{5}$)	(5	20	$\bar{4}$)

Observe que pode-se assumir, sem perda de generalidade, que os pesos de todas as arestas $e \in E$ do grafo de entrada G sejam diferentes. É necessário apenas modificar o peso original de cada aresta da seguinte maneira. Considere uma aresta $(v_i, v_j) \in E$ e o rótulo do vértice u_k adicionado a esta aresta para construção do grafo bipartido. Se o novo peso considerado para (v_i, v_j) for a concatenação do peso original e o rótulo u_k, todos os novos pesos das arestas serão diferentes. Por exemplo, na Figura 3.1, os pesos originais das arestas $(1, 2)$, $(1, 5)$ e $(3, 4)$ são os mesmos (igual a 10). Os novos pesos dessas arestas podem ser 101, 103 e 105, respectivamente. Isso torna a etapa de construção do esteio ainda mais fácil. Para cada $v_i \in V$, a aresta do esteio é a aresta (v_i, u_j) com o menor peso novo. Esta suposição será utilizada na Seção 3.5 para simplificar a prova de

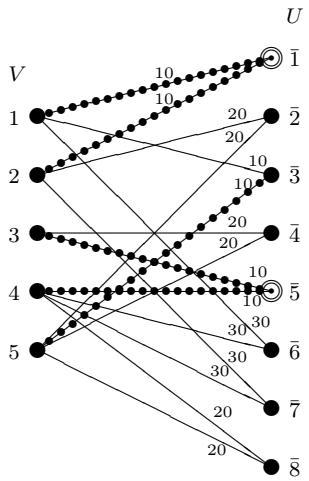


Figura 3.3: O esteio S representado por linhas pontilhadas no caso da árvore geradora mínima. Os vértices $\bar{1}$, $\bar{3}$ e $\bar{5}$ têm pelo menos uma *aresta-esteio* incidente. Os vértices $\bar{1}$ e $\bar{5}$ têm duas *aresta-esteio* incidentes e, por isso, são vértices *zero-diferença*.

correção. Pode-se ver que se (v_i, u_j), com peso w_{ij}, é uma *aresta-esteio*, então nenhuma aresta (v_i, u_k) em E_H tem peso menor que o peso w_{ij}.

3.2.5 Adicionando arestas ao conjunto solução (Algoritmo 1 - linha 26 a linha 31)

Ao final do algoritmo o conjunto solução deve conter as arestas da árvore geradora ou da árvore geradora mínima. Inicialmente este conjunto está vazio (linha 1). Depois de obter o esteio S, o algoritmo localiza todos os vértices u_j que são incidentes a arestas do esteio (ou seja, $d_S(u_j) \geq 1$) e adiciona $aresta\_original(u_j)$ ao conjunto solução.

Veja novamente Figura 3.2. Os vértices $\bar{1}, \bar{2}, \bar{4}$ e $\bar{5}$ são incidentes a arestas do esteio. Assim, adiciona-se ao conjunto solução $aresta\_original(\bar{1})$, $aresta\_original(\bar{2})$, $aresta\_original(\bar{4})$ e $aresta\_original(\bar{5})$ (respectivamente arestas (1, 2), (2, 5), (3, 5) e (3, 4)). A Figura 3.4 mostra as arestas adicionadas ao conjunto solução até o momento, na primeira rodada do algoritmo, para o caso da árvore geradora. As arestas adicionadas são mostradas através de linhas pontilhadas.

Para o exemplo da Figura 3.3, que considera o problema da árvore geradora mínima, os vértices $\bar{1}$, $\bar{3}$ e $\bar{5}$ são incidentes a arestas do esteio. Então, são adicionadas ao conjunto solução $aresta\_original(\bar{1})$, $aresta\_original(\bar{3})$ e $aresta\_original(\bar{5})$ (respectivamente arestas (1 , 2), (1, 5) e (3, 4)). A Figura 3.5 mostra as arestas adicionadas ao conjunto solução através de linhas pontilhadas, após a primeira rodada do algoritmo, no caso da árvore geradora mínima.

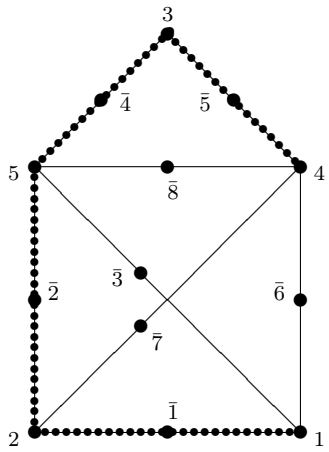


Figura 3.4: Conjunto solução composto pelas arestas do esteio, após a primeira rodada do algoritmo, no caso da árvore geradora.

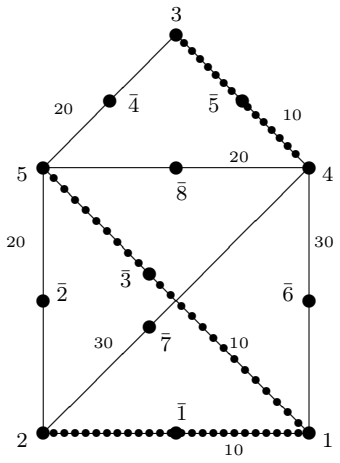


Figura 3.5: Conjunto solução intermediário composto pelas arestas do esteio, após a primeira rodada do algoritmo, no caso da árvore geradora mínima.

3.2.6 Calculando o número de vértices *zero-diferença* (Algoritmo 1 - linha 33 a linha 38)

Considere o grafo bipartido $H = (V_H \cup U_H, E_H)$ e o esteio S. Seja u_j um vértice de U_H. Como visto anteriormente, o grau de qualquer vértice u_j em H, ou $d_H(u_j)$, é sempre igual a dois. Assim, o vértice u_j é *zero-diferença* se seu grau em S também for dois. Na Figura 3.2 o vértice $\bar{1}$ tem duas arestas no esteio e, portanto, é um vértice *zero-diferença*. Na Figura 3.3 os vértices $\bar{1}$ e $\bar{5}$ ambos possuem duas arestas no esteio e, portanto, são vértices de *zero-diferença*. Nas Figuras 3.2 e 3.3 os vértices *zero-diferença* são envolvidos por círculos duplos ao redor de seus rótulos.

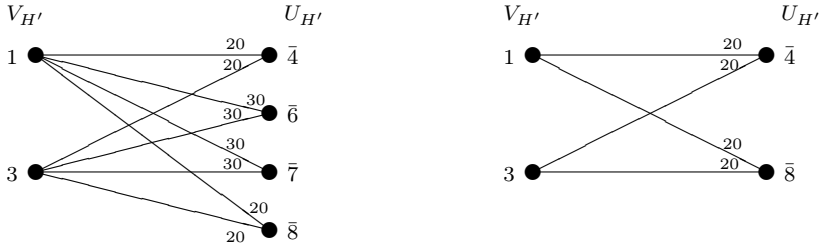
Se houver apenas um vértice *zero-diferença* no esteio obtido, o problema está resolvido. Veja a prova deste fato na Seção 3.5. Assim, o conjunto de arestas que compõem a solução apresentado na Figura 3.4 está completo e representa uma árvore geradora para o grafo de entrada. No entanto, o conjunto de arestas da solução apresentado na Figura 3.5 ainda não representa uma árvore geradora mínima. O algoritmo precisará de outra rodada para

encontrar a solução completa neste caso.

3.2.7 Compactando o grafo bipartido (Algoritmo 1 - linha 42 a linha 54)

Quando há dois ou mais vértices *zero-diferença*, deve-se compactar o grafo bipartido H para a execução de uma nova iteração (rodada) do algoritmo. Para fazer isso, deve-se analisar cada vértice $u_t \in U_H$ que é incidente a uma aresta do esteio (ou seja, $d_S(u_t) \geq 1$). Os vértices v_i e v_j ($v_i, v_j \in V$, tal que $v_i < v_j$) adjacentes a u_t devem ser unidos em um super-vértice v_i (mantendo o menor rótulo dos vértices). Sejam as arestas (v_i, u_t), (v_j, u_t) e (v_k, u_r) arestas do esteio e suponha que v_i ou v_j sejam adjacentes a u_r, por consequência, os v_i, v_j e v_k serão todos unidos no mesmo super-vértice no grafo compactado. Observe o exemplo da Figura 3.3, as arestas do esteio são $(1, \bar{1})$, $(2, \bar{1})$, $(3, \bar{5})$, $(4, \bar{5})$ e $(5, \bar{3})$. Como a outra aresta incidente a $\bar{3}$ é $(1, \bar{3})$, os super-vértices resultantes são $\{1, 2, 5\}$, rotulado com 1 no grafo compactado, e $\{3, 4\}$, rotulado com 3.

O algoritmo também realiza a supressão de vértices de U_H que são adjacentes a vértices de V_H que foram unidos. Isto é ilustrado com o exemplo na Figura 3.3, onde os vértices $\bar{1}$, $\bar{2}$, $\bar{3}$ e $\bar{5}$ devem ser suprimidos. A Figura 3.6 (a) mostra o resultado da compactação, onde os vértices originais 2 e 5 foram unidos ao vértice 1, e o vértice original 4 foi unido ao vértice 3. No novo grafo bipartido $H' = (V_{H'} \cup U_{H'}, E'_{H'})$, resultante da compactação, $|V_{H'}|$ é igual ao número de vértices *zero-diferença* do esteio S (veja o Corolário 2 na Seção 3.5).



(a) (b)

Figura 3.6: Para o caso da árvore geradora mínima: (a) Grafo bipartido H' após a compactação. (b) Grafo bipartido H' após a compactação otimizada.

O grafo resultante após a compactação pode ter vários vértices de $U_{H'}$ que são adjacentes ao mesmo par de vértices de $V_{H'}$. No exemplo da Figura 3.6 (a), todos os vértices $\bar{4}$, $\bar{6}$, $\bar{7}$ e $\bar{8}$ são adjacentes aos vértices 1 e 3. Para otimizar o algoritmo, podemos remover os vértices de $U_{H'}$ que possuem as arestas mais pesadas. A Figura 3.6 (b) mostra o grafo compacto resultante para essa otimização.

3.2.8 Finalizando o algoritmo (outra iteração do Algoritmo 1)

Na etapa de compactação, atualizamos os vértices e arestas do grafo bipartido H, para facilitar a compreensão aqui chamado de H'. Na segunda iteração (rodada), o algoritmo obtém um novo esteio para H', atualiza o conjunto de arestas da solução, calcula o número de vértices *zero-diferença* e repete todo o processo até que haja apenas um vértice *zero-diferença*. Para obter um esteio para este grafo compactado, o mesmo procedimento

descrito anteriormente é repetido. Para cada vértice v_i de $V_{H'}$, considere todas as arestas (v_i, u_j) incidentes a v_i e escolha a aresta mais leve. Considerando o grafo da Figura 3.6 (a), o resultado desta escolha é ilustrado na Tabela 3.3, onde agora tem-se dois grupos $(v_i\ w_{ij}\ u_j)$, um para cada v_i, 1 e 3. (Observe que aqui o vértice 1 representa a compactação dos vértices originais 1, 2 e 5, e o vértice 3 representa a compactação dos vértices originais 3 e 4.). As arestas mais leves, escolhidas para compor o esteio da segunda iteração do exemplo estão marcadas com asterisco ("*").

No modelo BSP/CGM, é esperado que o algoritmo tenha $O(\log p)$ rodadas, portanto, foi estabelecido que, após $\log p$ rodadas, se a árvore geradora ou a árvore geradora mínima não for encontrada, o algoritmo continua o processamento localmente na CPU.

Tabela 3.3: Arestas do grafo da Figura 3.6 (a)

$(v_i$	w_{ij}	$u_j)$		$(v_i$	w_{ij}	$u_j)$
* (1	20	$\bar{4}$)		* (3	20	$\bar{4}$)
(1	30	$\bar{6}$)		(3	30	$\bar{6}$)
(1	30	$\bar{7}$)		(3	30	$\bar{7}$)
(1	20	$\bar{8}$)		(3	20	$\bar{8}$)

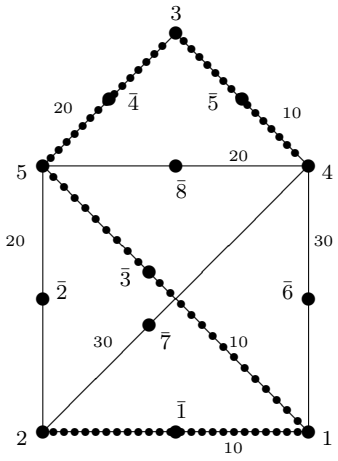


Figura 3.7: A árvore geradora mínima resultante para o exemplo da Figura 3.1.

O novo esteio é composto pelas arestas correspondentes à primeira linha de cada um dos dois grupos da Tabela 3.3, ou seja, (1 20 $\bar{4}$) e (3 20 $\bar{4}$). Tendo obtido o esteio, adicionamos a $aresta\_original(\bar{4})$ ao conjunto solução. Observe que agora tem-se apenas um vértice *zero-diferença* e, portanto, o algoritmo termina. A Figura 3.7 mostra a árvore geradora mínima obtida para o grafo da Figura 3.1. As arestas adicionadas ao conjunto solução são mostradas através de linhas pontilhadas.

3.3 Algoritmo paralelo proposto para computar a árvore geradora (ST)

Os Algoritmos 2 e 3 apresentam mais detalhes do algoritmo paralelo para GPU que computa uma árvore geradora utilizando o grafo bipartido. Como mencionado anteriormente, o algoritmo foi projetado usando o modelo BSP/CGM [60, 10] e executa um conjunto de passos de computação local (super passos) alternando com as fases de comunicação global, separadas por uma barreira de sincronização. Observe que no Algoritmo 1, é possível identificar muitas etapas que são executadas em paralelo, por exemplo, da linha 3 à linha 9. Essas linhas são um exemplo de super passo do algoritmo. O Algoritmo 2 apresenta as chamadas de funções *kernel* CUDA representando as etapas descritas no Algoritmo 1. O Algoritmo 3 detalha algumas das funções *kernel* CUDA chamadas no Algoritmo 2.

Algoritmo 2 Implementação em alto nível

Entrada: Um grafo conexo $G = (V, E)$, 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), onde v_i e v_j são vértices de V.
Saída: Uma árvore geradora de G cujas arestas estão em *ConjuntoSolucao*.

1: *ConjuntoSolucao* := vazio.
2: // Criação do grafo bipartido $H = (V_H \cup U_H, E_H)$ correspondente ao grafo de entrada G.
3: copiar os vértices e arestas do grafo de entrada para a GPU
4: **kernel call** CriaGrafoBipartido()
5: // Encontrando a solução para a ST.
6: *terminou* := false
7: $r := 0$
8: **while** $(r < \log p)$ **AND** (*terminou*$= false$) **do** // p é o número de processadores
9: // Obtendo o esteio S.
10: **kernel call** InicializaArestasLeves()
11: **kernel call** EncontraArestasLeves()
12: // Obtendo o esteio e calculando o número de vértices *zero-diferença*
13: *numzerodif* = 0
14: **kernel call** ObtemEsteioANDCalculaNumzerodif()
15: // Adicionando arestas com $d_S(u) \geq 1$ ao ConjuntoSolucao
16: **kernel call** AtualizaConjuntoSolucao()
17: **if** (*numzerodif* == 1) **then**
18: *terminou* := true
19: **else** // Compactação do grafo H
20: **kernel call** ComputaComponentesConexos()
21: **kernel call** MarcaArestasParaDelecao()
22: **kernel call** ReorganizaArestas()
23: **kernel call** AtualizaVerticesV()
24: **kernel call** AtualizaVerticesU()
25: **kernel call** AtualizaArestas()
26: **end if**
27: $r := r + 1$
28: **end while**
29: copia *ConjuntoSolucao* para CPU

Algoritmo 3 Algoritmo para ST com grafo bipartido - Descrição de Procedimentos

```
1: Procedure CriaGrafoBipartido
2: // thread_id é a identificação de cada thread durante a execução da função kernel
3: // nOG é o número de vértices do grafo original
4: // mOG é o número de arestas do grafo original
5: // considere cada aresta do grafo original representada pelos vértices v1 e v2
6: // considere cada aresta do grafo bipartido representada pelos vértices v e u
7: if (thread_id < mOG) then
8:   if (thread_id < nOG) then
9:     V_H[thread_id].id = V[thread_id].id;
10:  end if
11:  U_H[thread_id].id = thread_id;
12:  E_H[2 * thread_id].v = E[thread_id].v1;
13:  E_H[2 * thread_id].u = U_H[thread_id]]).id;
14:  E_H[2 * thread_id + 1].v = E[thread_id].v2;
15:  E_H[2 * thread_id + 1].u = U_H[thread_id]]).id;
16: end if
17: EndProcedure
18:
19: Procedure EncontraArestasLeves
20: // thread_id é a identificação de cada thread durante a execução da função kernel
21: // mBG é o número de arestas do grafo bipartido
22: // lightest_edge[v] armazena a identificação da aresta mais leve para o vértice v
23: if (thread_id < mBG) then
24:   v = E_H[thread_id].v;
25: // Begin atomic function
26:   if (E_H[lightest_edge[v]].u > E_H[thread_id].u) then
27:     lightest_edge[v] = thread_id
28:   end if
29: // End atomic function
30: end if
31: EndProcedure
32:
33: Procedure ObtemEsteioANDCalculaNumzerodif
34: // thread_id é a identificação de cada thread durante a execução da função kernel
35: // nGB é o número de vértices V do grafo bipartido
36: if (thread_id < nBG) then
37:   v = V_H[thread_id];
38:   S[thread_id].v = v;
39:   S[thread_id].u = E_H[lightest_edge[v]].u;
40:   d_S[E_H[lightest_edge[v]].u] + +;
41:   if (d_S[E_H[lightest_edge[v]].u] == 2) then
42:     numzerodif + +
43:   end if
44: end if
45: EndProcedure
46:
47: Procedure AtualizaConjuntoSolucao
48: // thread_id é a identificação de cada thread durante a execução da função kernel
49: // mBG é o número de arestas do grafo bipartido
50:
51: if (thread_id < (mBG/2)) then
52:   if (d_S[thread_id] ≥ 1) then
53:   // Begin atomic function
54:     ConjuntoSolucao[TamSolucao] = U_H[thread_id].id
55:     TamSolucao + +
56:   // End atomic function
57:   end if
58: end if
```

3.4 Algoritmo paralelo proposto para computar a árvore geradora mínima (MST)

O algoritmo paralelo proposto usado para calcular uma árvore geradora mínima é quase o mesmo apresentado no Algoritmo 2 para calcular a árvore geradora. A diferença entre eles consiste na maneira como as arestas são selecionadas para compor o esteio, pois para calcular a MST deve-se usar os pesos das mesmas. O Algoritmo 4 esclarece essa distinção detalhando os Procedimentos $CriaGrafoBipartido$ e $EncontraArestasLeves$. Todas as outras etapas do Algoritmo 2 são as mesmas para o cômputo da árvore geradora mínima.

Algoritmo 4 Algoritmo para MST com grafo bipartido - Descrição de Procedimentos

```
1: Procedure CriaGrafoBipartido
2: // thread_id é a identificação de cada thread durante a execução da função kernel
3: // nOG é o número de vértices do grafo original
4: // mOG é o número de arestas do grafo original
5: // considere cada aresta do grafo original representada pelos vértices v1 e v2 e peso w
6: // considere cada aresta do grafo bipartido representada pelos vértices v e u e peso w
7: if (thread_id < mOG) then
8:    if (thread_id < nOG) then
9:       V_H[thread_id].id = V[thread_id].id;
10:   end if
11:   U_H[thread_id].id = thread_id;
12:   E_H[2 * thread_id].v = E[thread_id].v1;
13:   E_H[2 * thread_id].w = E[thread_id].w;
14:   E_H[2 * thread_id].u = U_H[thread_id]]).id;
15:   E_H[2 * thread_id + 1].v = E[thread_id].v2;
16:   E_H[2 * thread_id + 1].w = E[thread_id].w;
17:   E_H[2 * thread_id + 1].u = U_H[thread_id]]).id;
18: end if
19: EndProcedure
20:
21: Procedure EncontraArestasLeves
22: // thread_id é a identificação de cada thread durante a execução da função kernel
23: // mBG é o número de arestas do grafo bipartido
24: // lightest_edge[v] armazena a identificação da aresta mais leve para o vértice v
25: if (thread_id < mBG) then
26:   v = E_H[thread_id].v;
27: // Begin atomic function
28:   if (E_H[lightest_edge[v]].w > E_H[thread_id].w) OR
29:    ((E_H[lightest_edge[v]].w == E_H[thread_id].w) AND (E_H[lightest_edge[v]].u > E_H[thread_id].u)) then
30:      lightest_edge[v] = thread_id
31:   end if
32: // End atomic function
33: end if
34: EndProcedure
```

3.5 Correção dos algoritmos

Nesta seção, a correção dos algoritmos propostos é abordada. Como observado anteriormente, será assumido, sem perda de generalidade, que todos os pesos das arestas $e \in E$ do grafo de entrada $G = (V, E)$ são diferentes.

Lema 1. *Considere um grafo conexo $G = (V, E)$. Seja $H = (V \cup U, E')$ o grafo bipartido correspondente gerado através do Procedimento $CriaGrafoBipartido$ e S o esteio obtido no Procedimento $ObtemEsteioANDCalculaNumzerodif$, com as arestas marcadas pelo Procedimento $EncontraArestasLeves$, todos do Algoritmo 3. Seja G' o grafo obtido adicionando as arestas associadas aos vértices u de U (isto é, $aresta\_original(u)$) tal que*

$d_S(u) \geq 1$ (linha 54 do Algoritmo 3). Então G' é acíclico e o número de componentes conexas de G', representado por $c(G')$, é igual o número de vértices zero-diferença de S, representados pelo conjunto Z, ou seja, $c(G') = |Z|$, onde $Z = \{u \in U$, tal que, $d_H(u) - d_S(u) = 0$, ou seja, $d_S(u) = 2\}$.

Demonstração. Sejam $C_1, C_2, ..., C_k$ as componentes conexas de G'. Seja C_i, uma componente conexa qualquer de G'. Pela construção de G', cada aresta de C_i está associada a um vértice $u \in U$ do esteio S (isto é, $d_S(u) \geq 1$). Tome u_x como sendo o vértice de U com menor índice (ou custo) dentre os associados a arestas de C_i. Logo, pela construção de S e de G', $d_S(u_x) = 2$, isto é, u_x é um vértice *zero-diferença*. Sendo assim, temos que C_i tem no máximo $|V(C_i)| - 1$ arestas. Como C_i é conexa, segue que $|E(C_i)| = |V(C_i)| - 1$ e assim C_i é acíclico.

Como essa observação vale para cada componente conexa, temos que $k = |Z|$, isto é, uma componente C_i não pode ter 2 vértices *zero-diferença*, caso contrário, C_i não seria conexa. □

Corolário 1. *Considere um grafo conexo $G = (V, E)$. Seja $H = (V \cup U, E')$ o grafo bipartido correspondente, S o esteio obtido e G' o grafo obtido pelas arestas originais de G associadas às arestas do esteio S. Seja Z os vértices zero-diferença de S. Se $|Z| = 1$ então G' é uma árvore geradora de G.*

Corolário 2. *Considere um grafo conexo $G = (V, E)$. Seja $H = (V \cup U, E')$ o grafo bipartido correspondente, S o esteio obtido e G' o grafo obtido pelas arestas originais de G associadas às arestas do esteio S. Sejam $C_1, C_2, ..., C_k$ as k componentes conexas de G' e $H' = (V_{H'} \cup U_{H'}, E')$ o grafo resultante da compactação de H (descritas nas linhas 20 a 25 do Algoritmo 2), então o número de vértices em $V_{H'}$ é igual a k, pois cada vértice de $V_{H'}$ corresponde à contração de uma componente conexa de G'.*

Observe que, se S tiver mais de um vértice *zero-diferença*, o grafo G' terá menos de $|V| - 1$ arestas e não será uma árvore geradora. Assim, o algoritmo precisará de mais iterações para completar o grafo G'.

Antes de apresentar a prova do teorema a seguir, algumas definições serão apresentadas. Um corte $(W, V \backslash W)$ de um grafo $G = (V, E)$, com $W \subseteq V$, é uma partição de V. Uma aresta de E *cruza* o corte $(W, V \backslash W)$ se uma de suas extremidade estiver em W e a outra estiver em $V \backslash W$. Seja T uma árvore geradora de G. Por consequência, a remoção de qualquer aresta $e \in T$ resultará nos componentes $(W, V \backslash W)$, onde uma extremidade de e está em W e o outro em $V \backslash W$.

Teorema 1. *Considere um grafo conexo $G = (V, E)$. Seja $H = (V \cup U, E')$ o grafo bipartido correspondente, gerado através do Procedimento $CriaGrafoBipartido$ do Algoritmo 4. Seja S o esteio obtido no Procedimento $ObtemEsteioANDCalculaNumzerodif$ do Algoritmo 3, executado após o Procedimento $EncontraArestasLeves$ do Algoritmo 4, utilizado escolher as arestas mais leves. Seja G' o grafo obtido adicionando as arestas associadas aos vértices u de U (ou seja, $aresta\_original(u)$) tal que $d_S(u) \geq 1$ (linha 54 do Algoritmo 3). Se S contiver exatamente um vértice zero-diferença, então G' é uma árvore geradora* **mínima** *de G.*

Demonstração. Pelo Corolário 1 sabe-se que G' é uma árvore geradora de G. Considere um vértice $v_i \in V$ e a aresta $(v_i, v_j, w) \in E$ tal que o peso de (v_i, v_j, w) seja o menor entre todas as arestas incidentes a v_i. Na linha 54 do Algoritmo 3, a aresta (v_i, v_j, w) é adicionada ao conjunto de arestas da árvore geradora resultante. Considere o corte $(\{v_i\}, V \backslash \{v_i\})$. Suponha, por contradição, que a aresta (v_i, v_j, w) não faz parte da árvore geradora mínima. Então, há outra aresta $(v_i, v_k, z) \in E$, entre as arestas que cruzam o corte, que conecta v_i à árvore geradora mínima. No entanto, o peso da aresta (v_i, v_k, z) é maior que o da aresta (v_i, v_j, w). Portanto, se removermos a aresta (v_i, v_k, z) e adicionarmos a aresta (v_i, v_j, w), o peso total das arestas seria menor. Isso é uma contradição. Portanto, concluí-se que G' é uma árvore geradora mínima de G. □

Teorema 2. *O número de vértices zero-diferença é pelo menos dividido por 2 em cada iteração.*

Demonstração. Sejam V_H e U_H as partições do grafo H imediatamente antes da compactação e sejam $V_{H'}$ e $U_{H'}$ as partições logo após a linha 25 do Algoritmo 2. Seja k o número de vértices *zero-diferença* em U_H. Pelo Corolário 2, o número de vértices em $V_{H'}$ também é k.

Visto que cada vértice *zero-diferença* em $U_{H'}$ tem grau 2, o número de vértices *zero-diferença* $U_{H'}$ é no máximo $|V_{H'}|/2$, ou seja, $k/2$. □

Após $O(\log p)$ rodadas, o número de vértices *zero-diferença* será n/p, com $O(m/p)$ arestas. Então, podemos mover o grafo bipartido compactado restante para a CPU e terminar o algoritmo. Visto que o número de vértices do grafo H é, no mínimo, reduzido a metade na iteração seguinte, ou seja $|V_{H'}| \leq |V_H|/2$, temos $O(\log n)$ iterações necessárias para a convergência do algoritmo, no pior do caso. Na prática, como mostrado em nossos experimentos, o número de iterações é bem menor, em torno de $\frac{\log n}{2}$, como podemos ver na segunda coluna da Tabela 3.6.

3.6 Análise experimental

Foram apresentados dois algoritmos paralelos, um para calcular a árvore geradora e outro para calcular a árvore geradora mínima. Como o cálculo da árvore geradora mínima com todas as arestas com peso igual a um é igual a árvore geradora de um grafo não ponderado, testamos apenas o algoritmo de árvore geradora mínima.

Para mostrar a eficiência das soluções propostas neste capítulo, foram implementadas duas versões do Algoritmo 4 que encontra a árvore geradora mínima (MST). Desenvolveu-se uma versão sequencial usando ANSI C e uma versão paralela, para GPGPU, usando CUDA. Ambas as implementações estão disponíveis para download em https://github.com/jucele/MinimumSpanningTree. Vale ressaltar que a implementação paralela é uma implementação CUDA simples, sem explorar os vários recursos de alto desempenho disponíveis para as arquiteturas GPGPU. O principal objetivo foi apresentar um algoritmo eficiente no modelo BSP/CGM que pudesse ser facilmente implementado em uma máquina paralela real.

A versão CUDA implementa as etapas do algoritmo usando 12 funções *kernel*. Uma função para criar o grafo bipartido imediatamente depois de ler o grafo de entrada e transferir os dados para a GPU. São utilizadas duas funções para encontrar a menor aresta para cada vértice. Duas funções são usadas para obter o esteio. Outra função foi implementada para calcular os componentes conexos, onde usamos a proposta apresentada em [18]. E seis funções têm a responsabilidade de compactar o grafo, marcando os itens a serem removidos e criando o novo grafo bipartido a ser usado na próxima iteração. Em todos os testes descritos neste capítulo foram utilizados blocos de *threads* de tamanho 64.

A implementação usada nos testes experimentais apresentados em [63] tem uma função para otimizar a compactação como exemplificado na Figura 3.6. Como esta função usa uma estrutura de dados específica, consumindo mais espaço de memória, optou-se por suprimi-la para permitir a realização de testes com grafos de entrada maiores. Portanto, o desempenho apresentado neste capítulo é diferente dos resultados mostrados em [63].

Os resultados obtidos com a implementação do algoritmo apresentado foram comparados com os resultados de um algoritmo eficiente publicado recentemente chamado *Edge Pruned Minimum Spanning Tree* (EPMST) [33]. O EPMST escolhe um subconjunto de arestas usando amostragem aleatória. Ele usa a ideia do algoritmo de Kruskal [29] no pequeno subconjunto de arestas e, se necessário, o algoritmo de Prim [52]. A implementação do EPMST está disponível para download no GitHub, o que facilitou a comparação dos resultados. Como o EPMST é uma solução sequencial, uma versão para CPU e outra paralela do algoritmo MST proposto foram desenvolvidas. Vale ressaltar que os conjuntos de dados de entrada usados para realizar os testes para este artigo são diferentes daqueles usados em [33], possivelmente produzindo resultados diferentes.

3.6.1 Grafos de entrada utilizados

Foram utilizados dois conjuntos de grafos de entrada para os testes realizados. O primeiro conjunto de grafo usado foi gerado usando um gerador de grafos aleatórios [21], com implementação disponível em http://condor.depaul.edu/rjohnson/source/graph_ge.c. Foram gerados 27 grafos conexos aleatórios. Os grafos de entrada denominados *graph10a, graph10b, graph10c, graph10d* e *graph10e* têm 10.000 vértices e densidade 0,02, 0,05, 0,1, 0,15 e 0,2, respectivamente. Os grafos identificados com *graph20, graph25* e *graph30* têm características semelhantes, mas com 20.000, 25.000 e 30.000 vértices, respectivamente. Geramos oito grafos com 15.000 vértices (*graph15*) com densidades de 0,02, 0,05, 0,1, 0,15, 0,2, 0,5, 0,75 e 1 .A Tabela 3.4 apresenta as principais características dos grafos, onde n é o número de vértices e m o número de arestas.

Os grafos das redes rodoviárias dos Estados Unidos da América (EUA) compõem o segundo conjunto de grafos de entrada. Estes grafos foram disponibilizados no Nono Desafio de Implementação DIMACS (*Center for Discrete Mathematics and Theoretical Computer Science* (9DIMACS) [6]. O 9DIMACS disponibiliza doze grafos de redes rodoviárias nos Estados Unidos. Visto que as implementações desenvolvidas trabalham 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 3.5 mostra as

Tabela 3.4: Características básicas dos grafos de entrada gerados artificialmente.

Grafo de entrada	n (núm. de vértices)	m (núm. de arestas)	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

informações desse conjunto de grafos. Uma diferença significativa entre os dois conjuntos é a densidade dos mesmos. No segundo conjunto, as densidades são muito menores.

Tabela 3.5: Características básicas dos grafos do nono desafio DIMACS, considerando os grafos não dirigidos e eliminando as arestas duplicadas.

Grafo de entrada	n (núm. de vértices)	m (núm. de arestas)	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

3.6.2 Resultados dos testes realizados

Para cada grafo de entrada, as implementações sequenciais e CUDA foram executas 20 vezes e os tempos de execução foram coletados. As médias dos tempos de execução foram utilizadas para analisar o comportamento experimental do algoritmo. Para o algoritmo EPMST também foram realizadas 20 execuções e utilizado o menor tempo de execução obtido. Os tempos não levam em conta a leitura do dados de entrada.

A Tabela 3.6 apresenta os resultados dos testes obtidos para os dois conjuntos de entrada usando o Ambiente 5 (M4000 - UFMS), descrito na Seção 2.2. Cada linha da tabela mostra, para cada grafo de entrada, o número de iterações do algoritmo, o tempo de execução da implementação para CPU, o tempo de execução da implementação CUDA, o tempo de execução do EPMST, o *speeup* da implementação para CPU do algoritmo proposto em comparação com a implementação do EPMST, o *speedup* da implementação CUDA do algoritmo proposto em comparação com a implementação do EPMST e o *speedup* da implementação CUDA em comparação com a implementação para CPU do algoritmo proposto. Pelo Teorema 2 sabemos que o algoritmo precisa de $\log n$ iterações no pior caso; no entanto, é possível observar que, na prática, esse número é muito menor (veja a segunda coluna da Tabela 3.6). Da mesma forma, a Tabela 3.7 e a Tabela 3.8 apresentam os resultados de testes obtidos usando, respectivamente, o Ambiente 6 (K40M - UFG) e Ambiente 4 (GTX745 - UFMS), descritos na Seção 2.2.

O objetivo foi projetar um algoritmo paralelo que seja eficiente no modelo BSP/CGM ($O(\log p)$ rodadas de computação/comunicação, onde p é o número de processadores) e adequado para ser executado em ambientes paralelos reais. As implementações desenvolvidas usaram somente recursos padrão da linguagem C e da biblioteca CUDA. Além disso, descobriu-se que o algoritmo tem maior *s*peedup quando temos um número significativo de arestas, já que a implementação da árvore geradora mínima usa mais de 40% do tempo de execução na criação do grafo bipartido. Se o grafo não tiver pelo menos 3.000.000 arestas, aproximadamente, a árvore geradora será calculada muito rapidamente e o tempo gasto na criação do grafo bipartido dominará o tempo total. Testou-se a implementação em diferentes ambientes de GPUs. O comportamento do algoritmo nos três ambientes é muito similar usando GPUs com diferentes capacidades computacionais.

A Figura 3.8 ilustra o tempo de execução das implementações testadas para grafos com 15.000 vértices do primeiro conjunto de grafos de entrada no Ambiente 5 (M4000 - UFMS). É possível observar que, conforme a densidade aumenta, o desempenho do algoritmo proposto melhora. Para a densidade 0,02, as implementações do algoritmo proposto têm resultados piores do que o EPMST, mas acima da densidade 0,05 a implementação CUDA já supera o tempo do EPMST. A implementação para CPU do algoritmo é melhor que o EPMST a partir da densidade 0,1. O *speedup* apresentado pela implementação CUDA em comparação com o EPMST aumenta substancialmente à medida que a densidade do grafo aumenta. O *speedup* seria melhor se a estratégia de compactação presente na implementação descrita em [63] fosse utilizada. No entanto, devido ao tamanho da memória das máquinas disponíveis para teste, a função de compactação não foi usada.

Analisando a Figura 3.9, que apresenta os tempos de execução para os grafos com densidade de 0,02 e 0,1, usando o Ambiente 5 (M4000 - UFMS), é possível observar que

Tabela 3.6: Resultados dos testes, em segundos, usando o Ambiente 5 (M4000 - UFMS).

Grafo de Entrada	**num. iterações**	**Tempo CPU** (seg.)	**Tempo CUDA** (seg.)	**Tempo EPMST** (seg.)	**Speedup CPU x EPMST**	**Speedup CUDA x EPMST**	**Speedup CUDA x CPU**
graph10a	5	0,858	0,763	0,068	0,079	0,089	1,125
graph10b	4	1,852	0,827	0,260	0,140	0,315	2,241
graph10c	3	2,568	0,910	1,301	0,507	1,429	2,821
graph10d	4	4,445	1,100	3,908	0,879	3,554	4,042
graph10e	3	4,686	1,117	9,517	2,031	8,520	4,195
graph15a	5	2,397	0,863	0,414	0,173	0,479	2,777
graph15b	5	4,676	1,065	1,262	0,270	1,185	4,392
graph15c	3	5,884	1,189	6,807	1,157	5,723	4,947
graph15d	3	8,322	1,406	19,812	2,381	14,092	5,919
graph15e	3	10,524	1,649	48,508	4,609	29,412	6,381
graph15f	2	15,070	2,366	92,278	6,123	38,993	6,368
graph15g	2	22,122	3,228	205,924	9,309	63,798	6,854
graph20a	5	3,993	0,967	0,527	0,132	0,545	4,131
graph20b	4	7,458	1,276	4,057	0,544	3,180	5,846
graph20c	4	12,533	1,788	20,737	1,655	11,600	7,010
graph20d	3	14,474	1,955	61,792	4,269	31,601	7,402
graph20e	3	18,050	2,298	153,033	8,478	66,593	7,854
graph25a	5	5,950	1,114	1,257	0,211	1,128	5,339
graph25b	4	10,845	1,565	9,700	0,894	6,197	6,929
graph25c	4	20,116	2,454	52,785	2,624	21,513	8,199
graph25d	2	13,251	2,007	151,694	11,447	75,589	6,603
graph25e	3	28,272	3,177	414,316	14,655	130,431	8,900
graph30a	6	10,176	1,459	2,657	0,261	1,821	6,973
graph30b	5	18,385	2,218	20,321	1,105	9,163	8,289
graph30c	3	22,840	2,619	104,559	4,578	39,921	8,721
graph30d	3	33,143	3,536	332,890	10,044	94,154	9,374
graph30e	3	40,840	4,448	988,326	24,200	222,193	9,182
USA-road-d.NY	9	0,158	0,730	0,082	0,515	0,112	0,217
USA-road-d.BAY	10	0,167	0,729	0,080	0,480	0,110	0,229
USA-road-d.COL	9	0,209	0,742	0,084	0,403	0,114	0,282
USA-road-d.FLA	10	0,540	0,809	0,520	0,964	0,643	0,667
USA-road-d.NW	10	0,559	0,821	0,332	0,593	0,404	0,681
USA-road-d.NE	10	0,819	0,856	0,760	0,927	0,888	0,957
USA-road-d.CAL	11	0,984	0,892	0,955	0,970	1,071	1,104
USA-road-d.LKS	11	1,466	0,988	1,766	1,204	1,787	1,484
USA-road-d.E	11	1,831	1,066	2,584	1,411	2,424	1,718
USA-road-d.W	11	3,233	1,323	5,224	1,616	3,949	2,444
USA-road-d.CTR	12	9,778	2,331	26,405	2,700	11,329	4,195
USA-road-d.USA	12	12,599	3,112	71,107	5,644	22,848	4,048

o EPMST apresenta melhor desempenho para os grafos de entrada com 10.000, 15.000 e 20.000 vértices, mas para grafos maiores, com 25.000 e 30.000 vértices, a implementação CUDA proposta leva menos tempo para executar. Para os grafos de entrada com densidade de 0,1, os tempos de execução das implementações propostas são melhores que os tempos de EPMST em quase todos os testes. Como podemos observar nos dados apresentados na Tabela 3.6, Tabela 3.7 e Tabela 3.8, para cerca de metade dos grafos do 9DIMACS a implementação CUDA foi mais lenta que a solução EPMST. Como os grafos do 9DIMACS são muito esparsos e o algoritmo EPMST usa uma heurística, a árvore geradora mínima é quase encontrada após a aplicação desta heurística, já que em várias situações, há apenas uma estrada ligando duas cidades. Nosso algoritmo tem uma etapa que consome muito tempo (criação do grafo bipartido), mas quando comparamos com grandes grafos, o tempo usado na criação do grafo bipartido é compensado.

A Figura 3.10 apresenta os tempos obtidos pela implementação CUDA do algoritmo proposto usando como entrada os grafos do 9DIMACS em três ambientes de teste: Am-

Tabela 3.7: Resultados dos testes, em segundos, usando o Ambiente 6 (K40M - UFG).

Grafo de Entrada	**num. iterações**	**Tempo CPU** (seg.)	**Tempo CUDA** (seg.)	**Tempo EPMST** (seg.)	**Speedup CPU x EPMST**	**Speedup CUDA x EPMST**	**Speedup CUDA x CPU**
graph10a	5	1,084	0,898	0,083	0,077	0,093	1,207
graph10b	4	2,494	1,036	0,317	0,127	0,306	2,407
graph10c	3	3,601	1,163	1,607	0,446	1,382	3,097
graph10d	4	7,104	1,476	4,814	0,678	3,263	4,814
graph10e	3	7,916	1,540	11,754	1,485	7,634	5,141
graph15a	5	4,025	1,061	0,507	0,126	0,478	3,795
graph15b	5	8,153	1,375	1,558	0,191	1,133	5,929
graph15c	3	10,323	1,644	8,407	0,814	5,114	6,279
graph15d	3	14,497	2,061	24,510	1,691	11,890	7,033
graph15e	3	18,024	2,467	59,959	3,327	24,303	7,305
graph15f	2	24,700	3,866	113,986	4,615	29,481	6,388
graph15g	2	35,827	5,430	255,183	7,123	46,993	6,598
graph20a	5	7,227	1,220	0,651	0,090	0,534	5,922
graph20b	4	13,800	1,733	5,009	0,363	2,890	7,962
graph20c	4	21,801	2,718	25,658	1,177	9,441	8,022
graph20d	3	25,414	3,029	76,408	3,007	25,226	8,390
graph20e	3	30,611	3,598	189,168	6,180	52,582	8,509
graph25a	5	10,854	1,521	1,557	0,143	1,023	7,134
graph25b	4	19,792	2,289	11,994	0,606	5,240	8,648
graph25c	4	35,413	3,873	65,263	1,843	16,849	9,143
graph25d	2	22,130	3,389	187,733	8,483	55,393	6,530
graph25e	3	48,612	5,330	470,622	9,681	88,294	9,120
graph30a	6	19,145	2,062	3,282	0,171	1,592	9,286
graph30b	5	33,556	3,460	25,107	0,748	7,256	9,698
graph30c	3	41,477	4,433	129,282	3,117	29,162	9,356
graph30d	3	59,575	6,167	403,514	6,773	65,431	9,660
graph30e	3	70,957	7,528	1095,140	15,434	145,484	9,426
USA-road-d.NY	9	0,185	0,924	0,105	0,569	0,114	0,201
USA-road-d.BAY	10	0,196	0,950	0,097	0,495	0,102	0,207
USA-road-d.COL	9	0,270	0,965	0,114	0,421	0,118	0,280
USA-road-d.FLA	10	0,762	1,027	0,720	0,945	0,701	0,742
USA-road-d.NW	10	0,800	1,062	0,491	0,613	0,462	0,753
USA-road-d.NE	10	1,197	1,118	1,062	0,887	0,950	1,071
USA-road-d.CAL	11	1,451	1,210	1,322	0,911	1,093	1,200
USA-road-d.LKS	11	2,220	1,369	2,712	1,221	1,980	1,621
USA-road-d.E	11	2,800	1,504	3,902	1,393	2,594	1,862
USA-road-d.W	11	5,010	1,981	8,133	1,623	4,104	2,528
USA-road-d.CTR	12	16,590	3,773	39,092	2,356	10,362	4,398
USA-road-d.USA	12	19,476	5,493	99,591	5,114	18,131	3,546

biente 4 (GTX745 - UFMS), Ambiente 5 (M4000 - UFMS) e Ambiente 6 (K40M - UFG). Notamos que o Ambiente 6 apresentou os piores tempos para todos os grafos do conjunto de testes utilizado. O Ambiente 5 passou a apresentar vantagem no caso dos grafos maiores, a partir do grafo "USA-road-d.E ". Seria necessário uma análise detalhada do comportamento do algoritmo e das características específicas das placas gráficas utilizadas para gerar uma relação a esse respeito.

O artigo [35], publicado recentemente, propõe uma solução eficiente usando GPU baseada em transação do algoritmo de Borůvka. Um dos ambientes de teste usados pelos autores tem como base a GPU NVIDIA Tesla K40, semelhante ao Ambiente 6 (K40M - UFG), descrito na Seção 2.2. A tabela 3.9 apresenta a comparação das implementações do algoritmo proposto neste capítulo e os resultados disponíveis em [35]. Como é possível observar, a implementação proposta neste capítulo apresenta melhores resultados para grafos de entrada maiores.

Tabela 3.8: Resultados dos testes, em segundos, usando o Ambiente 4 (GTX745 - UFMS).

Grafo de Entrada	**num. iterações**	**Tempo CPU** (seg.)	**Tempo CUDA** (seg.)	**Tempo EPMST** (seg.)	**Speedup CPU x EPMST**	**Speedup CUDA x EPMST**	**Speedup CUDA x CPU**
graph10a	5	0,778	0,315	0,061	0,079	0,195	2,470
graph10b	4	1,617	0,440	0,234	0,145	0,532	3,680
graph10c	3	2,207	0,707	2,272	0,531	1,657	3,121
graph10d	4	3,799	1,165	3,521	0,927	3,023	3,262
graph10e	3	3,995	1,100	8,575	2,147	7,793	3,630
graph15a	5	2,099	0,554	0,192	0,091	0,346	3,788
graph15b	5	4,091	1,050	1,136	0,278	1,081	3,894
graph15c	3	4,994	1,243	6,133	1,228	4,934	4,018
graph15d	3	7,070	1,927	17,855	2,525	9,267	3,670
graph15e	3	9,723	2,578	43,677	4,492	16,945	3,772
graph15f	2	17,001	memória insuficiente	83,129	4,890	-	-
graph15g	2	25,282	memória insuficiente	196,735	7,782	-	-
graph20a	5	3,499	0,931	0,475	0,136	0,510	3,757
graph20b	4	6,453	1,466	3,654	0,566	2,492	4,401
graph20c	4	10,752	3,269	18,691	1,738	5,718	3,289
graph20d	3	12,076	3,984	55,723	4,615	13,987	3,031
graph20e	3	15,631	4,938	139,219	8,907	28,191	3,165
graph25a	5	5,051	1,206	1,132	0,224	0,939	4,188
graph25b	4	9,177	2,660	8,742	0,953	3,286	3,450
graph25c	4	16,969	5,377	47,599	2,805	8,852	3,156
graph25d	2	11,345	4,678	137,808	12,147	29,461	2,425
graph25e	3	29,636	memória insuficiente	382,659	12,912	-	-
graph30a	6	8,845	1,811	2,392	0,270	1,320	4,883
graph30b	5	15,884	4,477	18,309	1,153	4,177	3,572
graph30c	3	23,203	5,884	94,278	4,063	16,022	3,943
graph30d	3	36,268	memória insuficiente	316,135	8,717	-	-
graph30e	3	45,430	memória insuficiente	839,328	18,475	-	-
USA-road-d.NY	9	0,132	0,443	0,067	0,511	0,152	0,298
USA-road-d.BAY	10	0,138	0,441	0,069	0,498	0,156	0,314
USA-road-d.COL	9	0,180	0,459	0,076	0,424	0,166	0,391
USA-road-d.FLA	10	0,474	0,578	0,439	0,927	0,760	0,819
USA-road-d.NW	10	0,494	0,578	0,302	0,611	0,523	0,856
USA-road-d.NE	10	0,724	0,675	0,677	0,934	1,002	1,073
USA-road-d.CAL	11	0,876	0,735	0,809	0,923	1,100	1,191
USA-road-d.LKS	11	1,306	0,927	1,523	1,166	1,643	1,409
USA-road-d.E	11	1,631	1,096	2,226	1,365	2,030	1,488
USA-road-d.W	11	2,877	1,573	4,519	1,571	2,873	1,829
USA-road-d.CTR	12	8,835	3,452	23,815	2,695	6,899	2,560
USA-road-d.USA	12	12,687	4,554	63,047	4,969	13,845	2,786

Tabela 3.9: Comparando alguns resultados obtidos, em segundos, usando o Ambiente 6 com os dados disponíveis em [35].

Grafo de Entrada	**Implementação de Manoochehri et al. (2017) usando GPU TeslaK40**	**Implementação CUDA usando Ambiente 6 (K40M - UFMS)**	**Speedup**
USA-road-d.NY	0,217	0,924	0,235
USA-road-d.FLA	0,736	1,027	0,717
USA-road-d.E	1,959	1,504	1,302
USA-road-d.W	3,451	1,981	1,742
USA-road-d.USA	13,407	5,493	2,441

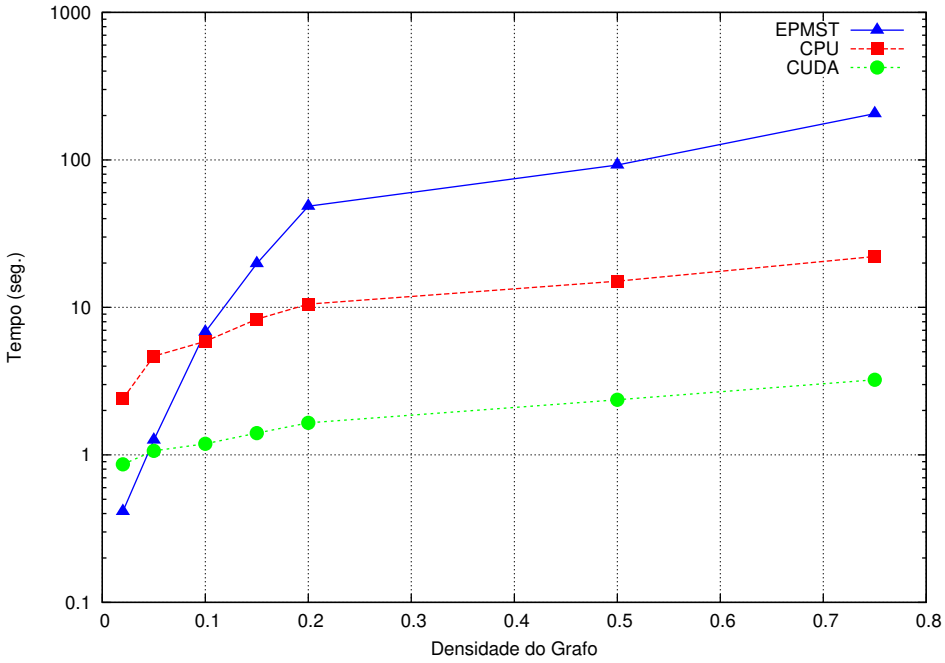


Figura 3.8: Tempos de execução para grafos com 15.000 vértices no Ambiente 5 (M4000 - UFMS).

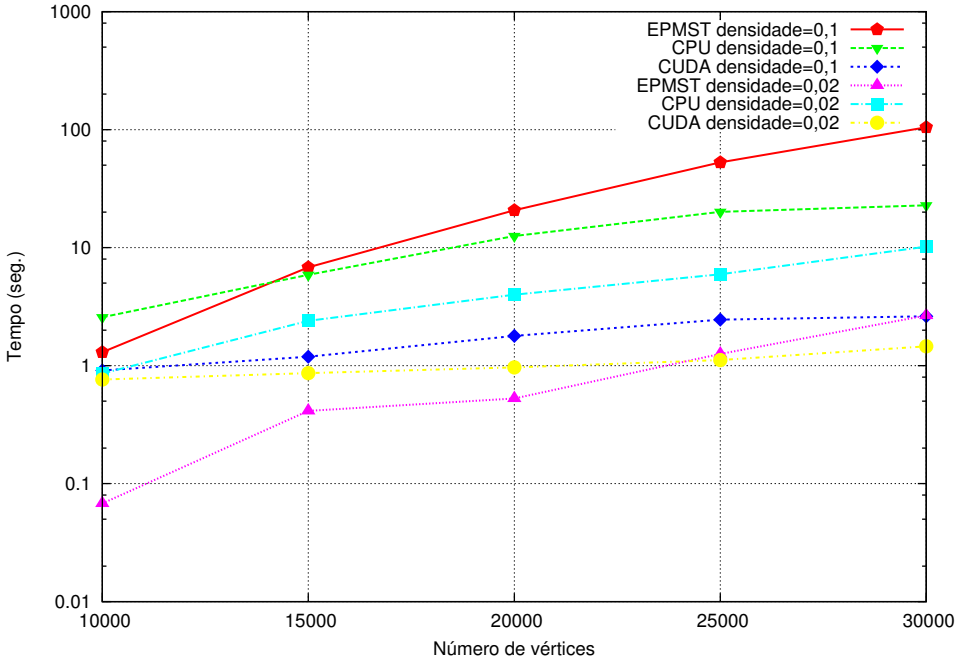


Figura 3.9: Tempos de execução para grafos com densidade de 0,02 e 0,1 no Ambiente 5 (M4000 - UFMS).

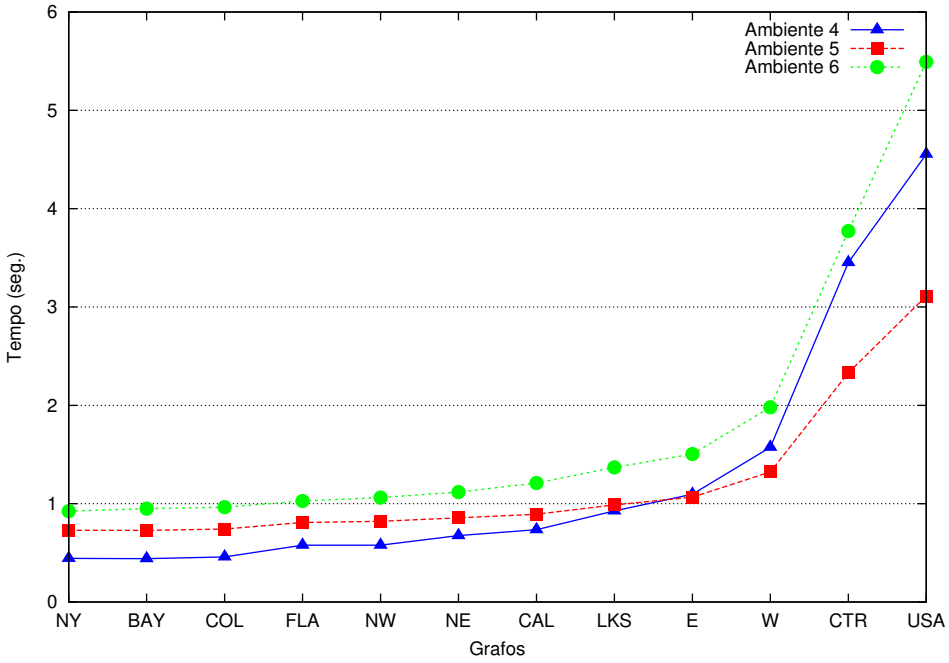


Figura 3.10: Comparação dos tempos de execução da implementação CUDA em três ambientes diferentes.

3.7 Contribuições

Dentre as contribuições alcançadas com o trabalho descrito neste capítulo tivemos:

- algoritmos paralelos, usando o modelo BSP/CGM, para o problema da árvore geradora e árvore geradora mínima com desempenho eficiente;
- um artigo publicado nos anais do *Oitavo Workshop on Applications for Multi-Core Architectures* (WAMCA2017), com título *"A parallel algorithm for minimum spanning tree on GPU"* [63], que recebeu o prêmio de melhor artigo do workshop; e
- uma versão extendida do artigo do WAMCA2017 aceita para publicação no periódico *"The International Journal of High Performance Computing Applications"*, com o título *"New BSP/CGM algorithms for spanning trees"* [65].

Capítulo 4

Algoritmo Paralelo para Árvore Geradora sem Grafo Bipartido

4.1 Introdução

Este capítulo apresenta uma evolução da abordagem apresentada no Capítulo 3. Ao implementar a proposta do capítulo anterior para o problema da árvore geradora notou-se que a etapa de criação do grafo bipartido consumia muito do tempo total necessário para encontrar a árvore geradora (ST). Assim, construiu-se outra proposta, apresentada neste capítulo, que não faz uso do grafo bipartido. A construção do esteio foi alterada para fazer uso das arestas do grafo original e criou-se o conceito de *aresta zero-diferença*.

Como não foram encontradas soluções paralelas para árvore geradora (ST) que possibilitassem a replicação de testes, optou-se por apresentar nos resultados experimentais a comparação de tempos alcançados pelas implementações para CPU e paralela do novo algoritmo.

A Seção 4.2 apresenta o novo algoritmo para encontrar a árvore geradora sem a necessidade do grafo bipartido correspondente ao grafo original. Em seguida, na Seção 4.3, são mostrados mais detalhes da correção e complexidade da solução proposta. Na Seção 4.4 os resultados experimentais são apresentados. Ao final do capítulo tem-se as conclusões e contribuições na Seção 4.5.

4.2 Algoritmo paralelo para o problema da árvore geradora

Para acompanhar a descrição e análise do algoritmo que será apresentado é necessário o conhecimento dos conceitos apresentados na Subseção 3.1.1. O algoritmo paralelo que será apresentado foi projetado usando o modelo BSP/CGM [60, 10]. De forma resumida, 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.

No algoritmo é utilizado o conceito denominado esteio, mas diferente da definição apresentada em [5]. No contexto deste capítulo, 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 4.1 apresenta um grafo G com cinco vértices e oito arestas. O esteio para este grafo é composto por quatro arestas (arestas pontilhadas na figura), onde apenas uma delas é *zero-diferença* (aresta (1, 2)). No caso desse exemplo o esteio gerado na primeira iteração do algoritmo é uma árvore geradora de G.

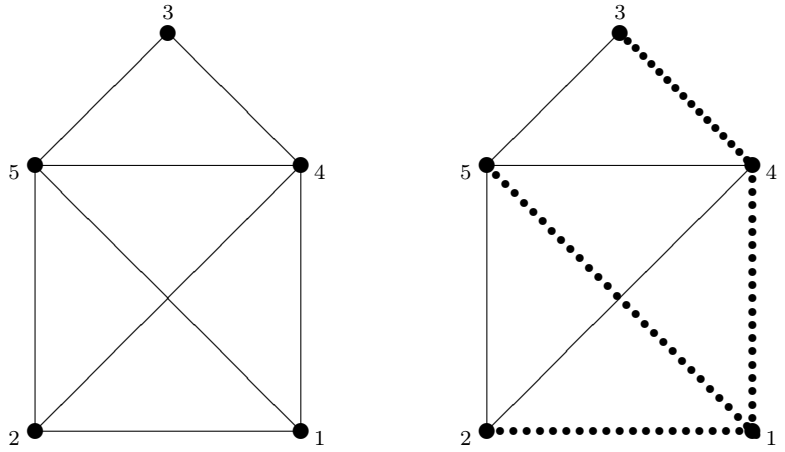


Figura 4.1: Exemplo 1, onde à esquerda é apresentado o grafo $G = (V, E)$, sendo $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente de G, formado pela aresta $(1, 2)$ (escolhida pelos vértices 1 e 2), aresta $(1, 4)$ (escolhida pelo vértice 4), aresta $(1, 5)$ (escolhida pelo vértice 5) e aresta $(3, 4)$ (escolhida pelo vértice 3), sendo $(1, 2)$ a única aresta zero-diferença.

A definição de esteio é a mesma utilizada no capítulo 3, mas a solução apresentada no capítulo anterior trabalha com um grafo bipartido correspondente ao grafo de entrada e aqui trabalha-se diretamente com o grafo original. Lá a definição usada no algoritmo é de *vértice zero-diferença* e aqui usa-se *aresta zero-diferença*.

Algoritmo 5 apresenta a ideia do funcionamento da proposta. Este consiste basicamente em encontrar o esteio do grafo e se necessário, caso a solução não seja uma árvore geradora, compactá-lo para uma nova iteração do algoritmo. A solução consiste de um algoritmo paralelo, ou seja, os passos são executados por diversos processadores encontrando a solução de forma colaborativa.

O número de iterações do algoritmo e a árvore geradora produzida depende da rotulação dos vértices. A Figura 4.2 apresenta o mesmo grafo da Figura 4.1 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 4.2 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, caso existam. Para o exemplo da

Algoritmo 5 Algoritmo para ST sem o grafo bipartido

Entrada: Um grafo conexo $G = (V, E)$, onde $V = \{v_1, v_2, \ldots, v_n\}$ é um conjunto com n vértices e E é um conjunto com m arestas (v_i, v_j), onde v_i e v_j são vértices de V. Cada aresta (v_i, v_j) tem um peso correspondente representado por w_{ij}.
Saída: Uma árvore geradora de G cujas arestas estão em *ConjuntoSolucao*.

1: *ConjuntoSolucao* := vazio.
2: // Encontrando a solução para ST.
3: *numzerodif* = 0
4: $r := 0$
5: **while** $((r < \log p)$ **AND** $(numzerodif \neq 1))$ **do** // p é o número de processadores
6: // Encontrando as menores arestas para cada vértice e armazenando em $menor[v_i]$.
7: **for each** $(v_i \in V)$ **in parallel do**
8: $menor[v_i]$ = -1
9: **end for**
10: **for each** $(v_i \in V)$ **in parallel do**
11: Encontra aresta $menor[v_i]$.
12: **end for**
13: // Obtendo o esteio.
14: **for each** $(e_j \in E)$ **in parallel do**
15: $d_S(e_j) = 0$
16: **end for**
17: **for each** $(v_i \in V)$ **in parallel do**
18: $d_S(menor[v_i]) = d_S(menor[v_i]) + 1$ // função atômica
19: **end for**
20: // Adicionando as arestas do esteio a *ConjuntoSolucao*
21: // e calculando o número de arestas zero-diferença
22: *numzerodif* = 0 // *numzerodif* armazena o número de arestas zero-diferença
23: **for each** $(e_j \in E)$ **in parallel do**
24: **if** $(d_S(e_j) \geq 1)$ **then**
25: Adicione a aresta e_j a *ConjuntoSolucao*
26: **if** $(d_S(e_j) == 2)$ **then**
27: *numzerodif* = *numzerodif* +1 // função atômica
28: **end if**
29: **end if**
30: **end for**
31: **if** $(numzerodif \neq 1)$ **then**
32: // Compactação do grafo G
33: Computar os componentes conexos
34: **for each** $(v_i \in V)$ **in parallel do**
35: Atualizar v_i
36: **end for**
37: **for each** $(e_j \in E)$ **in parallel do**
38: Atualizar $e_j.v_1$
39: Atualizar $e_j.v_2$
40: **if** $(e_j.v_1 == e_j.v_2)$ **then**
41: Eliminar e_j
42: **end if**
43: **end for**
44: **end if**
45: $r := r + 1$
46: **end while**

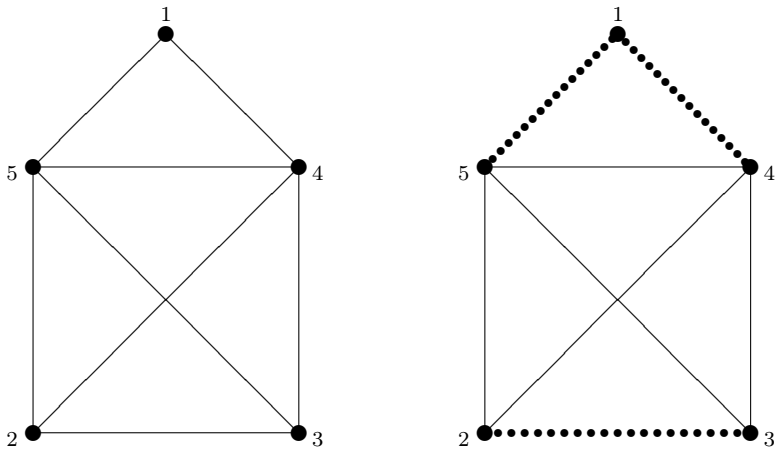


Figura 4.2: Exemplo 2, onde à esquerda é apresentado o grafo $G = (V, E)$, sendo $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente de G, formado pela aresta $(1, 4)$ (escolhida pelos vértices 1 e 4), aresta $(1, 5)$ (escolhida pelo vértice 5) e aresta $(2, 3)$ (escolhida pelos vértices 2 e 3), sendo $(1, 4)$ e $(2, 3)$ arestas zero-diferença.

Figura 4.2 o grafo resultante da compactação terá dois vértices e quatro arestas como pode ser visto na Figura 4.3. Na próxima iteração do algoritmo qualquer uma das quatro arestas que for escolhida forma a árvore geradora. Para garantirmos que a mesma aresta seja escolhida para ambos os vértices, escolhemos a aresta com menor índice no vetor de arestas do grafo, por exemplo a aresta (4,2). Assim, o a árvore geradora para o Exemplo 2 (Figura 4.2) é a árvore ilustrada na Figura 4.4.

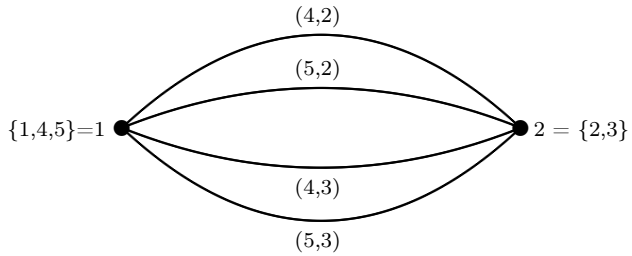


Figura 4.3: Grafo compactado correspondente ao grafo da Figura 4.2.

4.3 Correção e complexidade do algoritmo

Nesta seção, abordamos a correção e complexidade do algoritmo proposto.

Lema 2. *Considere um grafo conexo $G = (V, E)$. Seja G' o grafo obtido pela adição das arestas escolhidas para compor o esteio S (linha 18 do Algoritmo 5). 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 (v_i, v_j) com o menor v_j, o conjunto de arestas selecionadas (v_i, v_j), tal que $v_i > v_j$ não formam um ciclo, por

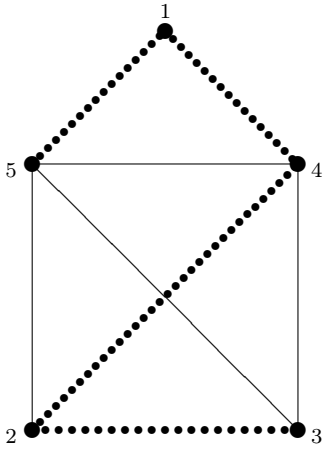


Figura 4.4: Árvore geradora para o grafo da Figura 4.2.

exemplo, para v_1, a aresta escolhida é tal que $v_1 < v_j$. 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 3. *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 4. *A cada iteração o número de componentes conexos computados na linha 33 do Algoritmo 5, que corresponderá ao número de vértices do grafo compactado após a linha 44 do algoritmo é igual ao número de arestas zero-diferença.*

Demonstração. Considere o grafo $G = (V, E)$, sendo $|V| = n$, e $numzerodif$ o número de arestas zero-diferença do esteio S calculado para G. O esteio S tem portanto $(n - numzerodif)$ arestas, visto que cada vértice de V escolhe uma aresta e algumas arestas $(numzerodif)$ são escolhidas duas vezes, resultando em um grafo G' acíclico conforme o Lema 2, sendo que, se $numzerodif = 1$ este grafo corresponde a uma árvore geradora de G. Sendo assim, cada aresta removida de G' desconecta o grafo, gerando mais uma componente conexa, ou seja, se $numzerodif = 1$ temos uma componente conexa, se tirarmos uma aresta de G' ($numzerodif = 2$) teremos duas componentes conexas, e assim por diante. Ou seja, $numzerodif$ é igual o número de componentes conexas do grafo G'. □

Teorema 5. *O algoritmo para a computação da árvore geradora utiliza* $\log p$ *rodadas de comunicação com computação* $O(n/p)$ *computação local.*

Demonstração. Pelo Teorema 3, 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 $O(n/p)$. □

4.4 Resultados experimentais

Considerando que os trabalhos mais recentes sobre árvores geradoras tais como [67, 44, 30, 41], resolvem problemas mais gerais, além de usarem em sua implementação recursos computacionais muito diferentes, o objetivo dos experimentos aqui descritos foi o de verificar se o algoritmo proposto tem um bom desempenho em máquinas paralelas reais de memória compartilhada, e qual o ganho obtido pelo algoritmo com relação a sua versão para CPU. Para efetuar essa comparação, uma versão para CPU do algoritmo foi desenvolvida usando ANSI C, ou seja, apenas instruções sequenciais, que podem ser paralelizadas por otimizações do compilador, mas sem interferência do programador. 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 as menores arestas 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 [18]. Também usamos funções atômicas disponíveis na biblioteca CUDA na implementação de alguns *kernels*. Em todos os testes descritos neste capítulo foram utilizados blocos de *threads* de tamanho 64.

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.

Para analisar a eficiência e escalabilidade do algoritmo utilizamos como entrada os dois conjuntos de grafos descritos na subseção 3.6.1. O primeiro conjunto é composto de grafos construídos artificialmente por meio de um gerador de grafos pseudo-aleatórios. O segundo é composto pelos grafos das redes rodoviárias dos Estados Unidos, disponibilizados no Nono Desafio de Implementação DIMACS.

Para cada um dos grafos de entrada, as implementações para CPU e CUDA foram executadas 20 vezes, sendo usada a média do tempo de execução para analisar o comportamento experimental do algoritmo. Os tempos não levam em conta a leitura do dados de entrada. A Tabela 4.1 apresenta os resultados obtidos pelos testes para o primeiro conjunto de grafos de entrada (Tabela 3.4), onde cada linha mostra o número de iterações do algoritmo, o tempo de execução da implementação para CPU, o tempo de execução da implementação CUDA e o *speedup* da implementação em CUDA relativa a implementação para CPU. Vale salientar que o número de iterações do algoritmo necessárias para encontrar a árvore geradora para esse conjunto de testes não passou de dois. Esses dados foram obtidos utilizando o Ambiente 5 (M4000 - UFMS) descrito na Seção 2.2.

Tabela 4.1: Resultados dos testes para os grafos gerados artificialmente usando o Ambiente 5 (M4000 - UFMS).

Grafo de entrada	Número de Iterações	Tempo CPU (s)	Tempo CUDA (s)	Speedup
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 4.2 mostra os resultados dos testes para o conjunto de grafos do nono desafio DIMACS (tabela 3.5), utilizando o Ambiente 5 (M4000- UFMS). Para essas entradas o *speed-up* da implementação em CUDA em relação à implementação para CPU 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 4.5 ilustra como o tempo da implementação para CPU é pior do que o tempo da implementação paralela utizando CUDA. A Figura 4.6 mostra o crescente *speed-up* da implementação CUDA à 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 das execuções paralelas são bem melhores que das sequenciais. Relembramos que o problema tratado tem um algoritmo sequencial ótimo que é linear para o tamanho da entrada, mas que não possui uma implementação paralela direta.

Tabela 4.2: Resultados dos testes para os grafos do conjunto nono desafio DIMACS usando o Ambiente 5 (M4000 - UFMS).

Grafo de entrada	Número de Iterações	Tempo CPU (s)	Tempo CUDA (s)	Speedup
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

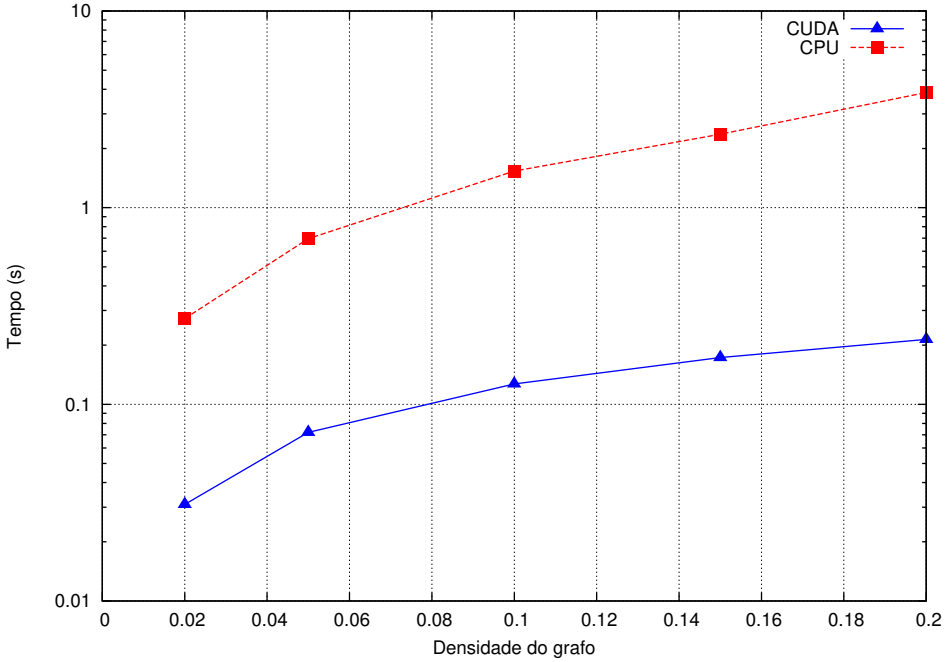


Figura 4.5: Tempo de execução para grafos com 30.000 vértices.

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.

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

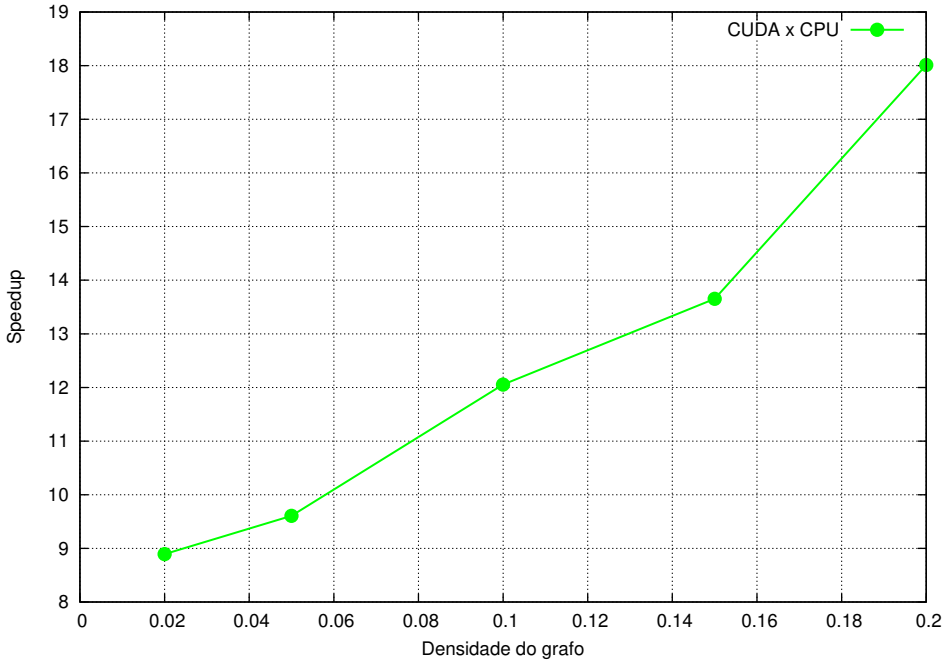


Figura 4.6: *Speedups* obtidos pela implementação CUDA em relação a implementação CPU para grafos com 30.000 vértices.

4.5 Contribuições

Dentre as contribuições alcançadas com a proposta descrita neste capítulo destaca-se:

- um novo algoritmo paralelo e eficiente, usando o modelo BSP/CGM, para o problema da árvore geradora, sem usar o grafo bipartido; e
- um artigo publicado nos anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD2017, com título *"Algoritmo Paralelo para Árvore Geradora Usando GPU"* [62], o qual recebeu menção honrosa por estar entre os três melhores artigos do simpósio.

Capítulo 5

Algoritmo Paralelo para Árvore Geradora Mínima sem Grafo Bipartido

5.1 Introdução

Neste capítulo, propomos outro algoritmo BSP/CGM para o cômputo da árvore geradora mínima de um dado grafo. O algoritmo proposto é eficiente no modelo BSP/CGM e usa $O(\log p)$ rodadas de computação. A solução baseia-se no algoritmo proposto por Cáceres et al. [4], no algoritmo de Borůvka [3] e nas propostas apresentadas em [63] e [62]. Esse novo algoritmo não precisa calcular o grafo bipartido correspondente ao grafo de entrada e a construção do esteio não requer algoritmos de ordenação, sendo assim mais eficiente. Para demonstrar que o novo algoritmo também funciona bem na prática, a implementação foi executada usando três ambientes diferentes baseados em GPU. Os resultados obtidos mostram que o algoritmo é competitivo, com *speedup* de até 21 se comparado a outra proposta paralela publicada recentemente [35], e que o modelo BSP/CGM é adequado para projetar algoritmos paralelos realistas. Mostramos a comparação dos tempos de execução de nossa implementação atual e os resultados apresentados em [35] e [63].

O algoritmo apresentado no Capítulo 3 lê um grafo G, cria um grafo bipartido correspondente H de G, adicionando um novo vértice no meio de cada aresta. Depois disso calcula-se o esteio S de H. Usando algumas propriedades do grafo bipartido H, provou-se que podemos encontrar a árvore geradora mínima de H e a árvore geradora mínima de G. A criação do gráfico bipartido é uma etapa que, paralelamente, percorre todas as arestas do grafo, mas, dependendo do tamanho do grafo e do número de processadores disponíveis, ainda é um passo muito demorado. Além disso, o grafo H ocupa o dobro do espaço de memória do grafo G, uma vez que duplica o número de arestas. O resultado apresentado neste capítulo encontra o esteio diretamente de G. Nesse caso, não é possível usar as propriedades do grafo bipartido H (por exemplo o grau dos novos vértices adicionados), mas criou-se uma maneira de provar que essa ideia também leva à árvore geradora mínima de G.

5.2 O novo algoritmo paralelo para árvore geradora mínima

Os conceitos básicos necessários para entender o algoritmo e sua análise foram apresentados na Subseção 3.1.1. O algoritmo paralelo que será apresentado foi projetado usando o modelo BSP/CGM [60, 10]. O algoritmo utiliza o conceito denominado esteio, mas diferente da definição apresentada em [5]. No contexto deste capítulo, 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, w_{ij}, v_j) tal que w_{ij} é o menor peso dentre as arestas ligadas a v_i e caso exista mais de uma aresta incidente a v_i que possua o mesmo peso w_{ij}, então v_j é o vértice com menor rótulo. Vejamos o exemplo do grafo representado na Figura 5.1. O grafo da figura possui cinco vértices e oito arestas. O esteio para este grafo é composto por quatro arestas, representadas por linhas pontilhadas na figura. Consideremos o grafo da Figura 5.1, onde o vértice 1 possui duas arestas com menor peso igual a 10, a aresta $(1, 10, 2)$ e a aresta $(1, 10, 5)$. Para o vértice 1 a aresta escolhida para compor o esteio é a aresta $(1, 10, 2)$ e não a aresta $(1, 10, 5)$, pois 2 é menor que 10 (menor v_j é escolhido).

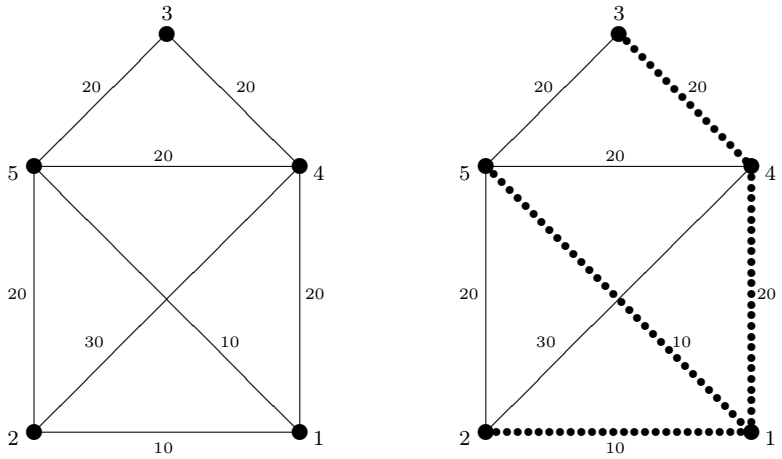


Figura 5.1: Primeiro exemplo onde, à esquerda, está o grafo de entrada $G = (V, E)$, onde $V = \{1, 2, 3, 4, 5\}$, e à direita o esteio correspondente de G, composto pelas arestas $(1, 10, 2)$ (escolhida pelos vértices 1 e 2), aresta $(1, 20, 4)$ (escolhida pelo vértice 4), aresta $(1, 10, 5)$ (escolhida pelo vértice 5) e aresta $(3, 20, 4)$ (escolhido pelo vértice 3), onde $(1, 10, 2)$ é a única aresta *zero-diferença*.

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. No exemplo da Figura 5.1, apenas uma das arestas é *zero-diferença*: a aresta (1, 10, 2). No caso desse exemplo, o esteio gerado na primeira iteração do algoritmo é uma árvore geradora de peso mínima (MST) de G. A condição de parada do algoritmo é ter apenas uma aresta *zero-diferença*. Caso isso não ocorra, uma etapa adicional de compactação precisa ser executada para que o algoritmo tenha uma nova iteração.

O Algoritmo 6 apresenta a ideia de como a proposta funciona. Consiste em encontrar o esteio do grafo e, se necessário, ou seja, se a floresta geradora não for uma árvore, compactar o grafo para uma nova iteração do algoritmo. A solução consiste em um algoritmo

paralelo, ou seja, as etapas são executadas por vários processadores, encontrando a árvore geradora mínima de forma colaborativa.

Algoritmo 6 Algoritmo para MST sem o grafo bipartido

Entrada: Um grafo conexo $G = (V, E)$, onde $V = \{v_1, v_2, \ldots, v_n\}$ é um conjunto com n vértices e E é um conjunto com m arestas (v_i, w_{ij}, v_j), onde v_i e v_j são vértices de V e w_{ij} é o peso correspondente à aresta entre v_i e v_j.
Saída: Uma árvore geradora mínima de G cujas arestas estão em *ConjuntoSolucao*.

1: *ConjuntoSolucao* := vazio.
2: // Encontrando a solução para MST.
3: *numzerodif* = 0
4: $r := 0$
5: **while** (($r < \log p$) **AND** (*numzerodif* $\neq 1$)) **do** // p é o número de processadores
6: // Encontrando as arestas mais leves para cada vértice e armazenando em *mais_ leve*$[v_i]$.
7: **for each** ($v_i \in V$) **in parallel do**
8: *mais_ leve*$[v_i]$ = -1
9: **end for**
10: **for each** ($v_i \in V$) **in parallel do**
11: Encontra aresta *mais_ leve*$[v_i]$.
12: **end for**
13: // Obtendo o esteio.
14: **for each** ($e_j \in E$) **in parallel do**
15: $d_S(e_j) = 0$
16: **end for**
17: **for each** ($v_i \in V$) **in parallel do**
18: $d_S(mais\_\ leve[v_i]) = d_S(mais\_\ leve[v_i]) + 1$ // função atômica
19: **end for**
20: // Adicionando as arestas do esteio a *ConjuntoSolucao*
21: // e calculando o número de arestas zero-diferença
22: *numzerodif* = 0 // *numzerodif* armazena o número de arestas zero-diferença
23: **for each** ($e_j \in E$) **in parallel do**
24: **if** ($d_S(e_j) \geq 1$) **then**
25: Adicione a aresta e_j a *ConjuntoSolucao*
26: **if** ($d_S(e_j) == 2$) **then**
27: *numzerodif* = *numzerodif* +1 // função atômica
28: **end if**
29: **end if**
30: **end for**
31: **if** (*numzerodif* $\neq 1$) **then**
32: // Compactação do grafo G
33: Computar os componentes conexos
34: **for each** ($v_i \in V$) **in parallel do**
35: Atualizar v_i
36: **end for**
37: **for each** ($e_j \in E$) **in parallel do**
38: Atualizar $e_j.v_1$
39: Atualizar $e_j.v_2$
40: **if** ($e_j.v_1 == e_j.v_2$) **then**
41: Eliminar e_j
42: **end if**
43: **end for**
44: **end if**
45: $r := r + 1$
46: **end while**

Se houver arestas com o mesmo peso no grafo, a rotulação dos vértices influencia o número de iterações do algoritmo e a árvore geradora mínima calculada. A Figura 5.2 apresenta o mesmo grafo da Figura 5.1, mas a rotulagem dos vértices é diferente. Neste exemplo, o esteio gerado na primeira iteração do algoritmo, formado pelas arestas $(1, 20, 4)$, $(2, 10, 3)$ e $(3, 10, 50)$, apresenta duas arestas *zero-diferença* $(1, 20, 4)$ (escolhida pelos vértices 1 e 4) e $(2, 10, 3)$ (escolhida por 2 e 3), implicando em outra iteração do algoritmo.

Analisando o exemplo da Figura 5.2, vê-se que é necessário compactar o grafo G, já que o número de arestas *zero-diferença* é maior que um. Essa compactação começa com o cálculo dos componentes conexos formados pelas arestas do esteio e a eliminação das arestas que interconectam vértices do mesmo componente. Assim, cada componente será

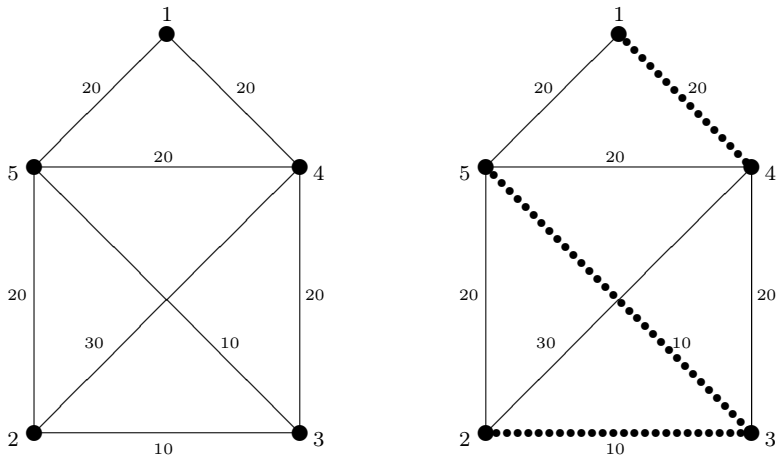


Figura 5.2: À esquerda, o grafo de entrada $G = (V, E)$, onde $V = \{1, 2, 3, 4, 5\}$, e à direita é ilustrado o esteio correspondente a G, composto pelas arestas $(1, 20, 4)$ (escolhida pelos vértices 1 e 4), aresta $(2, 10, 3)$ (escolhida pelos vértices 2 e 3) e aresta $(3, 10, 5)$ (escolhida pelo vértice 5), onde $(1, 20, 4)$ e $(2, 10, 3)$ são arestas *zero-diferença*.

reduzido a um único vértice, que será rotulado com o menor rótulo dentre os vértices do componente. Para o exemplo da Figura 5.2, o grafo resultante da compactação terá dois vértices (1 e 2) e quatro arestas como ilustrado na Figura 5.3.

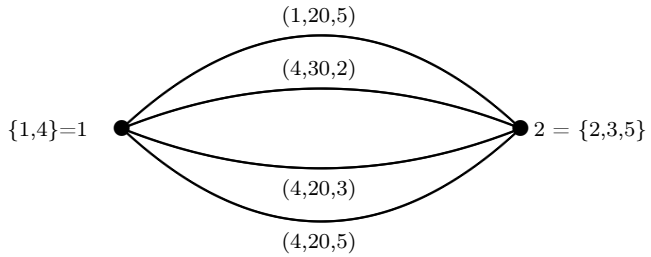


Figura 5.3: Grafo compactado ao final da primeira iteração do algoritmo para o exemplo da Figura 5.2.

Na segunda iteração do algoritmo um esteio é computado para o grafo compactado. No exemplo da Figura 5.3 o grafo possui apenas dois vértices, assim, o esteio terá uma única aresta e esta será uma aresta *zero-diferença*, resultando na condição de parada do algoritmo. Vale salientar que existem três arestas com mesmo peso 20 interligando os vértices 1 e 2 do grafo compactado. Qualquer uma delas pode ser escolhida. No nosso algoritmo escolhemos a que aparece primeiro, portanto, a aresta $(1, 20, 5)$ será escolhida tanto para o vértice 1 quanto para o vértice 2. Assim, uma árvore geradora de peso mínimo resultante da execução do Algoritmo 6, tendo como entrada o grafo do exemplo ilustrado na Figura 5.2, apresenta peso total 60, sendo formada pelas arestas $(1, 20, 4)$, $(2, 10, 3)$, $(3, 10, 5)$ e $(1, 20, 5)$ e está ilustrada na Figura 5.4.

Como no modelo BSP/CGM é esperada a execução de $O(\log p)$ rodadas, estabelecemos que, após $\log p$ rodadas, se a árvore geradora mínima não for encontrada, o algoritmo continua o processamento localmente na CPU .

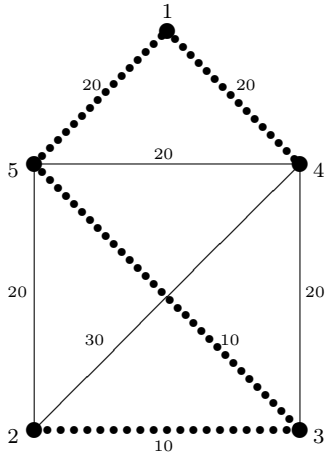


Figura 5.4: Árvore geradora mínima para o grafo da Figura 5.2.

5.3 Correção e complexidade

Nesta seção, são apresentados alguns lemas e teoremas, e as provas correspondentes, sobre a correção e complexidade do algoritmo proposto.

Lema 3. *Considere um grafo conexo $G = (V, E)$. Seja G' o grafo obtido adicionando as arestas escolhidas para compor o esteio S (linha 18 do Algoritmo 6). Então G' é acíclico. Além disso, se S contiver exatamente uma aresta zero-diferença, então G' é uma árvore geradora mínima de G.*

Demonstração. A maneira como as arestas são selecionadas para construir o esteio S, não cria ciclos. Para cada vértice v_i, seleciona-se a aresta com o menor peso. Como tem-se $|V| = n$, são escolhidas no máximo $n - 1$ arestas. Suponha que as arestas selecionadas criem um ciclo $C = v_{i_1}, v_{i_2}, .., v_{i_k}$, em que $v_{i_1} = v_{i_k}$. A menor aresta em C foi selecionada por pelo menos dois vértices. Isso contradiz o fato de que C era um ciclo. Se G' tem apenas uma aresta *zero-diferença*, G' tem $n - 1$ arestas. Suponha que G' não seja uma árvore geradora mínima de G. Então existe uma árvore geradora G'' com peso total menor que G', isto é, pelo menos uma das arestas de G'' tem um peso menor que uma das arestas de G'. Mas pela regra de construção do esteio, as arestas de G' são sempre as de menor peso. Então G' é uma árvore geradora mínima de G. □

Lema 4. *Seja V' o conjunto de vértices do grafo compactado H, resultante do passo 35 do Algoritmo 6. Seja k o número de arestas zero-diferença no esteio S obtido no passo 18, então o número de vértices em V' é k.*

Demonstração. Após a primeira rodada, cada vértice seleciona a menor aresta adjacente. Se todas as arestas de G' foram selecionadas por seus dois vértices, $|G'| = k = \lceil n/2 \rceil$, todas as k arestas (v_i, v_j) do esteio serão unidas em novos vértices v'_i na nova rodada do algoritmo. Se existem arestas de G' que não foram escolhidas pelos seus dois vértices, este fato indica que existem vértices que se conectam a outros que possuem arestas *zero-diferença* em G', e eles todos formam um componente conexo. □

Teorema 6. *O número de arestas com diferença zero é pelo menos dividido por 2 em cada iteração.*

5.4 Detalhes da Implementação

Esta seção descreve a implementação desenvolvida para verificar se o algoritmo proposto neste capítulo possui um bom desempenho em máquinas paralelas reais. São apresentados os ganhos obtidos pelo algoritmo em comparação com sua versão para CPU (sequencial) e com o algoritmo proposto no Capítulo 3. Vale lembrar que o algoritmo anterior constrói explicitamente um grafo bipartido correspondente ao grafo de entrada e que esse passo tem um custo razoável em termos de tempo e memória. A versão para CPU (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/NewMinimumSpanningTree. O Algoritmo 7 representa a implementação de CUDA.

Algoritmo 7 Algoritmo para MST sem grafo bipartido com chamadas de procedimentos

Entrada: Um grafo conexo $G = (V, E)$, onde $V = \{v_1, v_2, \ldots, v_n\}$ é um conjunto com n vértices e E é um conjunto com m arestas (v_i, w_{ij}, v_j), onde v_i e v_j são vértices de V e w_{ij} é o peso correspondente à aresta entre v_i e v_j.
Saída: Uma árvore geradora mínima de G cujas arestas estão em *ConjuntoSolucao*.

```
1: ConjuntoSolucao := vazio.
2: // Encontrando a solução para MST.
3: numzerodif = 0
4: r := 0
5: while ((r < log p) AND (numzerodif ≠ 1)) do // p é o número de processadores
6:    // Encontrando as arestas mais leves para cada vértice e armazenando em mais_leve[v_i].
7:    set mais_leve[v_i] to -1
8:    kernel call EncontraArestasMaisLeves1()
9:    kernel call EncontraArestasMaisLeves2()
10:   // Obtendo o esteio.
11:   set d_S(e_j) to 0
12:   kernel call MarcaArestasEsteio()
13:   // Adicionando as arestas do esteio a ConjuntoSolucao
14:   // e calculando o número de arestas zero-diferença.
15:   numzerodif = 0
16:   kernel call AtualizaConjuntoSolucaoANDCalculaNumzerodif()
17:   if (numzerodif ≠ 1) then
18:      // Compaction of graph G
19:      kernel call InitializaArrestasParaCalcularComponentesConexos()
20:      kernel call CalculaComponentesConexos()
21:      kernel call AtualizaVertices()
22:      kernel call MarcaVerticesParaDelecao()
23:      kernel call MarcaArestasParaDelecao()
24:   end if
25:   r := r + 1
26: end while
27: copy ConjuntoSolucao to CPU
```

A versão CUDA implementa as etapas do algoritmo usando nove funções *kernel* CUDA. Imediatamente após a leitura dos dados do grafo de entrada, esses dados são copiados para a memória global da GPGPU. Duas funções *kernel* são usadas para escolher a aresta com o menor peso para cada vértice (Procedimentos *EncontrarArestasMaisLeves1* e *EncontrarArestasMaisLeves2*). Depois de selecionar as arestas, usamos outra função para marcá-la como uma aresta do esteio (Procedimento *MarcaArestasEsteio*). Também usamos uma

função para calcular o número de arestas de *zero-diferença* e copiar as arestas do esteio para o vetor *ConjuntoSolucao* (Procedimento *AtualizaConjuntoSolucaoANDCalculaNumzerodif*). Algoritmo 8 apresenta a representação dessas quatro funções *kernel*.

Outras cinco funções *kenel* são usadas para a compactação do grafo, incluindo o cálculo dos componentes conectados, onde é usada a proposta apresentada em [18]. Algumas funções atômicas disponíveis na biblioteca CUDA são necessárias na etapa de compactação. Em todos os testes descritos neste capítulo foram utilizados blocos de *threads* de tamanho 64. A implementação do algoritmo não utilizou nenhuma técnica especial de programação CUDA, pois o objetivo principal do nosso trabalho era apresentar um algoritmo eficiente no modelo BSP/CGM que pudesse ser facilmente implementado em uma máquina paralela real.

5.5 Resultados Experimentais

Como descrito anteriormente, o objetivo primário da implementação é demonstrar a funcionalidade do algoritmo proposto em uma máquina paralela real. Para isso, usamos o Ambiente 4 (GTX745 - UFMS), Ambiente 5 (M4000 - UFMS) e Ambiente 6 (K40M - UFG), descritos na Seção 2.2, para executar as implementações sequenciais e CUDA. Para analisar a eficiência e escalabilidade do algoritmo, usamos como entrada os mesmos dois conjuntos de grafos descritos na Seção 3.6.1. O primeiro conjunto consiste em grafos construídos artificialmente usando um gerador de grafos aleatórios. O segundo é composto por grafos das redes rodoviárias dos Estados Unidos disponibilizados no Nono Desafio da Implementação DIMACS.

Para cada um dos grafos de entrada, as implementações para CPU e CUDA foram executadas 20 vezes, utilizando o tempo médio de execução para analisar o comportamento experimental do algoritmo. Os tempos apresentados não levam em conta a leitura do dados de entrada. A Tabela 5.1 apresenta os resultados dos testes para ambos os conjuntos de grafos de entrada (Tabela 3.4 e Tabela 3.5) usando o Ambiente 6 (K40M - UFG).

Cada linha da Tabela 5.1 mostra o número de iterações do algoritmo, o tempo de execução do algoritmo descrito no capítulo 3 usando CUDA (chamado de "CUDA antigo"), o tempo de execução da implementação para CPU (chamada de "CPU novo"), o tempo de execução da nova implementação CUDA (chamada de "CUDA novo"), o *speedup* relacionando algoritmo antigo e o algoritmo novo, na versão para CPU e versão CUDA. Vale ressaltar que o número de iterações do algoritmo necessárias para encontrar a árvore geradora mínima para o primeiro conjunto de grafos de entrada não excedeu a seis, e o *speedup* da nova implementação CUDA em comparação com a implementação do algoritmo anterior variou de sete a 83. A Tabela 5.2 e a Tabela 5.3 mostram alguns resultados de testes usando o Ambientes 5 (M4000 - UFMS) e o Ambiente 4 (GTX745 - UFMS), respectivamente.

Outra vantagem deste novo algoritmo, comparando com nossa proposta anterior, apresentada no capítulo 3 e no artigo [63], é a menor quantidade de memória utilizada, já que não é mais necessário criar o grafo bipartido, que duplica o número de arestas do grafo original. Este fato é evidenciado na Tabela 5.3 que mostra os resultados de testes usando

Algoritmo 8 Algoritmo para MST sem grafo bipartido - Alguns Procedimentos

```
1: Procedure FindLightestEdges1
2: // thread_id is the identification of each thread during a kernel execution
3: // m is the number of edges of the graph
4: // lightest_edge[v] stores the identification of the lightest edge for the vertex v
5: if (thread_id < m) then
6:    v = E[thread_id].v1;
7: // Begin atomic function
8:    if ((lightest_edge[v] == −1) OR
9:     (E[lightest_edge[v]].w > E[thread_id].w) OR
10:    ((E[lightest_edge[v]].w == E[thread_id].w)
11:    AND (E[lightest_edge[v]].v2 > E[thread_id].v2)) then
12:       lightest_edge[v] = thread_id
13:    end if
14: // End atomic function
15: end if
16: EndProcedure
17:
18: Procedure FindLightestEdges2
19: // thread_id is the identification of each thread during a kernel execution
20: // m is the number of edges of the graph
21: // lightest_edge[v] stores the identification of the lightest edge for the vertex v
22: if (thread_id < m) then
23:    v = E[thread_id].v2;
24: // Begin atomic function
25:    if ((lightest_edge[v] == −1) OR
26:     (E[lightest_edge[v]].w > E[thread_id].w) OR
27:     ((E[lightest_edge[v]].w == E[thread_id].w)
28:     AND (E[lightest_edge[v]].v1 > E[thread_id].v1)) then
29:       lightest_edge[v] = thread_id
30:    end if
31: // End atomic function
32: end if
33: EndProcedure
34:
35: Procedure MarkStrutEdges
36: // thread_id is the identification of each thread during a kernel execution
37: // n is the number of vertices of the graph
38: if (thread_id < n) then
39:    v = thread_id
40:    dS[lightest_edge[v]] = dS[lightest_edge[v]] +1; // atomic function
41: end if
42: EndProcedure
43:
44: Procedure UpdateSolutionSetANDCalculateNumdiff
45: // thread_id is the identification of each thread during a kernel execution
46: // m is the number of edges of the graph
47: if (thread_id < m) then
48:    if (dS[thread_id] ≥ 1) then
49:    // Begin atomic function
50:       SolutionEdgeSet[SolutionLenght] = E[thread_id]
51:       SolutionLenght + +
52:    // End atomic function
53:       if (dS[thread_id] == 2) then
54:          numdiff = numdiff +1; // atomic function
55:       end if
56:    end if
57: end if
58: EndProcedure
```

o Ambiente 4 (GTX745 - UFMS) que tem menos memória disponível, apenas 4 GB (conforme especificado na Tabela 2.2).

A Tabela 5.2 também mostra os resultados dos testes usando como entrada os grafos do nono Desafio de Implementação DIMACS (Tabela 3.5). Para essas entradas, o *speedup* da implementação de CUDA comparada com a implementação para CPU variou de sete a 14. Como o número de vértices desse conjunto de grafos de entrada é muito maior que o anterior, o algoritmo precisou de nove a 12 iterações para encontrar a árvore geradora

Tabela 5.1: Resultados dos testes, em segundos, para ambos os conjuntos de grafos de entrada usando o Ambiente 6 (K40M - UFG).

Grafos de entrada	**Número de Iterações**	**Tempo CUDA antigo (seg.)**	**Tempo CPU novo (seg.)**	**Tempo CUDA novo (seg.)**	**Speedup CUDA antigo x CPU novo**	**Speedup CUDA antigo x CUDA novo**	**Speedup CPU novo x CUDA novo**
grafo10a	5	0,898	0,154	0,011	5,826	83,052	14,254
grafo10b	4	1,036	0,301	0,021	3,439	48,824	14,196
grafo10c	3	1,163	0,459	0,039	2,531	29,954	11,836
grafo10d	4	1,476	0,936	0,061	1,576	23,997	15,226
grafo10e	3	1,540	0,964	0,075	1,598	20,412	12,775
grafo15a	5	1,061	0,380	0,023	2,790	45,992	16,485
grafo15b	5	1,375	0,850	0,049	1,618	27,866	17,223
grafo15c	3	1,644	1,123	0,088	1,465	18,774	12,819
grafo15d	3	2,061	1,669	0,127	1,235	16,293	13,194
grafo15e	3	2,467	2,234	0,163	1,104	15,110	13,681
grafo15f	2	3,866	3,920	0,341	0,986	11,333	11,489
grafo15g	2	5,430	5,866	0,492	0,926	11,033	11,918
grafo15h	2	6,903	7,828	0,631	0,882	10,948	12,414
grafo20a	5	1,220	0,624	0,036	1,957	33,936	17,343
grafo20b	4	1,733	1,299	0,084	1,335	20,634	15,462
grafo20c	4	2,718	2,639	0,162	1,030	16,758	16,272
grafo20d	3	3,029	3,062	0,217	0,989	13,943	14,097
grafo20e	3	3,598	3,741	0,283	0,962	12,692	13,197
grafo25a	5	1,521	1,000	0,055	1,522	27,492	18,067
grafo25b	4	2,289	2,063	0,130	1,110	17,572	15,837
grafo25c	4	3,873	4,242	0,254	0,913	15,255	16,708
grafo25d	2	3,389	3,411	0,315	0,994	10,758	10,827
grafo25e	3	5,330	5,747	0,481	0,927	11,088	11,955
grafo30a	6	2,062	1,749	0,087	1,179	23,644	20,057
grafo30b	5	3,460	3,683	0,220	0,939	15,692	16,705
grafo30c	3	4,433	4,650	0,374	0,953	11,862	12,442
grafo30d	3	6,167	6,982	0,539	0,883	11,441	12,952
grafo30e	3	7,528	9,081	0,703	0,829	10,712	12,923
USA-road-d.NY	9	0,924	0,091	0,012	10,182	76,743	7,537
USA-road-d.BAY	10	0,950	0,101	0,014	9,362	69,590	7,433
USA-road-d.COL	9	0,965	0,133	0,017	7,241	56,760	7,839
USA-road-d.FLA	10	1,027	0,361	0,033	2,846	30,690	10,783
USA-road-d.NW	10	1,062	0,381	0,036	2,785	29,175	10,476
USA-road-d.NE	10	1,118	0,541	0,046	2,068	24,197	11,702
USA-road-d.CAL	11	1,210	0,660	0,056	1,832	21,558	11,768
USA-road-d.LKS	11	1,369	1,003	0,083	1,366	16,427	12,030
USA-road-d.E	11	1,504	1,453	0,113	1,035	13,358	12,900
USA-road-d.W	11	1,981	2,274	0,183	0,871	10,821	12,421
USA-road-d.CTR	12	3,773	6,258	0,446	0,603	8,466	14,044
USA-road-d.USA	12	5,493	9,178	0,771	0,598	7,127	11,909

mínima.

É difícil comparar os tempos de execução da solução apresentada com outras propostas paralelas que foram publicadas recentemente porque a maioria desses algoritmos usou grafos de entrada para problemas específicos e recursos computacionais diferentes ou não disponibilizaram o código fonte. No entanto, foi possível comparar os resultados obtidos com a solução paralela proposta por Manoochehri et al. no artigo [35] visto que os testes foram executados em um ambiente similar (NVIDIA Tesla K40) e com o mesmo subconjunto de grafos de entrada (um subconjunto dos grafos do Nono DIMACS). A Tabela 5.4 apresenta os valores dos tempos de execução e os *speedups* obtidos comparando a solução paralela apresentada neste capítulo e a solução de Manoochehri et al..

A Figura 5.5 ilustra, usando tempos de execução para alguns gráficos do 9DIMACS apresentados na Tabela 5.4), a comparação entre as implementações do novo algoritmo

Tabela 5.2: Alguns resultados de testes, em segundos, usando o Ambiente 5 (M4000 - UFMS).

Grafos de entrada	**Tempo CUDA antigo (seg.)**	**Tempo CPU novo (seg.)**	**Tempo CUDA novo (seg.)**	**Speedup CUDA antigo x CPU novo**	**Speedup CUDA antigo x CUDA novo**	**Speedup CPU novo x CUDA novo**
grafo10a	0,763	0,133	0,011	5,737	68,812	11,995
grafo10e	1,117	0,810	0,061	1,379	18,267	13,244
grafo15a	0,863	0,329	0,023	2,621	37,015	14,125
grafo15e	1,649	1,878	0,133	0,878	12,387	14,106
grafo20a	0,967	0,528	0,035	1,831	27,558	15,051
grafo20e	2,298	3,108	0,227	0,739	10,136	13,711
grafo25a	1,114	0,838	0,052	1,329	21,452	16,139
grafo25e	3,177	4,773	0,348	0,666	9,136	13,728
grafo30a	1,459	1,451	0,082	1,006	17,902	17,793
grafo30e	4,448	7,553	0,492	0,589	9,042	15,352
USA-road-d.NY	0,659	0,081	0,012	8,163	54,270	6,648
USA-road-d.BAY	0,656	0,086	0,014	7,594	47,899	6,307
USA-road-d.COL	0,663	0,115	0,017	5,782	38,587	6,674
USA-road-d.FLA	0,750	0,321	0,032	2,341	23,647	10,102
USA-road-d.NW	0,752	0,335	0,034	2,243	21,887	9,758
USA-road-d.NE	0,814	0,473	0,044	1,720	18,440	10,724
USA-road-d.CAL	0,858	0,580	0,054	1,479	15,981	10,803
USA-road-d.LKS	0,985	0,875	0,077	1,126	12,754	11,328
USA-road-d.E	1,086	1,282	0,100	0,847	10,859	12,820
USA-road-d.W	1,423	1,974	0,159	0,721	8,938	12,400
USA-road-d.CTR	2,698	5,693	0,409	0,474	6,603	13,936
USA-road-d.USA	3,758	8,039	0,617	0,467	6,094	13,037

Tabela 5.3: Alguns resultados de testes, em segundos, usando o Ambiente 4 (GTX745 - UFMS).

Grafos de entrada	**Tempo CUDA antigo (seg.)**	**Tempo CPU novo (seg.)**	**Tempo CUDA novo (seg.)**	**Speedup CUDA antigo x CPU novo**	**Speedup CUDA antigo x CUDA novo**	**Speedup CPU novo x CUDA novo**
grafo10a	0,315	0,111	0,028	2,833	11,398	4,023
grafo10e	1,100	0,728	0,141	1,512	7,792	5,152
grafo15a	0,554	0,276	0,063	2,011	8,735	4,344
grafo15e	2,578	1,644	0,312	1,568	8,272	5,277
grafo20a	0,931	0,432	0,097	2,156	9,586	4,445
grafo20e	4,938	2,660	0,547	1,857	9,028	4,863
grafo25a	1,206	0,679	0,149	1,777	8,116	4,567
grafo25e	insufficient memory	4,022	0,840	-	-	4,788
grafo30a	1,811	1,180	0,240	1,535	7,537	4,909
grafo30e	insufficient memory	6,424	1,201	-	-	5,349
USA-road-d.NY	0,443	0,069	0,023	6,394	19,273	3,014
USA-road-d.BAY	0,441	0,078	0,026	5,676	16,848	2,968
USA-road-d.COL	0,459	0,103	0,033	4,457	13,931	3,126
USA-road-d.FLA	0,578	0,278	0,074	2,076	7,853	3,783
USA-road-d.NW	0,578	0,295	0,079	1,961	7,288	3,717
USA-road-d.NE	0,675	0,415	0,107	1,629	6,315	3,877
USA-road-d.CAL	0,735	0,507	0,133	1,450	5,524	3,811
USA-road-d.LKS	0,927	0,771	0,198	1,203	4,679	3,891
USA-road-d.E	1,096	1,137	0,266	0,965	4,122	4,274
USA-road-d.W	1,573	1,730	0,426	0,909	3,688	4,057
USA-road-d.CTR	3,452	5,331	1,182	0,647	2,920	4,510
USA-road-d.USA	4,554	7,718	1,854	0,590	2,456	4,162

proposto neste capítulo e as soluções CUDA apresentadas por Manoochehri et al. em [35] (rotulado de "Manoochehri et al.") e a solução descrita no Capítulo 3 (rotulado "CUDA

Tabela 5.4: Tempos de execução em milissegundos (ms) usando NVIDIA Tesla K40.

Grafos de entrada	**Manoochehri et al.**	**CUDA antigo**	**CPU novo**	**CUDA novo**	**Speedup Manoochehri x CUDA novo**
USA-road-d.NY	217	923,8	90,7	12,0	18,08
USA-road-d.FLA	736	1.026,8	360,8	33,5	21,97
USA-road-d.E	1.959	1.504,1	1.452,5	112,6	17,40
USA-road-d.W	3.451	1.981,4	2.274,3	183,1	18,85
USA-road-d.USA	13.407	5.492,9	9.177,8	770,7	17,40

antigo"). O tempo de execução do "CUDA novo" é praticamente imperceptível na Figura 5.5 para os grafos USA-road-d.NY e USA-road-d.FLA. Os resultados mostram a funcionalidade e eficiência do algoritmo em uma máquina paralela real, em comparação com dois trabalhos recentemente publicados. Os testes experimentais mostraram que o algoritmo apresenta bons *speedups* usando uma implementação relativamente simples.

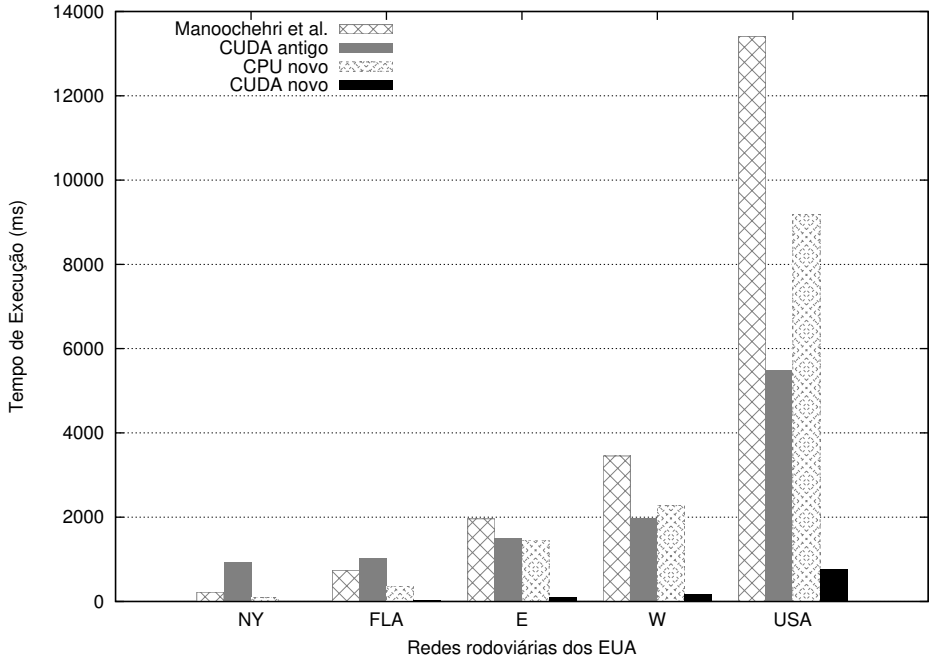


Figura 5.5: Comparação do tempo de execução tendo como entrada alguns grafos do 9DIMACS usando a GPGPU NVIDIA Tesla K40.

5.6 Contribuições

Dentre as contribuições alcançadas com a proposta descrita neste capítulo destaca-se:

- um novo algoritmo paralelo e eficiente, usando o modelo BSP/CGM, para o problema da árvore geradora mínima, sem usar o grafo bipartido; e
- um artigo publicado nos anais do *30th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD 2018), com título *"A new*

efficient parallel algorithm for minimum spanning tree" [64].

Capítulo 6

Alinhamento de Sequências Biológicas

A estrutura mais utilizada para armazenar dados biológicos são as sequências de caracteres, também chamadas de biosequências. Vários algoritmos para processamento de cadeias de caracteres já foram desenvolvidos e publicados, sendo apresentados em diversos livros de algoritmos e estrutura de dados [17, 22]. No entanto, para atender os pesquisadores de bioinformática com maior presteza, uma alternativa é propor soluções mais eficientes e/ou confiáveis para os problemas existentes. O projeto e a implementação dessas soluções podem envolver as abordagens sequencial e paralela, tolerante ou não a falhas, com soluções aproximadas ou exatas. Neste capítulo descrevemos as soluções propostas para alinhamento de sequências, tanto duas as duas (*pairwise*) quanto em conjunto de três ou mais.

6.1 Alinhamento *Pairwise*

Os projetos de sequenciamento de genomas têm produzido uma grande quantidade de dados biológicos nas últimas décadas. No entanto, para agregar valor a esse conjunto de dados é necessário realizar análises que reflitam as informações contidas nos mesmos e as relações existentes com outros dados disponíveis.

Visto que o DNA é formado por sequências de nucleotídeos, uma forma de inferir propriedades a respeito de novos organismos envolve a comparação entre as sequências que compõem o DNA recentemente identificado e as de espécies previamente estudadas, pois sabe-se que a similaridade de sequências de DNA geralmente envolve a similaridade de estruturas e de funcionalidades [12].

Para apoiar este contexto a biologia molecular computacional tem dedicado esforços para o estudo e desenvolvimento de ferramentas que permitam a eficiente comparação de sequências. O conceito de alinhamento é muito importante quando se fala em comparação de sequências. Alinhar duas sequências consiste em representá-las "uma sobre a outra", deixando explícita a correspondência entre seus caracteres [39]. Em certas situações, pode ser necessário inserir espaços nas sequências a fim de tornar o alinhamento mais apropriado. Tais espaços podem ser inseridos em qualquer posição em ambas as sequências formando lacunas ou buracos (*gaps*). Existem três tipos de alinhamento [56]:

- Global: que contém todos os caracteres das duas sequências;

- Local: que contém *substrings* das sequências; e
- Semi-global: não penaliza espaços nas extremidades das sequências, sendo usado para encontrar alinhamentos em que uma sequência está contida na outra ou entre um sufixo de uma e um prefixo da outra.

Para medir a qualidade de um alinhamento é utilizado um sistema de pontuação que atribui valores a para de caracteres. A Figura 6.1 ilustra um alinhamento entre duas sequências e sua pontuação (*score*) correspondente a cada emparelhamento de caractere e o valor total do alinhamento que consiste na soma dos valores individuais. O caractere de uma sequencia pode ser alinhado com um espaço na outra sequência. Um espaço é denotado pelo símbolo '-'. A pontuação apresentada foi obtida usando o seguinte sistema de pontuação:

- +1: se os caracteres forem os mesmos (*match*);
- -1: se os caracteres forem diferentes (*mismatch*); e
- -2: se um dos caracteres for um espaço (*gap*).

Seq. 1	A	C	T	T	C	C	–	–	A	G	A	
Seq. 2	A	G	T	T	C	C	G	G	A	G	G	
Score	+1	−1	+1	+1	+1	+1	−2	−2	+1	+1	−1	= +1

Figura 6.1: Exemplo de um alinhamento entre duas sequências e a pontuação correspondente.

No exemplo da Figura 6.1, todas as correspondências entre um caractere e um espaço foram pontuadas da mesma forma. Entretanto, do ponto de vista biológico, é mais significativo manter espaços juntos [12] do que criar novas lacunas. Por este motivo, um modelo de adição de espaços utilizando função afim é normalmente empregado, determinando que o primeiro espaço de uma lacuna tem uma penalidade maior do que os espaços contínuos subsequentes.

6.1.1 Algoritmo *Needleman-Wunsch* (NW)

O algoritmo *Needleman-Wunsch* (NW) [43] foi um dos primeiros métodos propostos para o alinhamento global de sequências. Ele utiliza a técnica de programação dinâmica para calcular a pontuação (*score*) e obter o alinhamento global ótimo entre duas sequências S e T. O alinhamento ótimo é aquele que possui o maior valor possível dentre todos os alinhamentos que podem ser construídos para as sequências de entrada. Considerando que m e n são os tamanhos das sequências S e T a serem alinhadas ($|S| = m$ e $|T| = n$), assim como os algoritmos clássicos de programação dinâmica, NW tem complexidade quadrática relativa ao tamanho das sequências, $O(mn)$, em relação a tempo e memória. Para computar o alinhamento global ótimo é necessário construir uma matriz de similaridade,

também chamada de matriz de programação dinâmica, D, de dimensões $(m+1) \times (n+1)$, onde cada posição $D_{i,j}$, $1 \leq i \leq m$ e $1 \leq j \leq n$, é calculada usando a equação 6.1:

$$D_{i,j} = max \begin{cases} D_{i-1,j-1} + P_{S_i,T_j} & \text{match/mismatch} \\ D_{i,j-1} + g & \text{espaço na sequência } T \\ D_{i-1,j} + g & \text{espaço na sequência } S \end{cases} \quad (6.1)$$

O valor da penalidade por inserir um espaço no alinhamento é representado por g. P é a matriz que contém a pontuação para alinhar cada par de símbolos do alfabeto empregado. Os elementos da primeira linha recebem o valor $j \times g$ e da primeira coluna recebem o valor $i \times g$, onde j é o índice da coluna e i o índice da linha. Para qualquer valor de i e j, $D_{i,j}$ é a pontuação do melhor alinhamento entre os prefixos de tamanho i de S e j de T. Após preencher a matriz, o elemento $D_{m,n}$ contém a pontuação do alinhamento global ótimo. Para encontrar o alinhamento é necessário percorrer a matriz a partir do elemento $D_{m,n}$ até o elemento $D_{0,0}$ (*traceback*), passando por cada valor utilizado para computar a pontuação do alinhamento ótimo.

6.1.2 Algoritmo *Smith-Waterman* (SW)

O algoritmo *Smith-Waterman* (SW) [57] é uma adaptação de NW para encontrar alinhamentos locais entre duas sequências. Existem duas diferenças fundamentais entre estes algoritmos: os valores de inicialização da matriz de programação dinâmica e a localização da pontuação do alinhamento local ótimo.

Em SW os elementos da primeira linha e da primeira coluna da matriz são inicializados com zero. As posições restantes da matriz são calculadas seguindo a equação 6.2.

$$D_{i,j} = max \begin{cases} D_{i-1,j-1} + P_{S_i,T_j} \\ D_{i,j-1} + g \\ D_{i-1,j} + g \\ 0 \end{cases} , \quad (6.2)$$

A pontuação do melhor alinhamento local corresponde ao maior valor presente na matriz D. Para obter um alinhamento local ótimo é necessário percorrer a matriz a partir da posição onde se encontra o maior valor presente até encontrar um valor zero. Deve-se notar que vários alinhamento locais ótimos podem ser encontrados. A Tabela 6.1 ilustra um alinhamento local usando o mesmo sistema de pontuação considerado no exemplo da Figura 6.1.

Pode-se notar na Tabela 6.1 que a pontuação máxima calculada é 3 e este valor é encontrado em três diferentes posições da matriz. Para este exemplo existem três alinhamentos locais ótimos.

Tabela 6.1: Matriz de programação dinâmica para alinhamento de duas sequências usando SW.

	–	T	G	A	T	G	G	A
–	0	0	0	0	0	0	0	0
G	0	0	1	0	0	1	1	0
A	0	0	0	2	0	0	0	2
T	0	1	0	0	**3**	1	0	0
A	0	0	0	0	1	2	0	1
G	0	0	1	0	0	2	**3**	1
G	0	0	1	0	0	1	**3**	2

6.1.3 Uma Abordagem para Alinhamento *Pairwise* usando MPI

Uma dificuldade em paralelizar este tipo de algoritmo origina-se da sua grande dependência de dados para o cômputo da matriz de programação dinâmica. Uma abordagem paralela usando memória distribuída e MPI foi apresentada em [1]. A ideia principal dessa abordagem é dividir o cálculo da matriz de programação dinâmica entre p processadores, onde cada processador P_i fica responsável pelo cálculo de algumas colunas da matriz, denotado por D_i, conforme ilustrado na Figura 6.2 [1]. Usando o modelo BSP/CGM, são necessárias $O(p)$ rodadas de comunicação e $O(mn/p)$ espaço em memória local para o cálculo do alinhamento entre S e T, onde $|S| = m$ e $|T| = n$.

A sequência S é enviada para todos os processadores e a sequência T é dividida em p partes de tamanho n/p. Cada processador P_i, onde $1 \leq i \leq p$, recebe a i-ésima subsequência de T, cujos elementos correspondem a $t_{((i-1)n/p)+1}$ .. $t_{in/p}$. A notação D_i^j simboliza a j-ésima parte, onde $1 \leq j \leq p$, da submatriz D_i, que é calculada por P_i na rodada r, onde $r = i + j - 2$.

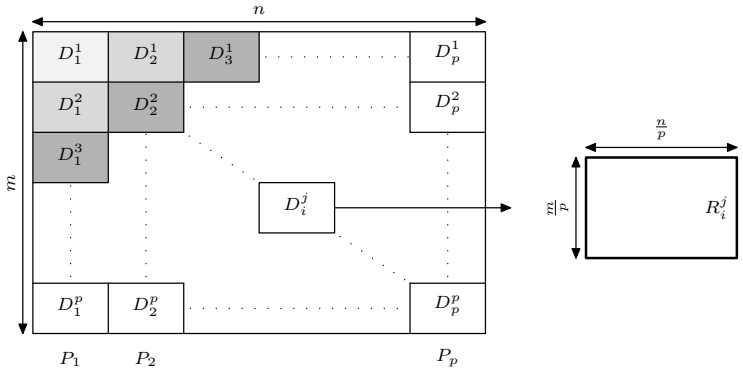


Figura 6.2: Rodadas para cálculo da matriz de programação dinâmica (adaptada de [1]).

Inicialmente P_1 começa calculando D_1^1 na rodada 0. Em seguida P_1 e P_2 podem trabalhar na rodada 1, calculando D_1^2 e D_2^1, respectivamente. Na rodada 2, P_1, P_2 e P_3 podem trabalhar, e assim por diante. Isto ocorre devido à dependência de dados existente no uso do algoritmo de programação dinâmica. Assim sendo, após o cálculo da j-ésima parte da submatriz D_i (representada por D_i^j), o processador P_i envia os elementos da fronteira direita (coluna mais à direita) de D_i^j para o processador P_{i+1}. Estes elementos são representados por R_i^j. Usando R_i^j, o processador P_{i+1} pode calcular a j-ésima parte da

submatriz D_{i+1}. Após $p-1$ rodadas, o processador P_p recebe R^1_{p-1} e calcula a primeira parte da submatriz D_p. Na rodada $2p-2$, o processador P_p recebe R^p_{p-1} e calcula a p-ésima parte da submatriz D_p e termina o cálculo do valor do alinhamento. O cálculo da matriz vai progredindo em "ondas" (cada rodada), por isso este tipo de abordagem é conhecida como "frente de ondas" ou *wavefront*. Pela Figura 6.2 é possível notar que o processador P_p inicia seu trabalho apenas quando o processador P_1 está terminando o cálculo da submatriz D_1 na rodada $p-1$. O Algoritmo 9 [1] apresenta os passos descritos neste parágrafo.

Os resultados apresentados no artigo [1] foram originados de implementações MPI testadas em um *cluster Beowulf* com 64 nós. O tamanho máximo de alinhamento testado foi utilizando uma sequência de 8.196 caracteres e outra de 16.384.

Algoritmo 9 Alinhamento *Pairwise* usando MPI.

Entrada: (1) O número p de processadores; (2) A identificação i do processador, sendo $1 \leq i \leq p$; (3) A sequência S e seu tamanho m; (4) Uma subsequência de T e seu tamanho n/p.
Saída: (1) Valor do alinhamento *pairwise*.

```
1: for 1 ≤ j ≤ p do
2:    if i = 1 then
3:       for (j − 1)m/p + 1 ≤ r ≤ j m/p and 1 ≤ s ≤ n/p do
4:          Calcule os elementos D(r, s);
5:       end for
6:       Envie R_i^j para P_{i+1};
7:    end if
8:    if i ≠ 1 then
9:       Receba R_{i−1}^j de P_{i−1};
10:      for (j − 1)m/p + 1 ≤ r ≤ j m/p and 1 ≤ s ≤ n/p do
11:         Calcule os elementos D(r, s);
12:      end for
13:      if i ≠ p then
14:         Envie R_i^j para P_{i+1};
15:      end if
16:   end if
17: end for
18: if i = p then
19:    Retorne D(m, n/p);
20: end if
```

6.1.4 Uma Abordagem para Alinhamento *Pairwise* usando CUDA

É possível encontrar na literatura várias implementações de alinhamento *pairwise* usando CUDA. Em [34] é apresentada uma das primeiras soluções para alinhamento de sequências usando o algoritmo *Smith-Waterman* e a arquitetura CUDA. A solução paralela do algoritmo *Smith-Waterman* apresentada em [24] faz uso de paralelismo de granularidade fina para testes usando uma única GPU e decomposição de granularidade grossa para uso de múltiplas GPUs.

Nesta seção vamos apresentar a solução desenvolvida por Sandes e Melo [55]. Esta solução paralela, chamada de *CUDAlign 2.1*, implementa o algoritmo de *Smith-Waterman* modificado por Gotoh [14] para contemplar o modelo de adição de espaços utilizando função afim. Este algoritmo proposto por Gotoh preenche três matrizes de programação dinâmica (H, E e F) usando as equações 6.3, 6.4 e 6.5, onde g_{ini} é o valor da penalidade para iniciar uma lacuna e g_{est} corresponde ao valor da penalidade para estender uma lacuna.

Para permitir a comparação de sequências muito grandes, os autores também fazem uso da proposta de Myers e Miller [40] que possibilita o cálculo de alinhamento em espaço linear. CUDAlign 2.1 usa como base o algoritmo CUDAlign 1.0 [54] e possui 6 estágios, finalizando com a visualização do alinhamento. No primeiro estágio os autores empregam um procedimento de *pruning* para reduzir o número de células processadas no cálculo do *score* do alinhamento ótimo.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + P_{S_i,T_j} \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \tag{6.3}$$

$$E_{i,j} = max \begin{cases} E_{i,j-1} + g_{est} \\ H_{i,j-1} + g_{ini} \end{cases} \tag{6.4}$$

$$F_{i,j} = max \begin{cases} F_{i-1,j} + g_{est} \\ H_{i-1,j} + g_{ini} \end{cases} \tag{6.5}$$

O paralelismo do cálculo do alinhamento ocorre em dois níveis na solução de Sandes e Melo. Foram criados blocos de células, de forma a criar antidiagonais de blocos, permitindo, o que foi chamado de paralelismo externo, e dentro de cada bloco existe o paralelismo interno através das antidiagonais das células de cada bloco. As estruturas de dados empregadas são armazenadas utilizando vários tipos de memória disponíveis na GPU: memória compartilhada, memória de textura e memória global.

Os testes realizados e apresentados no artigo [55] trabalham com sequências de tamanho entre 162.114 e 59.373.566 caracteres. Os resultados obtidos se mostraram bem eficientes.

6.2 Abordagens Propostas para o Alinhamento *Pairwise*

Nesta seção serão apresentadas duas abordagens que desenvolvemos para o alinhamento *pairwise*. A primeira utiliza a arquitetura paralela com memória compartilhada e CUDA, sendo que foram produzidas duas implementações, uma inicial e outra com coalescência de memória. A segunda abordagem é uma proposta tolerante a falhas que não foi implementada.

6.2.1 Abordagem utilizando acelerador

A abordagem proposta que utiliza uma placa gráfica como acelerador faz uso de um algoritmo *wavefront* ligeiramente diferente. Nesta versão a cada rodada é calculada uma antidiagonal e não uma submatriz da matriz de programação dinâmica. No entanto, tanto o *host* (CPU) quanto o *device* (GPU) calculam diferentes partes da matriz de alinhamento.

Implementação Inicial

A Figura 6.3 ilustra a ideia do cálculo do alinhamento baseado em antidiagonais. Na primeira rodada o *host* calcula a primeira célula (elemento) da matriz. Na segunda rodada o *host* calcula as duas células da segunda antidiagonal da matriz. Na terceira rodada, o *host* calcula as três células da terceira antidiagonal, e assim por diante. Após algumas rodadas, o *host* para de calcular as antidiagonais e esta tarefa passa a ser executada pelo *device*. Várias *threads* do , uma para cada elemento da antidiagonal, são utilizadas para calcular todos os elementos da antidiagonal em paralelo, através da chamada de uma função *kernel*. O *device* continua o cálculo das antidiagonais até que o número de elementos da antidiagonal seja suficientemente pequeno, tornando desvantajoso o uso da GPU. Não é vantagem acionar o *device* para executar poucas *threads*. Note que o número de células que compõem as antidiagonais cresce e depois decresce. O *host* fica responsável pelo cálculo das primeiras e últimas antidiagonais. Nos testes realizados, considerando a placa gráfica disponível, não obtivemos vantagem no uso do *device* para calcular antidiagonais com menos de 32 elementos.

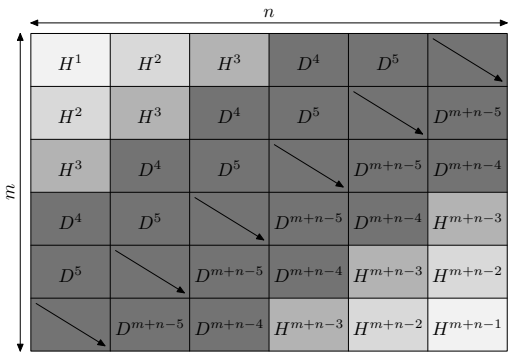


Figura 6.3: Rodadas de execução do Algoritmo 10.

O Algoritmo 10 funciona da seguinte maneira: o *host* e o *device* mantêm as duas sequências que serão alinhadas armazenadas em memória e calculam diferentes porções da matriz de similaridade. O *host* calcula a parte superior esquerda e mais tarde a porção inferior direita (como ilustrado na Figura 6.3). O *device* é responsável pelo cálculo da parte central da matriz.

Implementação com Coalescência

A fim de obter vantagem da coalescência de memória da NVIDIA, uma outra implementação foi proposta. A matriz de programação dinâmica ilustrada na figura 6.3 é armazenada usando um vetor onde a primeira linha é seguida pela segunda e assim por

Algoritmo 10 Alinhamento *Pairwise* utilizando GPGPU.

Entrada: (1) As sequências S e T.
Saída: (1) O valor do alinhamento.

1: *Host* calcula algumas antidiagonais da matriz de similaridade;
2: Realizar copia das sequências e da matriz calculada para o *Device*;
3: *Device* calcula várias antidiagonais da matriz de similaridade;
4: Realizar copia da matriz calculada para o *Host*;
5: *Host* finaliza o cálculo da matriz e encontra o alinhamento;

diante. Com isso posições distantes desse vetor precisam ser acessadas para o cálculo de cada antidiagonal. Por exemplo, para o cálculo da quinta antidiagonal é necessário acessar elementos da primeira, segunda, terceira e quarta linhas da matriz. A ideia para otimizar esse acesso à memória é alterar o *layout* dos dados, mudando a localização de armazenamento dos elementos da matriz. O algoritmo utilizado para o cálculo do alinhamento é o mesmo Algoritmo 10, que calcula uma antidiagonal por rodada. No entanto, passamos a armazenar os elementos da matriz dentro do vetor utilizando a ordem das antidiagonais. A Figura 6.4 ilustra a abordagem original e a nova estratégia de armazenamento dos elementos da matriz. Foi criado um procedimento que calcula a posição de armazenamento do elemento baseado em sua posição original, mostrado no Algoritmo 11. Assim, para o cálculo da quinta antidiagonal é necessário acessar elementos da primeira e segunda linhas da matriz.

e_{00}	e_{01}	e_{02}	e_{03}	e_{04}	e_{05}	e_{06}
e_{10}	e_{11}	e_{12}	e_{13}	e_{14}	e_{15}	e_{16}
e_{20}	e_{21}	e_{22}	e_{23}	e_{24}	e_{25}	e_{26}
e_{30}	e_{31}	e_{32}	e_{33}	e_{34}	e_{35}	e_{36}
e_{40}	e_{41}	e_{42}	e_{43}	e_{44}	e_{45}	e_{46}

$e_{00}=0$	$e_{10}=1$	$e_{01}=2$	$e_{20}=3$	$e_{11}=4$	$e_{02}=5$	$e_{30}=6$
$e_{21}=7$	$e_{12}=8$	$e_{03}=9$	$e_{40}=10$	$e_{31}=11$	$e_{22}=12$	$e_{13}=13$
$e_{04}=14$	$e_{41}=15$	$e_{32}=16$	$e_{23}=17$	$e_{14}=18$	$e_{05}=19$	$e_{42}=20$
$e_{33}=21$	$e_{24}=22$	$e_{15}=23$	$e_{06}=24$	$e_{43}=25$	$e_{34}=26$	$e_{25}=27$
$e_{16}=28$	$e_{44}=29$	$e_{35}=30$	$e_{26}=31$	$e_{45}=32$	$e_{36}=33$	$e_{46}=34$

Figura 6.4: Diferentes arranjos para os elementos da matriz de programação dinâmica.

6.2.2 Abordagem Tolerante a Falhas

O propósito de um algoritmo tolerante a falhas é assegurar que o alinhamento seja calculado ainda que possa ocorrer a falha de algum processador usado no cálculo da solução. A ideia é calcular a matriz de similaridade de forma redundante, isto é, mais de um processador executa o cálculo de cada parte da matriz.

Algoritmo 11 Cálculo do Índice dos Elementos da Matriz Coalescente.

Entrada: Índices originais linha l e coluna c do elemento e o tamanho das sequências m e n.
Saída: O novo índice ind.

```
1: if (l + c) < (m + 1) AND (l + c) < (n + 1) then
2:    ind = ((l + c) * (1 + l + c))/2 + c;
3: else
4:    if (l + c) > (m − 1) AND (l + c) > (n − 1) then
5:       ind = (m + 1) * (n + 1) − ((m + n + 1 − l − c) * (m + n + 2 − l − c)/2) + n − l;
6:    else
7:       less = (m < n) * m + (m >= n) * n;
8:       ind = ((less + 1) * (less + 2))/2 + (less + 1) * (l + c − less − 1);
9:       if (i + j) < n then
10:         ind = ind + c;
11:      else
12:         ind = ind + n − l;
13:      end if
14:   end if
15: end if
```

A Figura 6.2 ilustra a divisão da matriz de similaridade em p^2 partes que serão calculadas utilizando p processadores em $2p-2$ rodadas de comunicação. No algoritmo tolerante a falhas proposto, a parte D_1^1 da matriz será calculada por P_1 e P_p. Após calcular D_1^1, os processadores P_1 e P_p devem enviar as fronteiras das partes calculadas para P_2 e P_{p-1}. Generalizando, cada parte D_i^j será calculada duas vezes, uma por P_i e outra por P_{p-i+1}, sendo que, para $i \neq p$, ambos os processadores devem enviar a fronteira da submatriz calculada para $P_{(i+1) mod\, p}$ e P_{p-i}.

Teorema 7. *O algoritmo tolerante a falhas proposto calcula o valor do alinhamento pairwise entre as sequências S e T em $3p-3$ rodadas de comunicação usando, em cada processador, tempo de execução sequencial $O(\frac{mn}{p})$.*

Demonstração. Alves et al. [1] demonstraram que o alinhamento *pairwise*, usando a abordagem wavefront, necessita de $2p-2$ rodadas de comunicação, $2p-1$ rodadas de execução e tempo de execução sequencial $O(\frac{mn}{p})$ em cada processador, onde m e n são os tamanhos das sequências a serem alinhadas. No algoritmo tolerante a falhas proposto, o cálculo do alinhamento é executado duas vezes, mas não é necessário o dobro de rodadas, pois pela estratégia *wavefront* não temos todos os processadores trabalhando em todas as rodadas. De acordo com a proposta, teremos o dobro em rodadas apenas quando tivermos mais de $\frac{p}{2}$ diferentes submatrizes sendo calculadas em paralelo. Ou seja, iniciamos com $\frac{p}{2}$ rodadas, depois temos $p-1$ "rodadas" que devem ser duplicadas, e encerramos com mais $\frac{p}{2}$ rodadas de execução. O que resulta em $\frac{p}{2} + 2(p-1) + \frac{p}{2} = 3p-2$ rodadas de execução e $3p-3$ rodadas de comunicação. O tempo de execução sequencial de cada processador irá dobrar, o que continua resultando em $O(\frac{mn}{p})$. □

Teorema 8. *Ao final do algoritmo tolerante a falhas proposto, o valor do alinhamento entre as sequências S e T será encontrado mesmo que ocorra a falha de um processador.*

Demonstração. Suponha que o alinhamento *pairwise* seja calculado usando p processadores e que o processador P_i, sendo $1 \leq i \leq p$, falhe em algum momento antes da finalização do cálculo do alinhamento. Pelo algoritmo proposto, as submatrizes D_i^j, onde $1 \leq j \leq p$, que deveriam ser calculadas por P_i também serão calculadas por P_{p-i+1} e suas fronteiras serão enviadas aos processadores $P_{(i+1) mod\, p}$ e P_{p-i} assegurando que o cálculo da matriz de similaridade prossiga até o fim. Assim, ainda que P_i falhe, a redundância do cálculo da matriz realizada por P_{p-i+1} garante que o algoritmo conclua a tarefa. □

Teorema 9. *Pode ser assegurado que o alinhamento entre as sequências S e T é encontrado ao final do algoritmo, ainda que os processadores P_t e P_u falhem, desde que $(t+u) \neq (p+1)$.*

Demonstração. Para que o valor final do alinhamento entre S e T não seja calculado, é necessário que tanto P_t quanto P_v, sendo $v = p - t + 1$, falhem, pois, pelas regras do algoritmo, P_v é responsável pelo cálculo redundante das submatrizes que seriam calculadas por P_t. No entanto, se P_t falha ao mesmo tempo que qualquer outro processador P_u, tal que $u \neq p - t + 1$, o algoritmo chegará ao final do cálculo da matriz de similaridade. Por tanto, como por hipótese $(t+u) \neq (p+1)$, P_t e P_u podem falhar sem afetar o cálculo do alinhamento. □

Teorema 10. *O valor do alinhamento entre as sequências S e T é encontrado usando o algoritmo tolerante a falhas, mesmo que ocorra a falha de k processadores, desde que $0 \leq k \leq \frac{p}{2}$ e que para quaisquer dois processadores P_t e P_u que falhem temos $(t+u) \neq (p+1)$.*

Demonstração. Se for assegurado que para toda combinação de dois processadores que falharam, um não é o complemento do outro (que faz o cálculo redundante de suas submatrizes), é possível chegar ao final do cálculo do alinhamento. □

6.2.3 Resultados Experimentais

Foram conduzidos testes utilizando o algoritmo proposto em [1], representado pelo Algoritmo 9 (MPI), e o Algoritmo 10 (CUDA), tanto a implementação inicial quanto a coalescente, em dois diferentes ambientes. A implementação sequencial e MPI foram executadas no Ambiente 1 (*cluster* Carleton), descrito na Seção 2.2. As implementações utilizando GPU foram testadas usando o Ambiente 2 (GTX680 - UnB) descrito na Seção 2.2.

Os algoritmos foram implementados na linguagem C ANSI e os tempos de execução são apresentados em segundos. As implementações apenas calculam o valor do alinhamento ótimo. Foram implementados o alinhamento global (NW), o alinhamento global utilizando espaço linear (não armazenando a matriz inteira) e o alinhamento local (SW). Os resultados obtidos são apresentados nas próximas subseções.

6.2.4 Alinhamento Global usando Matriz

A Tabela 6.2 mostra os tempos de execução do algoritmo sequencial em uma máquina do Ambiente 1 (*cluster* Carleton), do algoritmo 9 (MPI) no Ambiente 1 (*cluster* Carleton) e do Algoritmo 10 (CUDA) no Ambiente 2 (GTX680 - UnB). Para calcular o alinhamento

entre duas sequências uma de tamanho 16384, ou 16K caracteres, e outra com 30K caracteres são necessários 1,92GB de memória para armazenar a matriz de programação dinâmica. Assim, este foi o limite de tamanho para nossos testes em virtude da memória disponível na GPU (2 GB). Os tempos dos testes usando 32 e 64 processadores do *cluster* foram, normalmente, bem piores do que o tempo usando 16 processadores. Isso provavelmente se deve à pouca computação realizada em cada processador e da alta carga de comunicação necessária entre eles.

A Figura 6.5 apresenta um gráfico comparativo dos tempos da Tabela 6.2. Pode-se notar que os tempos da implementação CUDA inicial estão muito próximos dos tempos alcançados pelo uso de 16 processadores na implementação MPI. A implementação coalescente apresenta, em média, um ganho maior que 50% em relação à implementação CUDA inicial. O número de *threads* por bloco (dim_bloco) definido para lançamento do *kernel* foi quatro e o número de blocos é dado pelo cálculo $((num\_elementos\_antidiagonal - 1)/dim\_bloco) + 1$.

Tabela 6.2: Tempos de execução (em segundos) para o alinhamento global usando matriz.

	2K x 4K	4K x 8K	8K x 16K	16K x 16K	16K x 30K
Sequencial	0,372	1,499	6,460	13,107	24,482
MPI (2 processadores)	0,299	1,194	4,835	9,733	18,141
MPI (4 processadores)	0,178	0,710	2,854	5,775	10,717
MPI (8 processadores)	0,096	0,385	1,552	3,133	5,823
MPI (16 processadores)	0,050	0,202	0,818	1,640	3,050
CUDA Inicial	0,060	0,212	0,807	1,628	3,019
CUDA Coalescente	0,037	0,099	0,299	0,521	0,938

6.2.5 Alinhamento Global usando Vetor

Modificamos o Algoritmo 9 (MPI) e o Algoritmo 10 (CUDA) para armazenar apenas parte da matriz que permita o prosseguimento do algoritmo, possibilitando a execução de testes com sequências bem maiores, com até duas sequências de tamanho 524288 caracteres (512k caracteres). A Tabela 6.3 mostra os tempos de execução destes testes. Em CUDA optamos por manter apenas três antidiagonais. Com isso a implementação coalescente, que faz uso do procedimento para cálculo de novos índices (Algoritmo 11), não se mostrou eficiente. Assim, coletamos apenas os tempos da versão CUDA inicial. O número de *threads* por bloco (dim_bloco) definido para lançamento do *kernel* foi quatro e o número de blocos é dado pelo cálculo $((num\_elementos\_antidiagonal - 1)/dim\_bloco) + 1$.

Tabela 6.3: Tempos de execução (em segundos) para o alinhamento global usando vetor.

	16K x 16K	16K x 30K	128K x 128K	256K x 256K	512K x 512K
Sequencial	11,144	20,112	712,970	2850,940	11404,720
MPI (2 processadores)	9,743	18,165	622,674	2517,372	10035,784
MPI (4 processadores)	5,743	10,849	363,263	1465,899	5919,870
MPI (8 processadores)	3,136	5,845	200,908	801,382	3229,193
MPI (16 processadores)	1,657	3,071	104,600	422,491	1678,206
MPI (32 processadores)	3,840	1,577	54,009	215,310	861,922
MPI (64 processadores)	3,423	0,811	30,180	108,768	434,398
CUDA	0,149	0,212	3,112	10,509	37,758

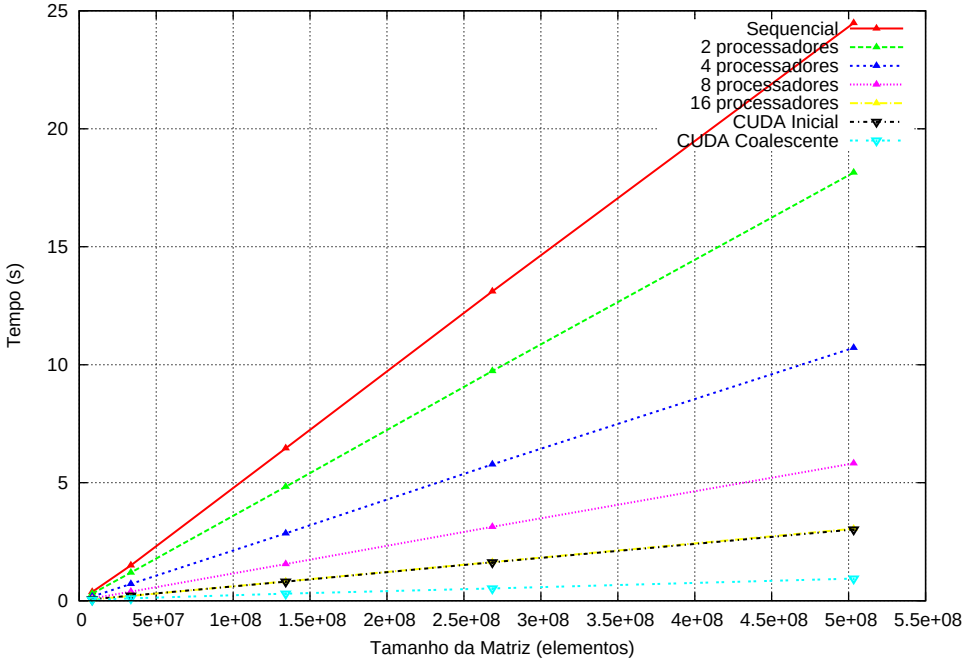


Figura 6.5: Gráfico com tempos de execução (em segundos) do alinhamento global usando matriz.

Na Figura 6.6 é possível analisar o comportamento dos tempos apresentados na tabela 6.3. Nota-se a melhora progressiva do tempo à medida que são empregados mais processadores. Nas instâncias de testes MPI com sequências a partir de 16K x 30K temos uma melhora significativa à medida que aumentamos o número de processadores empregados no cálculo da solução. A implementação CUDA apresenta os melhores resultados sendo, na média, 86% melhor que o uso de 64 processadores na implementação MPI.

6.2.6 Alinhamento Local

Uma pequena modificação foi feita no Algoritmo 9 para que fosse calculado o alinhamento local. A Tabela 6.4 mostra os tempos de execução nestes testes. O alinhamento local envolve o cálculo da matriz de programação dinâmica e a busca pelo maior valor calculado, por isso os tempos são piores que os tempos apresentados na Tabela 6.2. Os tempos dos testes da implementação MPI usando 32 e 64 processadores do Ambiente 1 (*cluster* Carleton) foram piores do que o tempo usando 16 processadores. O número de *threads* por bloco (dim_bloco) definido para lançamento do *kernel* foi quatro e o número de blocos é dado pelo cálculo $((num\_elementos\_antidiagonal - 1)/dim\_bloco) + 1$.

O gráfico da Figura 6.7 apresenta um comparativo dos tempos da Tabela 6.4. Os tempos da implementação CUDA inicial são piores do que os tempos da implementação MPI utilizando 16 processadores. Para sequências maiores o tempo da versão CUDA coalescente foi melhor que da versão MPI usando 16 processadores.

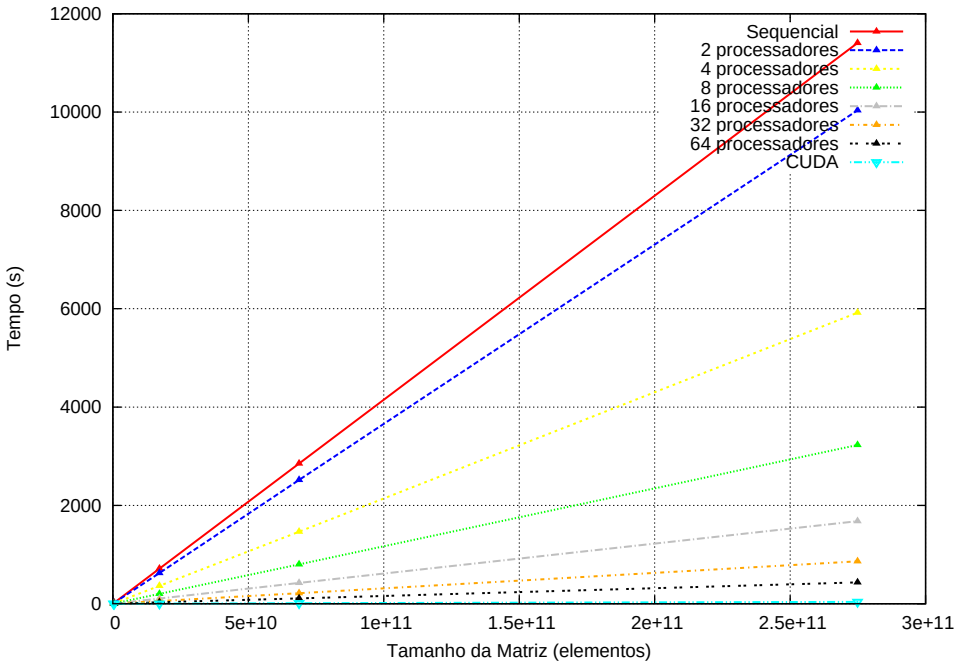


Figura 6.6: Gráfico com tempos de execução (em segundos) do alinhamento global usando vetor.

Tabela 6.4: Tempos de execução (em segundos) para o alinhamento local.

	2K x 4K	4K x 8K	8K x 16K	16K x 16K	16K x 30K
Sequencial	0,450	1,841	7,447	14,893	27,808
MPI (2 processadores)	0,365	1,463	5,893	11,776	22,085
MPI (4 processadores)	0,213	0,853	3,439	6,884	12,899
MPI (8 processadores)	0,116	0,460	1,847	3,699	6,933
MPI (16 processadores)	0,060	0,240	0,961	1,917	3,594
CUDA Inicial	0,195	0,485	1,478	2,650	4,849
CUDA Coalescente	0,140	0,353	0,782	1,195	1,916

6.3 Alinhamento Múltiplo de Sequências (AMS)

O alinhamento múltiplo de sequências (AMS) é uma técnica usada para descobrir informações funcionais, estruturais e evolutivas a respeito de sequências biológicas [17]. Ele é considerado um problema complexo e desafiador (NP-difícil) [69], visto que apresenta limitações computacionais relativas a tempo de execução e uso de memória. A necessidade de execuções mais rápidas deste tipo de problema ocorre devido ao recente crescimento da complexidade e volume de dados biológicos. Por este motivo, este problema tornou-se um bom candidato para uso de técnicas de programação paralela como *OpenMP* [11] e GPUs (*Graphics Processing Unit*) [2].

O AMS está relacionado a questões como análise filogenética, visto que é possível explicar relações evolutivas entre vários organismos fazendo o alinhamento múltiplo de suas sequências para construir árvores de filogenia. Outras aplicações do AMS são a

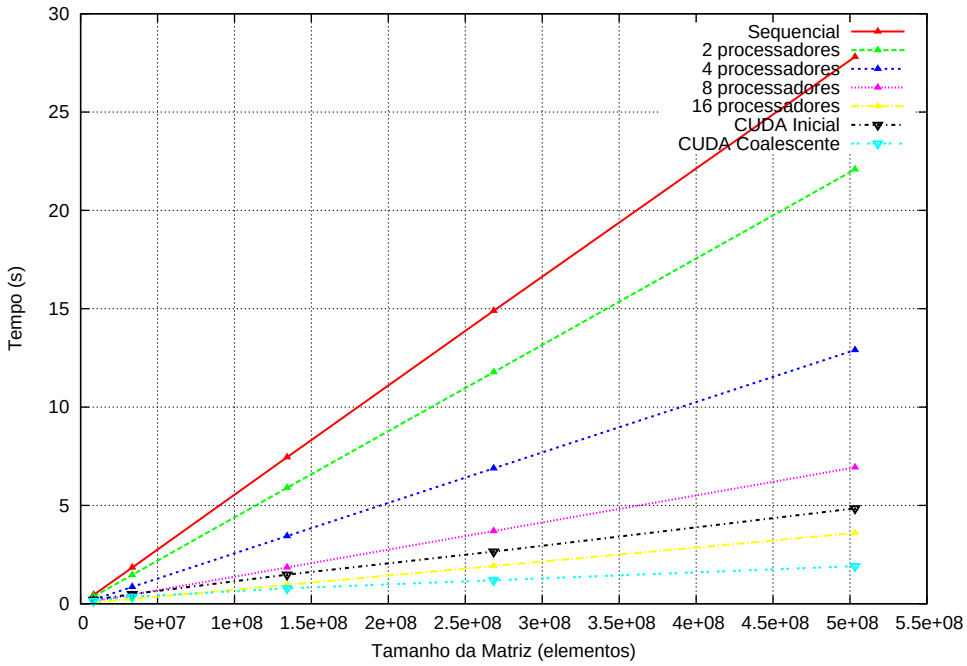


Figura 6.7: Gráfico com tempos de execução (em segundos) do alinhamento local.

identificação de *motifs* [45] e predição de estruturas secundárias e terciárias.

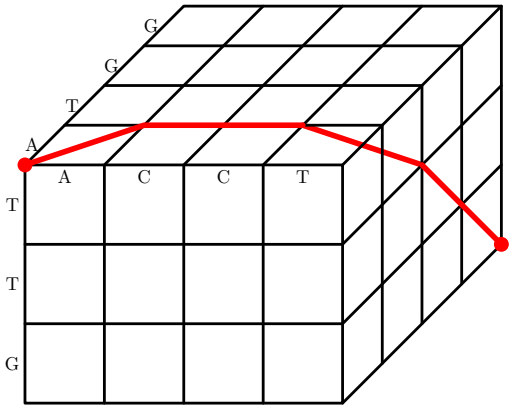
Para definir um AMS, considere um conjunto de k sequências, $s_1, \ldots, s_k$, que utilizam um mesmo alfabeto. Um AMS envolvendo $s_1, \ldots, s_k$ é obtido através da inserção de espaços nas sequências, em posições adequadas, de tal forma que todas as sequências passem a ter o mesmo tamanho [56]. A análise de similaridade entre os elementos das sequências irá determinar as melhores posições para que os espaços sejam inseridos. A representação do AMS normalmente é dada por uma matriz como da Figura 6.8 [56]. Não é permitida a existência de colunas que contenham apenas espaços.

```
- T T G C C A T T - -
A T G G C C A T T - -
A T C - C A A T T T T
A T C T T C - T T - -
A C T G A C C - - - -
```

Figura 6.8: Exemplo de alinhamento múltiplo.

O problema consiste em construir um alinhamento múltiplo de máxima pontuação (*score*), chamado de alinhamento múltiplo ótimo, dentre todos os possíveis alinhamentos. De forma simplificada, a pontuação do alinhamento múltiplo é dada pela soma das pontuações das colunas. A pontuação de cada coluna é dada pela soma dos escores de cada um dos seus pares de símbolos, de acordo com a matriz de pontuação de símbolos pré-definida. Este sistema de pontuação, chamado de SP-score (*score* baseado em soma de pares), é muito popular e amplamente utilizado [17].

Uma forma de encontrar o alinhamento múltiplo ótimo pode ser através de uma extensão do algoritmo Needleman-Wunsch (NW) [43]. Esta solução usa programação dinâmica e envolve a construção de uma matriz k-dimensional formada pelas sequências a serem alinhadas de forma a encontrar o melhor caminho entre os cantos opostos da matriz. Este caminho representa o alinhamento ótimo. A Figura 6.9 [7] mostra um pequeno exemplo para três sequências. Esta solução apresenta limitações práticas relativas ao número de sequências a serem alinhadas e seus tamanhos. Isto ocorre devido ao crescimento exponencial em relação ao número de sequências, da quantidade de memória e do número de passos de computação necessários. Para ser mais preciso, o custo total para construção da matriz multidimensional é $O(n^k)$, onde n é o tamanho da maior sequência e k é o número de sequências [8].



A - - T G G
A C C T - -
- - - T T G

Figura 6.9: Alinhamento múltiplo de três sequências (adaptada de [7].

6.3.1 Algumas Abordagens Existentes

Várias soluções para o problema de AMS já foram propostas. Estas soluções podem ser divididas em três categorias: algoritmos exatos (ou métodos exaustivos), algoritmos progressivos, e algoritmos iterativos. Notredame [45] enfatiza que apenas a minoria destes métodos descritos na literatura tem uso regular. Isto ocorre principalmente porque não existe uma solução que possa ser utilizada nas diversas situações. Há algoritmos que têm um comportamento melhor de acordo com um determinado contexto, considerando as características das sequências a serem alinhadas e o ambiente computacional a ser empregado.

A abordagem exata consiste em alinhar simultaneamente as diferentes sequências. O algoritmo exato tradicional (uma generalização do algoritmos Nedleman-Wunsch), por razões práticas (tempo e memória), não pode ser usado quando se tem mais de três sequências. Carrillo-Lipman [7] realizaram estudos sobre o problema do AMS usando visão geométrica. Eles notaram que nem todos os elementos da matriz de programação dinâmica k-dimensional são necessários para calcular o alinhamento. Consequentemente, propuseram uma heurística para reduzir o espaço de busca, tentando limitar o número

de elementos da matriz a serem calculados. O algoritmo de Carrillo-Lipman pode alinhar até dez sequências estreitamente relacionadas.

Uma das formas mais simples e eficazes de obter um AMS é usar a abordagem progressiva. O método proposto por Thompson e outros [58], chamado ClustalW, é o mais conhecido. Ele consiste de três fases. Primeiramente todas as sequências são alinhadas duas as duas (alinhamento *pairwise*) para que seja calculada a matriz de distância usando o grau de divergência entre elas. Em seguida, tendo como base a pontuação dos alinhamentos, uma árvore filogenética é construída. Finalmente, o alinhamento múltiplo é construído progressivamente usando a árvore filogenética. Existem outros métodos progressivos para o AMS como MUSCLE, apresentado em [13], T-Coffee, proposto em [47] e o alinhamento Estrela, proposto por Gusfield [17]. Neste trabalho desenvolvemos algumas soluções paralelas baseadas no algoritmo de Gusfield, descrito na seção 6.3.2.

As estratégias iterativas se baseiam na ideia de que a solução para um determinado problema pode ser calculada modificando soluções subótimas. Cada passo de modificação da solução é uma iteração. Muitos algoritmos realizam estas modificações usando programação dinâmica ou randômica empregando, por exemplo, técnicas como *simulated annealing* [32] e algoritmos genéticos [46].

Outras abordagens para resolver o problema do AMS usando métodos de inteligência artificial e algoritmos probabilísticos podem ser encontradas em [25, 37, 42].

Em virtude da alta complexidade de cálculo de grandes alinhamentos, alguns pesquisadores buscam melhorar o desempenho deste tipo de aplicação distribuindo a carga computacional entre vários processadores. Alguns dos primeiros relatos de algoritmos paralelos para o AMS foram publicados no início dos anos 2000 [27, 38, 59]. Entre as técnicas empregadas estão o uso de GPUs, computação em nuvem e computação em *grid* [2, 23, 68].

6.3.2 Algoritmo Estrela

O método proposto por Gusfield [17] é um algoritmo progressivo que compreende três passos: calcular todos os $k(k-1)/2$ alinhamentos *pairwise* entre as k sequências a serem alinhadas, encontrar a sequência centro S_c (que apresente uma menor distância em relação às demais sequências) e finalmente calcular o AMS progressivamente. Para calcular os alinhamentos *pairwise* é usado o algoritmo Needleman-Wunsch [43] descrito na seção 6.1.1.

Para encontrar a sequência centro deve-se selecionar a sequência que seja mais similar a todas as outras, ou seja, a sequência que tem o maior valor de alinhamento com as demais. Para isso deve-se preencher a matriz de distâncias com o valor dos alinhamentos *pairwise* calculados. Para exemplificar, com base nas sequências da Figura 6.8, a Tabela 6.5 mostra a matriz de distâncias resultante. Neste exemplo a sequência centro é s_1. Se houver duas ou mais sequências com o mesmo maior valor para soma de seus alinhamentos *pairwise* pode-se escolher qualquer uma.

Uma árvore, de topologia "estrela", pode ser construída colocando a sequência centro como "raiz" e as demais ligadas a esta. A Figura 6.10 mostra a árvore para as sequências do exemplo. Por este motivo o algoritmo de Gusfield é conhecido como *Algoritmo Estrela* (Algoritmo 12). Este algoritmo não produz uma solução exata, mas o valor do alinhamento

Tabela 6.5: Matriz de Distância entre cinco sequências.

	s_1	s_2	s_3	s_4	s_5	$\sum alinhamentos(s_i, s_j)$
s_1		7	-2	0	-3	2
s_2	7		-2	0	-4	1
s_3	-2	-2		0	-7	-11
s_4	0	0	0		-3	-3
s_5	-3	-4	-7	-3		-17

encontrado por ele é no mínimo metade do valor do alinhamento ótimo, ou seja, é um algoritmo de aproximação de razão 2 [16].

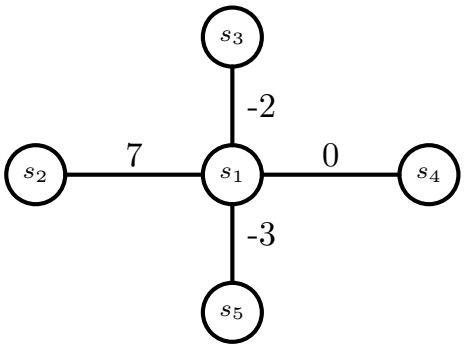


Figura 6.10: Árvore com $s1$ com sequência centro.

Algoritmo 12 Alinhamento Estrela.

Entrada: (1) Número de sequências k; (2) As k sequências.
Saída: (1) O AMS.

```
1: for 1 ≤ x ≤ k − 1 do
2:    for x + 1 ≤ y ≤ k do
3:       Calcular o alinhamento pairwise entre as sequências S_x e S_y;
4:       Salvar o valor do alinhamento entre S_x e S_y na matriz de distâncias;
5:    end for
6: end for
7: Encontrar a sequência centro S_c;
8: for 1 ≤ x ≤ k do
9:    if x ≠ c then
10:      Calcular o alinhamento pairwise entre S_c e S_x;
11:      Construir o alinhamento entre S_c e S_x;
12:      Adicionar o alinhamento S_c S_x ao AMS;
13:   end if
14: end for
```

6.3.3 Implementações Paralelas Propostas do Algoritmo Estrela

Neste trabalho foram desenvolvidas duas versões paralelas para o algoritmo estrela [17]. Visto que o número de alinhamentos *pairwise* a serem calculados é grande, decidiu-se usar

a estratégia *wavefront* [1], descrita na seção 6.1, para otimizar estes trechos da solução. A primeira implementação foi desenvolvida usando MPI para sistemas de memória distribuída. A segunda versão faz uso de CUDA para acelerar o cálculo dos alinhamentos executados em uma GPGPU. Estas implementações e os resultados dos testes conduzidos foram publicados no artigo [66].

Implementação MPI

O Algoritmo 13 mostra como os alinhamentos *pairwise* são utilizados para calcular o AMS. O primeiro passo é calcular os $k(k-1)/2$ alinhamentos *pairwise*, com participação dos p processadores, e enviar os resultados para o processador P_1, que armazena os valores e encontra a sequência centro S_c. Em seguida a sequência centro é distribuída por P_1 a todos os demais processadores. A cada iteração do próximo laço é construído o alinhamento entre a sequência centro S_c e cada uma das demais sequências (S_x). Este alinhamento S_cS_x deve ser acrescentado ao AMS por P_1. Se o alinhamento S_cS_x adiciona "espaços" (*gaps*) à sequência centro, estes devem ser propagados para o restante do AMS já construído.

Algoritmo 13 Alinhamento Estrela usando MPI.

Entrada: (1) Número de processadores p; (2) Identificação do processador i, onde $1 \leq i \leq p$; (3) Número de sequências k; (4) Arquivo que contém as k sequências.

Saída: (1) O AMS.

1: P_1 lê todas as sequências;
2: **for** $1 \leq x \leq k-1$ **do**
3: P_1 envia a sequência S_x a todos os demais processadores;
4: **for** $x+1 \leq y \leq k$ **do**
5: P_1 envia as subsequências S_y para demais processadores;
6: Algoritmo 9 (p, i, S_x, S_y);
7: P_p envia o valor do alinhamento *pairwise* entre S_x e S_y (último elemento calculado por P_p) para P_1 ;
8: **end for**
9: **end for**
10: P_1 encontra a sequência centro S_c;
11: P_1 envia a sequência centro S_c para todos os demais processadores;
12: **for** $1 \leq x \leq k$ **do**
13: **if** $x \neq c$ **then**
14: P_1 envia as subsequências de S_x;
15: Algoritmo 9 (p, i, S_c, S_x);
16: Cada processador constrói parte do alinhamento entre S_cS_x e envia o alinhamento para P_1;
17: P_1 adiciona o alinhamento entre S_cS_x ao AMS;
18: **end if**
19: **end for**

Teorema 11. *Algoritmo 13 usa $O(k^2p)$ rodadas de comunicação com tempo de computação sequencial $O(k^2\frac{n^2}{p})$ em cada processador, sendo k o número de sequências, n o*

tamanho das sequências a serem alinhadas e p o número de processadores.

Demonstração. O Algoritmo 13 possui três partes: calcular a pontuação do alinhamento *pairwise* entre todas as sequências, encontrar a sequência centro e construir, progressivamente, o alinhamento entre S_c e as demais sequências, adicionando-o ao AMS.

Para executar a primeira parte é preciso calcular $(k^2 - k)/2$ alinhamentos *pairwise*. Em [1] os autores demonstraram que cada alinhamento *pairwise*, usando a técnica de *wavefront*, precisa de $2p-2$ rodadas de comunicação com tempo de computação sequencial $O(\frac{mn}{p})$ em cada processador, sendo m e n o tamanho das sequências a serem alinhadas. Desta forma, o primeiro passo do Algoritmo 13 precisará de $((k^2 - k)/2)(2p - 2)$ rodadas de comunicação, onde cada processador usará no total tempo $O(k^2 \frac{n^2}{p})$ para a computação sequencial, considerando n o tamanho das sequências a serem alinhadas.

A segunda parte, que consiste em encontrar a sequência centro S_c, analisando a matriz de distância, é sequencialmente executada pelo processador P_1 em tempo $O(k^2)$.

Na terceira parte a sequência centro S_c deve ser alinhada com todas as demais sequências. Isto representa $k - 1$ alinhamentos *pairwise* com as mesmas $2p - 2$ rodadas de comunicação e tempo de computação sequencial $O(\frac{n^2}{p})$ em cada processador. Nesta parte ainda ocorre a construção do alinhamento múltiplo, o que representa, para cada alinhamento *pairwise*, mais p rodadas de comunicação com tempo de computação sequencial $O(\frac{n^2}{p})$ para cada processador. Assim, a terceira parte consiste de $(k - 1)(2p - 2)$ rodadas de comunicação com tempo de computação sequencial $O(k\frac{n^2}{p})$ mais $(k-1)(p-1)$ rodadas de comunicação com tempo de computação sequencial $O(\frac{n^2}{p})$.

Considerando as três partes, temos $O(k^2p)$ rodadas de comunicação na primeira, $O(1)$ na segunda e $O(kp)$ rodadas na terceira. O tempo de computação sequencial é $O(k^2\frac{n^2}{p})$ na primeira parte, $O(k^2)$ na segunda e $O(k\frac{n^2}{p})$ na terceira. Portanto, podemos concluir que o algoritmo paralelo usa $O(k^2p)$ rodadas de comunicação com tempo de computação sequencial $O(k^2\frac{n^2}{p})$ para cada processador. □

Implementações CUDA

As implementações CUDA do algoritmo estrela paralelizam os cálculos dos alinhamentos *pairwise* usando as abordagens descritas na seção 6.2.1.

O algoritmo estrela usando CUDA (Algoritmo 14) funciona da seguinte forma: o *host* e o *device* são utilizados para calcular cada um dos alinhamentos *pairwise* conforme descrito no Algoritmo 10. Em seguida o *host* define a sequência centro S_c. No último passo são calculados e construídos os alinhamentos *pairwise* de S_c e as demais sequências. Cada alinhamento com S_c construído deve ser adicionado ao AMS.

Algoritmo 14 Alinhamento Estrela usando GPGPU.

Entrada: (1) Número de sequências k; (2) Arquivo que contém as k sequências.
Saída: (1) O AMS.

1: Host lê todas as sequências;
2: **for** $1 \leq x \leq k-1$ **do**
3: Realizar copia da sequência S_x para o *Device*;
4: **for** $x+1 \leq y \leq k$ **do**
5: Realizar copia da sequência S_y para o *Device*
6: *Host* inicia cálculo do alinhamento pairwise entre S_x e S_y;
7: *Device* continua cálculo do alinhamento pairwise entre S_x e S_y;
8: *Host* finaliza cálculo do alinhamento pairwise entre S_x e S_y;
9: *Host* salva o valor (*score*) do alinhamento pairwise entre S_x e S_y na matriz de distância;
10: **end for**
11: **end for**
12: *Host* analisa a matriz de distância e encontra a sequência centro S_c;
13: Realizar copia da sequência S_c para o *Device*;
14: **for** $1 \leq x \leq k$ **do**
15: **if** $x \neq c$ **then**
16: Realizar copia da sequência S_x para o *Device*;
17: *Host* inicia cálculo do alinhamento pairwise entre S_c e S_x;
18: *Device* continua cálculo do alinhamento pairwise entre S_c e S_x;
19: *Host* finaliza cálculo do alinhamento pairwise entre S_c e S_x;
20: *Host* constrói o alinhamento S_cS_x;
21: *Host* adiciona S_cS_x ao AMS;
22: **end if**
23: **end for**

6.3.4 Resultados Experimentais

Os algoritmos 12, 13 e 14 foram executados em dois diferentes ambientes. A implementação MPI foi executada no Ambiente 1 (*cluster* Carleton) (descrito na Seção 2.2). A implementação sequencial e as implementações utilizando GPU foram testadas usando o Ambiente 2 (GTX680 - UnB) (também descrito na Seção 2.2).

Os algoritmos foram implementados na linguagem C ANSI e os tempos de execução são apresentados em segundos, considerando o tempo usado para distribuição de dados e construção do AMS. Os experimentos conduzidos tiveram instâncias de dados de entrada com 8, 10, 12 e 14 sequências com tamanhos variando de 1024 a 16384 caracteres. Em todos os conjuntos de entrada as sequências têm o mesmo tamanho, exceto para os conjuntos com sequências de 16384 caracteres. O rótulo 16384 (a) corresponde a testes contendo uma sequência de tamanho 16384 e as demais com 8192 caracteres. O rótulo 16384 (b) corresponde a testes onde todas as sequências do conjunto têm 16384 caracteres.

A Tabela 6.6 mostra o tempo de execução do Algoritmo 13 no *cluster*. Os tempos apresentados não levam em conta a leitura do dados de entrada. Nas instâncias de teste

menores (8 sequências com 1024 caracteres) foi observado que o aumento do número de processadores leva a uma diminuição do tempo de execução. No entanto, ao usar 64 processadores o tempo de execução aumenta, devido ao número de rodadas de comunicação que se faz necessário. Por outro lado, com um volume de dados maior, o tempo de execução alcança uma redução significativa, como visto na Tabela 6.6. A Figura 6.11 apresenta um gráfico com os tempos de execução da Tabela 6.6, para os testes com todas as sequências de tamanho 16384 (16384 (b)). É possível observar o ganho obtido quando aumenta-se o número de processadores empregados.

Tabela 6.6: Tempos de execução (em segundos) usando o *cluster* para diferentes instâncias de dados de entrada.

Tam, Seq,	Núm, de Seq,	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$	$P = 32$	$P = 64$
1024	8	1,779	1,331	0,822	0,463	0,269	0,196	0,803
	10	2,715	2,141	1,276	0,717	0,405	0,300	0,398
	12	3,943	2,982	1,866	1,019	0,588	0,471	0,487
	14	5,735	3,935	2,481	1,402	0,796	0,577	0,665
4096	8	30,741	21,788	13,151	7,409	6,017	5,192	4,326
	10	45,022	34,131	20,401	11,233	9,096	8,802	5,126
	12	64,348	47,874	28,614	15,929	11,896	8,006	5,322
	14	92,526	64,718	38,803	22,063	13,960	9,934	7,182
8192	8	125,253	90,758	52,209	28,961	18,337	11,637	7,963
	10	181,239	135,731	81,796	45,565	28,115	16,388	10,948
	12	258,999	201,233	115,658	65,570	39,115	21,981	14,605
	14	375,046	271,899	152,109	88,441	52,336	29,926	19,407
16384 (a)	8	154,527	115,399	64,645	35,362	22,970	11,817	9,513
	10	214,526	166,007	97,864	54,472	32,158	19,957	13,173
	12	299,001	239,347	139,494	77,189	47,121	26,733	15,295
	14	429,346	317,902	183,771	101,305	59,804	35,033	23,248
16384 (b)	8	494,873	377,287	210,823	116,237	66,163	37,793	19,652
	10	726,063	572,331	324,809	182,448	101,749	58,121	30,772
	12	1014,625	834,409	478,619	262,976	143,010	82,851	47,708
	14	1498,015	1127,618	632,952	347,858	199,098	111,323	69,959

Um achado curioso para as implementações CUDA é que o tamanho atribuído ao bloco de threads do *device* influencia muito nos resultados dos experimentos, que também variam dependendo do número mínimo de elementos da antidiagonal a ser calculada pelo *device*.

Para a implementação CUDA sem usar coalescência de memória, implementação inicial, foram realizados testes com tamanho de bloco igual a 4, 8, 16 e 32, e tamanho mínimo de antidiagonal igual a 32, 64, 128 e 256. O coeficiente de variação para estes testes foi entre 9.6% e 16.3%. Testes com blocos de tamanho maior não foram realizados visto que os resultados estavam piorando a medida que o tamanho crescia. Utilizando a implementação inicial, na maioria dos casos, os melhores resultados foram alcançados com tamanho de bloco igual a 4 e tamanho mínimo para antidiagonal igual a 128.

Para a implementação coalescente, os testes foram executados para bloco de tamanhos 4, 8, 16, 32, 64, 128, 256 e 512, e tamanho mínimo de antidiagonal igual a 32, 64, 128 e 256. O coeficiente de variação para estes testes ficaram entre 7.4% e 34.9%. Utilizando a implementação coalescente, na maioria dos casos, os melhores resultados foram alcançados com tamanho de bloco igual a 128 e tamanho mínimo para antidiagonal igual a 64. A

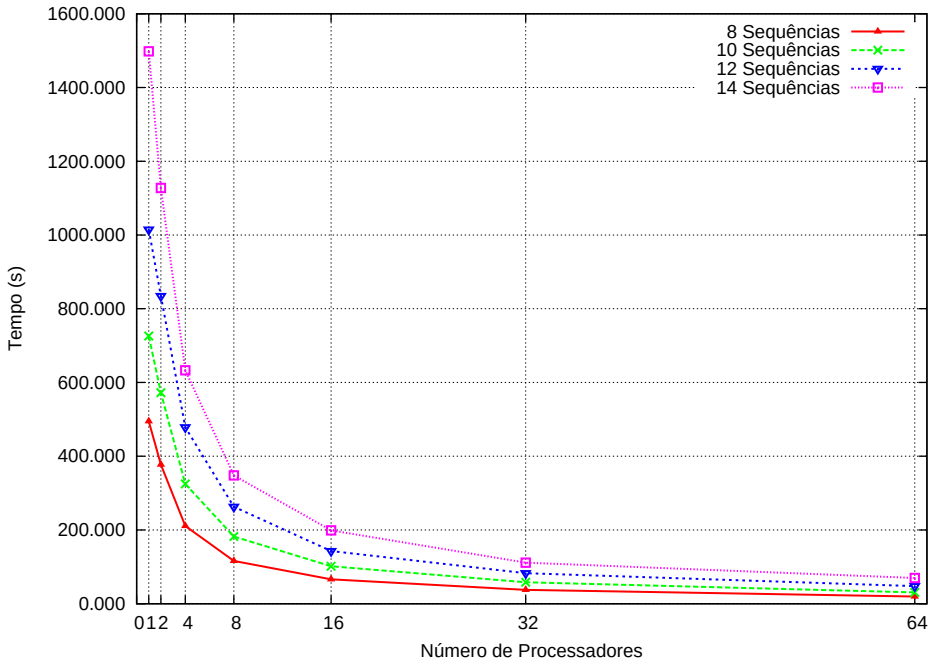


Figura 6.11: Gráfico com tempos de execução do Algoritmo 13 no *cluster* usando MPI.

mudança do tamanho do bloco influencia na quantidade de *warps* ativos, que acaba por impactar no desempenho da aplicação.

A Tabela 6.7 expõe os tempos de execução de testes realizados utilizando o *desktop*. Os tempos apresentados não levam em conta a leitura do dados de entrada. Os dados apresentados resultam da média dos tempos de execução, usando os melhores valores para tamanho de bloco e para tamanho mínimo de antidiagonal. Para estes casos de teste, o coeficiente máximo de variação observado foi 0.7%. A Figura 6.12 mostra os resultados presentes na Tabela 6.7 para alinhar sequências com 16384 caracteres (16384 (b)). Como esperado, o tempo de execução para a implementação sequencial é muito maior que os demais. Comparando os resultados da implementação coalescente com da implementação inicial nota-se um ganho de até 60% em alguns casos.

A Figura 6.13 apresenta de forma comparativa os tempos dos testes para alinhar 14 sequências, sendo todas de tamanho 16384, onde CUDA1 corresponde à implementação inicial, sem coalescência, e CUDA2 à implementação coalescente. Considerando as abordagens MPI e CUDA, CUDA1 apresenta resultados melhores que a implementação MPI com, entorno de, 16 processadores disponíveis. CUDA2 apresenta resultados um pouco melhor do alcançado usando MPI em 64 processadores. Considerando o custo dos dois ambientes (um *cluster* e uma máquina com uma GPU), o uso de GPGPU provou ser uma solução muito boa.

Tabela 6.7: Tempos de execução (em segundos) no desktop.

Implementação	Núm, de Seq,	1024	4096	8192	16384 (a)	16384 (b)
Sequencial	8	0,692	13,069	51,959	62,793	215,761
	10	1,015	18,724	75,143	89,540	306,090
	12	1,547	28,809	113,474	131,424	479,618
	14	2,072	39,180	154,700	175,554	651,858
CUDA Inicial	8	0,306	3,925	14,975	20,727	59,265
	10	0,465	6,055	23,119	30,392	91,595
	12	0,659	8,655	33,021	41,771	130,886
	14	0,887	11,692	44,722	54,860	176,793
CUDA Coalescente	8	0,303	2,047	6,239	8,116	20,110
	10	0,460	3,158	9,680	12,171	31,174
	12	0,651	4,491	13,857	16,743	44,706
	14	0,875	6,078	18,827	22,218	60,450

6.4 Contribuições

Dentre as contribuições alcançadas com o trabalho descrito neste capítulo tivemos:

- implementações paralelas do algoritmo estrela [17], usando MPI e CUDA;
- implementações paralelas do algoritmo para alinhamento *pairwise* de sequências, usando MPI e CUDA;
- proposta de alinhamento *pairwise* tolerante a falhas; e
- publicação de um artigo intitulado *Efficient parallel implementations of multiple sequence alignment using BSP/CGM model* no *International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM), realizado no âmbito do *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP) em 2014.

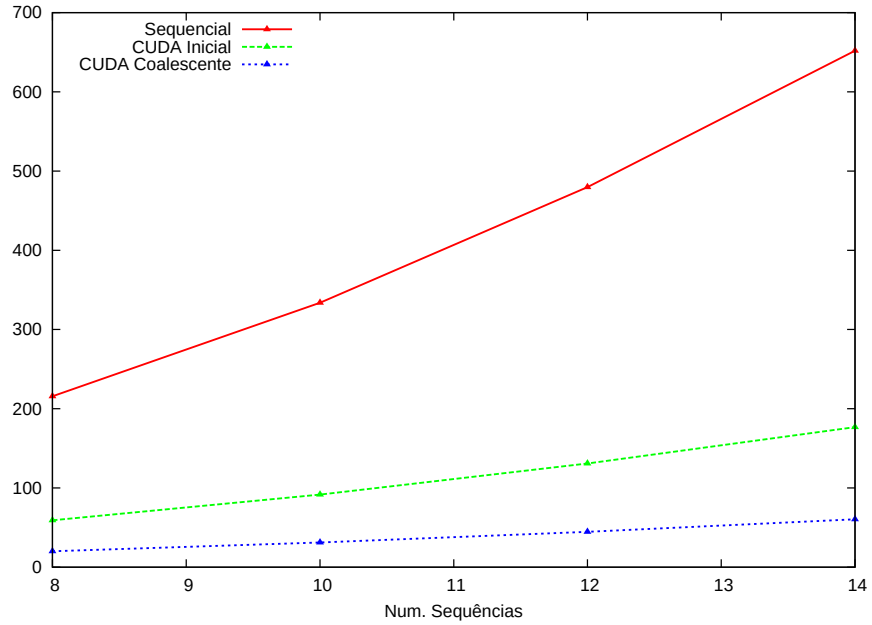


Figura 6.12: Gráfico com tempos de execução (em segundos) do Algoritmo 14.

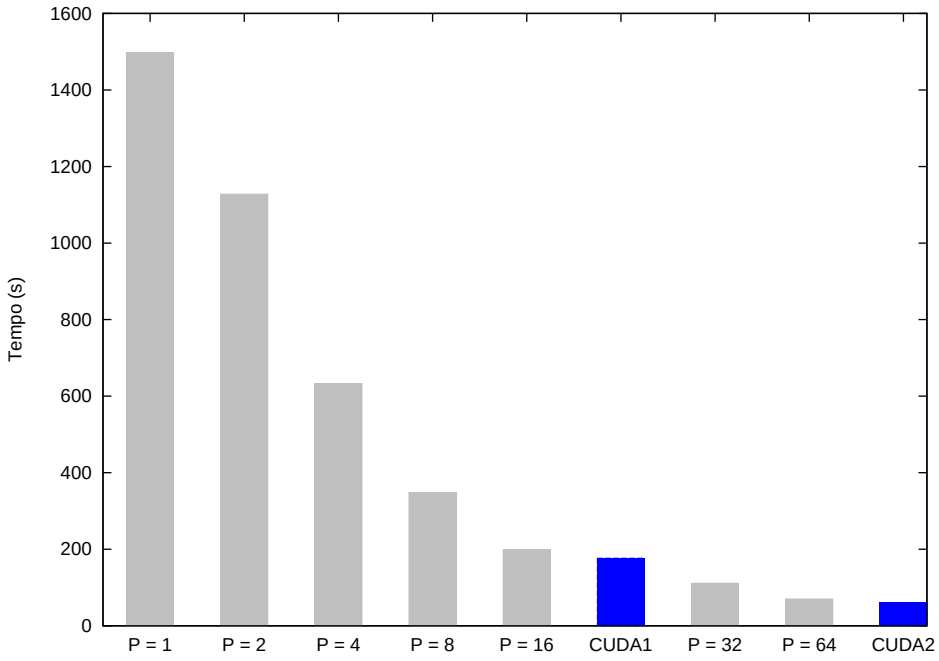


Figura 6.13: Gráfico com tempos de execução (em segundos) no *cluster* e usando a GPU com 14 sequências de maior tamanho.

Capítulo 7

Conclusões

Neste trabalho, foram apresentas soluções paralelas eficientes para duas classes de problemas: alinhamento de sequências e árvore geradora. Para propor estas soluções utilizou-se o modelo BSP/CGM. O modelo BSP/CGM continua se mostrado bastante adequado para o projeto de algoritmos paralelos, principalmente os que utilizam muita comunicação entre os processadores. Além disso, os algoritmos projetados nesse modelo têm obtido bons *speedups* quando implementados em máquinas paralelas reais.

Um importante resultado deste trabalho foi o projeto e implementação de algoritmos paralelos para o cálculo da árvore geradora e árvore geradora mínima, fazendo uso de um grafo bipartido correspondente ao grafo de entrada. Os algoritmos propostos são baseados no trabalho de Cáceres et al. [4], que não utiliza *list ranking*, mas considera como entrada um grafo bipartido e realiza ordenações do conjunto de arestas. Nos algoritmos propostos neste trabalho, descritos no Capítulo 3, o primeiro passo é criar um grafo bipartido correspondente ao grafo de entrada, que tem duas vezes o número de arestas do grafo original. Não é preciso ordenar as arestas, apenas realizar a seleção da menor. O desempenho destes algoritmos melhora quando tem-se grafos de entrada grandes, com mais de 3.000.000 de arestas, aproximadamente, em virtude do custo da geração do grafo bipartido, que, para entradas maiores, acaba se diluindo no restante do tempo de processamento.

Visando melhorar o desempenho, foi projetada uma extensão dos algoritmos que não necessita da criação do grafo bipartido, trabalhando diretamente com as arestas do grafo original. A ideia do algoritmo é semelhante a do algoritmo de Borůvka [3] mas usa o conceito de esteio e de aresta zero-diferença, tendo um critério de parada diferente. Além disso, nosso algoritmo é originalmente paralelo e não sequencial. Essa extensão também permitiu o uso de menos memória, visto que não precisa dobrar o número de arestas do grafo de trabalho.

Para demonstrar a eficiência dos algoritmos propostos, foram desenvolvidas e testadas implementações CUDA e para CPU, comparando os tempos obtidos entre si e com resultados disponíveis na literatura. Utilizou-se um gerador de grafos aleatórios para analisar a escalabilidade da implementação. Também foram realizados testes utilizando como entrada os grafos disponibilizados pelo novo desafio DIMACS (9DIMACS). Os tempos e *speedups* obtidos são competitivos, e mostram que o modelo BSP/CGM é apropriado para o projeto e desenvolvimento de algoritmos paralelos para máquinas paralelas reais.

Considerando o problema de alinhamento de sequências, foram propostas implemen-

tações paralelas usando MPI e CUDA do algoritmo estrela [17] para alinhamento múltiplo de sequências e do algoritmo de Needleman-Wunsch [43] para o alinhamento *pairwise* de sequências. Foram desenvolvidas implementações usando tanto MPI quanto CUDA. As implementações apresentaram bom desempenho principalmente no ambiente com GPGPU. Foi desenvolvida ainda uma proposta de alinhamento *pairwise* tolerante a falhas.

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 ainda 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, e outros ambientes computacionais paralelos.

Referências Bibliográficas

[1] C. Alves, E. Cáceres, F. Dehne, and S. Song. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. In International Conference on Computational Science and its Applications (ICCSA 2003), volume 2668, pages 249–258. Springer-Verlag Berlin Heidelberg, 2003.

[2] J. Blazewicz, W. Frohmberg, M. Kierzynka, and P. Wojciechowski. G-msa - a gpu-based, fast and accurate algorithm for multiple sequence alignment. J. Parallel Distrib. Comput., 73(1):32–41, January 2013.

[3] O. Borůvka. On a minimal problem. Prace Moravské Pridovedecké Spodecnosti, 3:37–58, 1926.

[4] E. N. Cáceres, F. Dehne, H. Mongelli, S. W. Song, and J. L. Szwarcfiter. A coarse-grained parallel algorithm for spanning tree and connected components. In Proceedings of Euro-Par 2004. Lecture Notes in Computer Science, volume 3149, pages 828–831. Springer Berlin Heidelberg, 2004.

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

[6] Andrew Goldberg Camil Demetrescu and David Johnson. 9th DIMACS Implementation Challenge - Shortest Paths, 2006 (accessed June, 2017). http://www.dis.uniroma1.it/challenge9/.

[7] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. SIAM J. Appl. Math., 48(5):1073–1082, October 1988.

[8] S.C. Chan, A.K.C. Wong, and D.K.T. Chiu. A survey of multiple sequence comparison methods. Bull. Math. Biol., 54:563–598, 1992.

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

[10] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In Proceedings of the Ninth Annual Symposium on Computational Geometry, SCG '93, pages 298–307, New York, NY, USA, 1993. ACM.

[11] X. Deng, E. Li, J. Shan, and W. Chen. Parallel implementation and performance characterization of muscle. In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 2006.

[12] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge university press, 1998.

[13] R. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Research, 32(5):1792–1797, March 2004.

[14] O. Gotoh. An improved algorithm for matching biological sequences. Journal of Molecular Biology, 162(3):705–708, 1982.

[15] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. IEEE Ann. Hist. Comput., 7(1):43–57, January 1985.

[16] D. Gusfield. Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds. Bulletin of Mathematical Biology, 55(1):141–154, 1993.

[17] D. Gusfield. Algorithms on Strings, Trees and Sequences, chapter 14. Cambridge University Press, 1997.

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

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

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

[21] R. Johnsonbaugh and M. Kalin. A graph generation software package. Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education, 23(1):151–154, March 1991.

[22] N. C. Jones and P. Pevzner. An introduction to bioinformatics algorithms. MIT press, 2004.

[23] S. Jung, J. Na, C. Choi, F. Nazareno, I. Jung, W. Cho, M. Tang, and S. Jun. A private cloud system for web-based high-performance multiple sequence alignment services. In Intelligent Systems Modelling Simulation (ISMS), 2013 4th International Conference on, pages 143–147, 2013.

[24] A. Khajeh-Saeed, S. Poole, and J. B. Perot. Acceleration of the smith–waterman algorithm using single and multiple graphics processors. Journal of Computational Physics, 229(11):4247–4258, 2010.

[25] M. Kim. Conditional alignment random fields for multiple motion sequence alignment. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(11):2803–2809, 2013.

[26] D. Kirk and W. Hwu. Programming Massively Parallel Processors. M. Kaufmann, Waltham, MA, second edition, 2013.

[27] J. Kleinjung, N. Douglas, and J. Heringa. Parallelized multiple alignment. Bioinformatics, 18(9):1270–1271, 2002.

[28] D. Kozen. The Design and Analysis of Algorithms. Springer, 1992.

[29] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society, 7(1):48–50, February 1956.

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

[31] A. C. Lima, R. G. Branco, S. Ferraz, E. N. Cáceres, R. R. A. Gaioso, W. S. Martins, and S. W. Song. 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, 2016.

[32] A. Lukashin, J. Engelbrecht, and S. Brunak. Multiple alignment using simulated annealing: branch point definition in human mRNA splicing. Nucleic Acids Research, 20(10):2511–2516, 1992.

[33] A. Mamun and S. Rajasekaran. An efficient minimum spanning tree algorithm. In Computers and Communication (ISCC), 2016 IEEE Symposium on, pages 1047–1052. IEEE, 2016.

[34] S. Manavski and G. Valle. Cuda compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics, 9(Suppl 2), 2008.

[35] S. Manoochehri, B. Goodarzi, and D. Goswami. An Efficient Transaction-Based GPU Implementation of Minimum Spanning Forest Algorithm. 2017 International Conference on High Performance Computing & Simulation (HPCS), pages 643–650, 2017.

[36] M. Mareš. The saga of minimum spanning trees. Computer Science Review, 2(3):165–221, December 2008.

[37] L. M. O. Matos, D. Pratas, and A. J. Pinho. A Compression Model for DNA Multiple Sequence Alignment Blocks. IEEE Transactions on Information Theory, 59(5):3189–3198, May 2013.

[38] D. Mikhailov, H. Cofer, and R. Gomperts. Performance optimization of clustalw: parallel clustalw, ht clustal, and multiclustal. White Paper, 2001.

[39] D. Mount. Bioinformatics: Sequence and Genome Analysis, chapter 4. Cold Spring Harbor Laboratory Press, New York, second edition, 2004.

[40] E. W. Myers and W. Miller. Optimal alignments in linear space. Computer Applications in the Biosciences: CABIOS, 4(1):11–17, 1988.

[41] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on GPUs. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 147–156, 2013.

[42] F. Naznin, R. Sarker, and D. Essam. Progressive Alignment Method Using Genetic Algorithm for Multiple Sequence Alignment. IEEE Transactions on Evolutionary Computation, 16(5):615–631, October 2012.

[43] S. B. Needleman and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology, 48(3):443–453, 1970.

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

[45] C. Notredame. Recent progresses in multiple sequence alignment: a survey. Pharmacogenomics, 3:131–144, January 2002.

[46] C. Notredame and D. G. Higgins. SAGA: sequence alignment by genetic algorithm. Nucleic Acids Research, 24(8):1515–1524, 1996.

[47] C. Notredame, D. G. Higgins, and J. Heringa1. T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment. J. Mol. Biol., 302:205–217, September 2000.

[48] NVIDIA Corporation. CUDA C Programming Guide 6.5, 2014.

[49] V. Osipov, P. Sanders, and J. Singler. The filter-kruskal minimum spanning tree algorithm. In Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX), pages 52–61. Society for Industrial and Applied Mathematics (SIAM), 2009.

[50] P. Pacheco. An Introduction to Parallel Programming. Morgan Kaufmann, Burlington, MA, 2011.

[51] D. Patterson. The trouble with multicore. IEEE Spectrum, 47(7):28–32, July 2010.

[52] R. C. Prim. Shortest connection networks and some generalizations. Bell Labs Technical Journal, 36(6):1389–1401, November 1957.

[53] J. H. Reif. Depth-first search is inherently sequential. Information Processing Letters, 20(5):229–234, June 1985.

[54] E. F. O. Sandes and A. C. M. A. de Melo. Cudalign: Using gpu to accelerate the comparison of megabase genomic sequences. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, pages 137–146. ACM, 2010.

[55] E. F. O. Sandes and A. C. M. A. de Melo. Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences using GPU. IEEE Transactions on Parallel and Distributed Systems, 24(5):1009–1021, May 2013.

[56] J. C. Setubal and J. Meidanis. Introduction to Computational Molecular Biology. PWS Publishing, Boston, MA, January 1997.

[57] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. Journal of Molecular Biology, 147(1):195–197, 1981.

[58] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Research, 22(22):4673–4680, November 1994.

[59] O. Trelles. On the parallelisation of bioinformatics applications. Brief. Bioinform., 2:181–194, 2001.

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

[61] L. G. Valiant. A bridging model for multi-core computing. Journal of Computer and System Sciences, 77(1):154–166, January 2011.

[62] J. F. A Vasconcellos, E. N. Cáceres, H. Mongelli, and S. W. Song. Algoritmo paralelo para Árvore geradora usando GPU. In Proceedings of XVIII Simpósio em Sistemas Computacionais de Alto Desempenho, WSCAD'17, pages 292–303, 2017.

[63] J. F. A. Vasconcellos, E. N. Cáceres, H. Mongelli, and S. W. Song. A parallel algorithm for minimum spanning tree on GPU. In Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2017 International Symposium on, pages 67–72. IEEE, 2017.

[64] J. F. A. Vasconcellos, E. N. Cáceres, H. Mongelli, and S. W. Song. A new efficient parallel algorithm for minimum spanning tree. In Computer Architecture and High Performance Computing (SBAC-PAD), 2018 International Symposium on, pages 107–114. IEEE, 2018.

[65] J. F. A. Vasconcellos, E. N. Cáceres, Song S. W. Dehne F. Mongelli, H., and J. L. Szwarcfiter. New BSP/CGM algorithms for spanning trees. The International Journal of High Performance Computing Applications, October 2018.

[66] J. F. A. Vasconcellos, C. Nishibe, N. F. Almeida, and E. N. Cáceres. Efficient Parallel Implementations of Multiple Sequence Alignment using BSP/CGM Model. In Proceedings of Programming Models and Applications on Multicores and Manycores, PMAM'14, pages 103:103–103:110, New York, NY, USA, 2014. ACM.

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

[68] L. Vinh, T. Lang, N. Du, and V. Chau. Multiple Sequence Alignment on the Grid Computing using Cache Technique. International Journal of Computer Science and Telecommunications, 3(7):46–51, 2012.

[69] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. Journal of Computational Biology, 1(4):337–348, 1994.