UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

VINÍCIUS DE SOUSA COELHO

Um estudo aplicado de paralelismo para o problema do subgrafo planar de peso máximo

Goiânia
2018

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[X] Dissertação** **[ ] Tese**

2. Identificação da Tese ou Dissertação:

Nome completo do autor: Vinícius de Sousa Coelho

Título do trabalho: Um estudo aplicado de paralelismo para o problema do subrafo planar de peso máximo

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Vinícius de Sousa Coelho

Assinatura do(a) autor(a)[2]

Ciente e de acordo:

Wellington Santos Martins

Assinatura do(a) orientador(a)[2] Data: 20 / 05 / 2018

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

Versão atualizada em setembro de 2017.

[2] A assinatura deve ser escaneada.

VINÍCIUS DE SOUSA COELHO

Um estudo aplicado de paralelismo para o problema do subgrafo planar de peso máximo

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Programa de Pós-Graduação em Ciência da Computação – PPGCC-UFG.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Wellington Santos Martins

Co-Orientador: Prof. Dr. Leslie Richard Foulds

Goiânia
2018

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

de Sousa Coelho, Vinicius
Um estudo aplicado de paralelismo para o problema do subgrafo planar de peso máximo [manuscrito] / Vinicius de Sousa Coelho. - 2018.
LXIX, 69 f.: il.

Orientador: Prof. Dr. Wellington Santos Martins; co-orientador Dr. Leslie Richard Foulds.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2018.
Bibliografia. Apêndice.
Inclui siglas, algoritmos, lista de figuras, lista de tabelas.

1. Planaridade. 2. MWPSP. 3. Paralelismo. 4. CUDA. I. Santos Martins, Wellington, orient. II. Título.

CDU 004

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UFG

ATA Nº 02/2018

ATA DA SESSÃO DE JULGAMENTO DA DISSERTAÇÃO DE MESTRADO DE VINÍCIUS DE SOUSA COELHO

Aos vinte e sete dias do mês de abril de dois mil e dezoito, às catorze horas, na sala 150 do Instituto de Informática da Universidade Federal de Goiás, Campus Samambaia, reuniu-se a banca examinadora designada na forma regimental pela Coordenação do Curso para julgar a dissertação de mestrado intitulada **"Um estudo aplicado de paralelismo para o problema do subgrafo planar de peso máximo"**, apresentada pelo aluno Vinícius de Sousa Coelho como parte dos requisitos necessários à obtenção do grau de Mestre em Ciência da Computação, área de concentração Ciência da Computação. A banca examinadora foi presidida pelo orientador do trabalho de dissertação, Professor Doutor Wellington Santos Martins (INF/UFG), tendo como membros os Professores Doutores Leslie Richard Foulds (INF/UFG – coorientador), Elisângela Silva Dias (INF/UFG) e Lúcia M. A. Drummond (IC/UFF). A professora Lúcia M. A. Drummond participou à distância por videoconferência. Aberta a sessão, o candidato expôs seu trabalho. Em seguida, o aluno foi arguido pelos membros da banca e:

(X) tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **aprovação** do candidato, sem restrições.

( ) não tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **reprovação** do candidato.

Os trabalhos foram encerrados às 16 horas. Nos termos do Regulamento Geral dos Cursos de Pós-Graduação desta Universidade, lavrou-se a presente ata que, lida e julgada conforme, segue assinada pelos membros da banca examinadora.

Prof. Dr. Wellington Santos Martins _____

Prof. Dr. Leslie Richard Foulds _____

Profa. Dra. Elisângela Silva Dias _____

Profa. Dra. Lúcia M. A. Drummond _____

Vinícius de Sousa Coelho

Graduou–se em Ciência da Computação na UFG - Universidade Federal de Goiás. Durante sua graduação, foi estagiário na DRF – Delegacia da Receita Federal, bolsista do CNPq pelo projeto Ciências sem Fronteiras no Canadá, finalista da Maratona de Programação – Etapa Nacional (2015), e bolsista da PROEC pelo projeto Programa de Treinamento para OBI – Olimpíada Brasileira de Informática. Atualmente, atua como Engenheiro de Software na SAP Labs Latin America.

Dedico esse trabalho aos meus pais, Amaro Martins Coelho e Maria Aparecida Tierte de Sousa, que me apoiaram em todas as escolhas feitas até o presente momento.

Agradecimentos

Agradeço aos meus pais que, apesar das adversidades da vida, fizeram sempre o possível para me proporcionar uma vida digna. Sou grato por estarem sempre presentes e por me apoiarem nas decisões que tomei.

Agradeço ao meu orientador prof. Dr. Wellington Santos Martins, por ter me aceitado como orientando e por me fornecer as ferramentas e conhecimentos necessários para o desenvolvimento deste trabalho.

Agradeço ao prof. Dr. Humberto José Longo, por proporcionar um direcionamento no graduação. A Maratona de Programação foi um elemento essencial na moldagem da pessoa que hoje sou.

Agradeço ao prof. Dr. Leslie Foulds, que impulsionou esta pesquisa, sempre puxando discussões relevantes e esclarecendo quaisquer dúvidas relacionadas ao tema.

Agradeço ao Me. Mateus Ferreira e Freitas, pela grande ajuda e contribuição a este trabalho, dando dicas úteis na criação do código executado na GPU.

Agradeço aos meus amigos Paulo Cézar Pereira Costa, Welton Cardoso do Carmo e Carlos Alexandre Xavier da Silva por me ajudarem nos diversos problemas de programação e pela paciência que tiveram.

Agradeço à Thaís Oliveira Mombach, por ser uma ótima companheira e estar sempre comigo nos momentos bons e, principalmente, nos que mais precisei.

Agradeço à minha irmã Myllena de Sousa Coelho pelo apoio, carinho e companherismo.

Agradeço aos meus amigos, tanto os de longa data quanto os que conheci ao longo de toda a graduação. Vocês foram essenciais para manter a minha sanidade durante todo este tempo.

Life is a mystery to be lived, not a problem to be solved.

Sarvepalli Radhakrishnan,
Genius - in Their Own Words: The Intellectual Journeys of Seven Great 20th Century Thinkers.

Resumo

de Sousa Coelho, Vinícius. **Um estudo aplicado de paralelismo para o problema do subgrafo planar de peso máximo**. Goiânia, 2018. 69p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

O problema do subgrafo planar de peso máximo consiste em extrair um subgrafo planar maximal, a partir de um grafo completo com pesos atribuídos às arestas, cuja soma dos pesos das arestas seja máxima. Este trabalho propõe soluções heurísticas, construídas por meio de Unidades de Processamento Gráfico (GPUs), baseadas em transformações locais na topologia do grafo através da inserção/realocação de vértices e arestas. Implementações sequencias e paralelas foram propostas, apresentando resultados satisfatórios. Em uma das propostas, a versão paralela requer cerca de 25 segundos de execução em uma instância de 100 vértices, sendo cerca de 200 vezes mais rápida que a versão sequencial correspondente. Em termos de qualidade da solução, as propostas superaram os resultados obtidos por algoritmos no estado da arte.

Palavras–chave

Planaridade, MWPSP, Paralelismo, CUDA.

Abstract

de Sousa Coelho, Vinícius. **An applied study using parallel programming for the weight maximal planar subgraph**. Goiânia, 2018. 69p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

The Maximum Weight Planar Subgraph Problem (MWPSP) consists of identifying a planar subgraph of maximum weight of a given edge-weighted graph. This work proposes new heuristic solutions, mainly using Graphic Processing Units, based on local transformations on the graph topology, consisting of vertex and edge insertion/relocation moves. Sequential and parallel implementations were built and applied to various numerical instances with promising results. One of the approaches requires only 25 seconds of execution, being more than 200 times faster than its corresponding sequential version, for a 100-vertex instance. In terms of quality, the proposed solutions obtained better results than cutting edge proposals.

Keywords

Planarity, MWPSP, Parallelism, CUDA.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

A análise contextual de objetos é comumente facilitada quando os elementos em questão são modelados para um grafo, onde vértices representam os objetos em questão (como ações, instalações industriais, circuitos elétricos, proteínas ou redes sociais) e arestas representam os relacionamentos entre estes elementos [33]. Para mais detalhes conceituais relacionados a grafos, veja [4]. A visualização destes grafos fica mais clara quando é possível apresentar um desenho planar, de forma que nenhuma de suas arestas se cruzem. Em algumas aplicações é comum atribuir pesos não-negativos a cada aresta do grafo, estipulando assim um grau de importância para cada relacionamento (quanto maior o valor, mais importante a relação é).

Contudo, estas aplicações envolvem grafos completos e à medida que o número de vértices cresce, dificulta a identificação das relações presentes, sendo assim necessário estabelecer algum critério de filtragem do grafo, eliminando relacionamentos de pouca importância. Uma das formas de filtragem pode ser alcançada gerando um subgrafo planar maximal ou resolvendo o *Maximum Weight Planar Subgraph Problem* (MWPSP). O MWPSP consiste em identificar um subgrafo planar de peso máximo a partir de um grafo simples completo com pesos não-negativos atribuídos às arestas. A planaridade garante uma visualização clara da solução por não apresentar arestas que se cruzem.

Uma descrição do problema é dada a seguir. Considere um grafo simples completo, não direcionado, de forma que $G = (V, E)$ seja um conjunto finito de vértices V e de arestas E, com $n = |V|$ representando a quantidade de vértices e $m = |E|$ a quantidade de arestas, tal que $w : E \to \mathbb{R}^+$. Dado um grafo G, o problema consiste em encontrar o maior subgrafo planar cuja soma do peso de suas arestas seja o máximo possível. Uma vez que as arestas possuem pesos não-negativos, sempre existirá uma solução que seja um subgrafo planar maximal de G, reduzindo o conjunto solução.

A análise de dados financeiros através de grafos planares é recente, mas bastante conhecida [3, 15, 33, 46]. Aplicações com grafos planares maximais são capazes de identificar atividades industriais e estruturas do mercado finaneiro [36, 37]. Assim, é possível construir um portifólio de risco financeiro, possibilitando a redução de riscos ao investir em um grupo de ações que ocupam regiões periféricas do grafo [41].

Além da análise de dados financeiros, o MWPSP tem outras aplicações importantes: (*i*) como um subproblema do *layout* de instalações, também conhecido como *facility layout problem*, onde os vértices representam atividades na instalação e as arestas sendo as adjacências entre as atividades [1, 32, 40, 42, 43] (*ii*) *design* de circuitos integrados – onde vértices são os componentes eletrônicos e arestas as ligações físicas entre eles [30], (*iii*) sistemas biológicos, onde vértices representam proteínas e arestas as interações de uma rede metabólica [45], e (*iv*) sistemas sociais – onde vértices representam agentes sociais (e.g. indivíduos ou grupos) e arestas representam interações sociais [13]. A natureza dos problemas (*i*) e (*ii*) implica que qualquer solução para o problem deve ser um subgrafo planar. Já as soluções para os problemas (*iii*) e (*iv*) não precisam necessariamente serem planares; a planaridade foi adicionada como uma forma de restrição que facilita a visualização da solução final.

Algoritmos exatos, heurísticos e de melhoria são bastante conhecidos para o MWPSP [1, 19, 33, 46]. Os algoritmos exatos são geralmente baseados em modelos de Programação Linear Inteira (PLI), aplicando técnicas de *branch and bound*, *cutting planes* e relaxação Lagrangeana. Os algoritmos heurísticos baseiam-se em construções vértice a vértice, busca local ou meta-heurísticas. Algoritmos de melhoria utilizam substituição de arestas e realocação de vértices de uma face planar para outra.

Arquiteturas paralelas compostas de diversos núcleos de processamento começaram a ser consideradas para o MWPSP, uma vez que surgiu a possibilidade de aumentar o tamanho das instâncias e de melhorar a qualidade das soluções. De forma similar, arquiteturas *many-core*, tais como GPUs (*Graphical Processing Units* ou Unidades de Processamento Gráfico), aumentaram a quantidade de núcleos de forma assustadora, com milhares de *cores* em uma única placa. Processadores *many-core*, também conhecidos como aceleradores, se tornaram mais acessíveis graças a crescente indústria de jogos. Apesar dos processadores serem mais simples e mais lentos, estes dispositivos foram desenvolvidos para lidar com um paralelismo massivo, tendo milhões de *threads* a seu dispor.

Este trabalho tem como objetivo propor algoritmos heurísticos, sequencial e paralelo para o problema do MWPSP, apresentando os resultados da implementação proposta com base em instâncias numéricas, sendo algumas de larga escala, tanto construídas sinteticamente como extraídas na literatura. Os experimentos realizados foram muito satisfatórios e, definitivamente promissores. É importante também relatar que os resultados obtidos possuem acurácia igual ou superior a qualquer outro algoritmo já apresentado na literatura.

O trabalho está organizado da seguinte maneira: no Capítulo 2 são apresentados conceitos relacionados à planaridade em grafos e às principais heurísticas e soluções exatas conhecidas, além de apresentar um estudo referente aos trabalhos relacionados. No Capítulo 3, são apresentados conceitos de paralelismo em arquiteturas *multicore* e *many-*

core. Os algoritmos propostos e suas descrições são apresentadas no Capítulo 4. Detalhes sobre as implementações paralelas são abordados no Capítulo 5, e um comparativo entre a solução sequencial proposta *versus* a paralela é apresentado no Capítulo 6. Por último, no Capítulo 7, são apresentadas as considerações finais e projetos futuros referentes a este trabalho.

CAPÍTULO 2

Revisão Bibliográfica

Neste capítulo, serão apresentados fundamentos, conceitos e trabalhos relacionados ao problema que foi estudado. Os fundamentos são baseados em um estudo bibliográfico, de onde extraiu-se as informações necessárias para a elaboração deste trabalho. Em especial, serão destacadas as propostas de Massara et al. [33], Tumminello et al. [46] e Ahmadi et al. [1], conduzindo uma breve explicação sobre suas abordagens. Para maiores detalhes conceituais sobre grafos, indica-se ao leitor Foulds [20].

2.1 Propostas da literatura

O problema do subgrafo planar maximal de peso máximo (ou MWPSP) consiste em, dado um grafo simples completo com pesos não-negativos atribuídos a suas arestas, selecionar um subgrafo planar maximal cuja soma do pesos das arestas seja máxima. De acordo com [46], subgrafos planares são relevantes em diversos domínios, como:

1. análise de dados financeiros, onde vértices representam ativos financeiros (tais como preços de ações na bolsa de valores, *spreads*, indicadores de risco e liquidez, etc.) e arestas representam correlações ou outras medidas de dependência entre eles;
2. *layouts* de instalações, onde vértices representam as instalações e arestas as afinidades entres elas;
3. *design* de circuitos integrados, onde vértices são componentes elétricos e arestas são as conexões físicas entre eles;
4. sistemas biológicos, onde vértices são proteínas e arestas são interações entre proteínas em uma rede metabólica; e
5. redes sociais, onde vértices são agentes sociais (indivíduos, empresas, grupos, etc.) e arestas são interações sociais.

Os primeiros modelos matemáticos foram introduzidos por [28], formulado como um problema de atribuição quadrática (*Quadratic Assignment Problem*, QAP). Elshafei [14] apresentou uma aplicação do QAP com o objetivo de minimizar os gastos

operacionais e o deslocamento de pacientes entre uma clínica e outra, de acordo com a localização de clínicas em um hospital.

A aplicação deste problema no cenário industrial foi apresentada por [35]: dado um conjunto de restrições que devem ser atendidas, organizar instalações individuais da melhor forma possível. Seppänen e Moore [42] apresentaram uma das primeiras aplicações para o problema envolvendo Teoria dos Grafos. A importância da aplicação está diretamente relacionada ao projeto de *layouts*, instalações e manuseamento de materiais. De acordo com Meller e Gau [34], cerca de 20 a 50% dos gastos operacionais de indústrias norte-americanas provém do manuseio de materiais. Com um planejamento adequado, é possível reduzir estes gastos em 10 a 30%.

Apesar do simples objetivo das aplicações mencionadas, o problema é bem complicado na prática. Para ser mais claro, [21] demonstrou que o MWPSP é um problema *NP*-Completo, ou seja, não se conhece uma solução em tempo polinomial capaz de resolver o problema. Para instâncias pequenas, digamos apenas 10 vértices, a quantidade de combinações possíveis já é enorme. Seja n a quantidade de vértices e $e = \frac{n \cdot (n-1)}{2}$ a quantidade de arestas em um grafo simples completo:

$$\binom{e}{3 \cdot n - 6} = \frac{45!}{24! \cdot 21!} = 3.773.655.750.150 \tag{2-1}$$

Mesmo com o uso de algum algoritmo de teste de planaridade eficiente, de complexidade assintótica linear [10, 24, 38], no pior caso seria necessário testar todas as 3.773.655.750.150 possibilidades, inviabilizando a solução. Foulds e Robinson [17] descreveram um algoritmo baseado em *branch-and-bound* para obter resultados ótimos. Apesar das restrições aplicadas, o tempo computacional necessário para apenas 12 vértices era inviável.

Assim, surgiu a necessidade de criar soluções que não fossem ótimas, mas que apresentassem uma qualidade satisfatória e próxima do ideial, conhecidas por heurísticas. Uma das soluções pioneiras foi proposta por [18, 19], descartando a necessidade de um teste de planaridade. A ideia consiste em calcular o grau de importância de cada vértice do grafo, somando os pesos das arestas incidentes em cada vértice. Assim, os 4 vértices mais "pesados" representam uma solução inicial, consistindo em uma estrutura chamada de tetraedro [19], representado pelo grafo simples completo K_4.

Uma característica particular de subgrafos planares maximais é a presença de regiões triangulares, chamadas de faces [20]. Isto é, toda face é composta por exatamente 3 vértices. Tendo isso em mente, a solução é construída a partir das faces do grafo. Para cada vértice que não esteja presente na solução atual, será calculada a importância da inserção deste vértice em uma das faces. A importância é calculada pela soma das 3 arestas que serão adicionadas caso o vértice seja escolhido.

Este método é também conhecido como *dimpling*, criando espécies de "covi-

nhas" no grafo. A vantagem deste método é a ausência de um teste de planaridade, pois a planaridade do grafo é mantida a cada etapa de inserção de vértices. O método é repetido até que não existam vértices a serem escolhidos. Por ser uma estratégia gulosa, o algoritmo procura a combinação de maior peso a cada etapa de escolha. No entanto, é possível que a escolha de maior peso não produza a melhor solução. O algoritmo é também conhecido por *heurística do deltaedro*, pois a solução obtida consiste em um poliedro com $2 \cdot n - 4$ faces.

Eades et al. [12] propôs algo semelhante a [18, 19], tendo como solução inicial um tetraedro. A principal diferença consiste na evolução da solução, consistindo em *expansões em roda*. Ela é assim chamada pois a cada passo de adição de um vértice, uma divisão (*split*) é aplicada, criando duas rodas a partir de uma. O conceito de roda provém do grafo roda. Uma roda formada por n vértices contém um ciclo com $n-1$ vértices e um vértice ao centro, chamado de cubo.

Outros autores basearam-se na estratégia de Foulds e Robinson [18, 19], com o objetivo de obter uma solução de melhor qualidade. Uma delas, proposta por Green e Al-Hakin [22], consiste em uma abordagem de duas etapas: uma de construção e outra de melhoria. O processo de construção desenvolve uma solução inicial, semelhante ao tetraedro, até uma possível solução representada por um subgrafo planar maximal (*Maximal Planar Subgraph*, MPG). Em seguida, é aplicada uma etapa de melhoria, composta de uma sequência de operações locais, com o objetivo de encontrar um MPG mais "pesado" que o anterior.

Dyer et al. [11] promoveram um comparativo entre alguns algoritmos existentes, a fim de avaliar a qualidade das suas soluções. Foulds et al. [16] realizaram um comparativo entre as propostas de Foulds e Robinson [18] e Eades et al. [12]. Kusiak e Heragu [29] apresentaram uma análise de modelos propostos por outros autores e, mais tarde, Heragu e Kusiak [23] propuseram uma modelagem baseada em programação linear.

Leung [31] propôs uma generalização do método de *dimpling* onde, ao invés da inserção de vértices, triangulações são inseridas, mantendo a propriedade de planaridade. Na prática, um outro problema surge: é necessário selecionar uma permutação de três vértices para a inserção, ao contrário de escolher somente um vértice. Apesar da difícil implementação, Leung [31] garante que matematicamente o método produz resultados superiores ao proposto por Foulds e Robinson [18, 19].

Boswell [5] apresentou um novo algoritmo chamado de TESSA, argumentando que a natureza da solução não é restrita ao tipo do *layout* produzido, ao contrário das propostas de [18, 22]. Os resultados obtidos tiveram acurácia de 90% próximos aos limites superiores conhecidos para cada instância.

Osman et al. [39] propuseram uma estratégia de busca aleatória de forma gulosa (*Greedy Random Adaptive Search Procedure*, GRASP) para solucionar o MWPSP.

Além disso, foi proposta uma estratégia dinâmica para atualizar uma lista restrita de candidatos, na qual utilizou-se uma estrutura de dados eficiente desenvolvida para a construção da heurística de [22]. A estrutura de dados reduz a complexidade da solução heurística de complexidade de tempo $O(n^3)$ para $O(n^2)$. A heurística de Green e Al-Hakin [22] e a estrutura de dados proposta por eles integram com uma busca local, realizando movimentos entre os vizinhos dos candidatos, descrito em [40], para obter uma implementação eficiente do GRASP.

Uma outra aplicação do MWPSP foi proposta por [46], utilizando dados da bolsa de valores. Os autores propuseram uma técnica de complexidade $O(n^3)$, chamada de *Planar Maximally Filtered Graph* (PMFG), que extrai um subgrafo contendo as relações de maior importância. A técnica mostra-se adequada para grafos baseados em correlacionamentos, fornecendo assim um grau de similaridade entre os objetos em questão. O uso deste procedimento em um conjunto de 100 ações do mercado de capitais norte-americano demonstrou que o subgrafo obtido continha relações significantes referentes a estrutura e propriedades do mercado.

Massara et al. [33] propuseram uma solução para a mesma aplicação, chamanda de *Triangular Maximally Filtered Graph* (TMFG), com complexidade de tempo $O(n^2)$. Os autores realizaram um comparativo do TMFG frente ao PMFG, sendo que o primeiro obteve um melhor desempenho computacional. Além disso, apresentaram movimentos locais para construção da solução e técnicas de melhoria subsequentes, e mencionaram o uso de paralelismo na solução deste problema.

Mais recentemente, Ahmadi et al. [1] investigou o MWPSP, apresentando uma nova modelagem em Programação Linear Inteira (PLI), com uso do CPLEX [25]. O algoritmo permite solucionar o problema de forma ótima, sendo mais eficiente que outros algoritmos conhecidos na literatura, além de fornecer um novo limite superior para avaliar a qualidade de soluções heurísticas. Resultados computacionais indicaram que instâncias de até 24 vértices puderam ser resolvidas de maneira ótima em um tempo computacional aceitável, convergindo ao resultado, na maioria dos casos, em até 6 iterações do algoritmo.

Por último, desenvolvemos uma nova heurística [7] baseada em [19, 33]. A ideia consiste em, ao invés de explorar somente um tetraedro como solução inicial, explorar todos os possíveis tetraedros de forma paralela. Dessa forma, cada solução é construída independentemente das outras, ampliando o espaço de busca e garantindo uma solução de qualidade igual ou superior às soluções anteriormente propostas. Nesta dissertação, obtivemos um novo limite superior para instâncias de até 40 vértices (resolvidos de forma ótima), tornando possível uma melhor análise qualitativa da solução proposta, além da introdução de novas heurísticas capazes de melhorar a qualidade das soluções.

Nas próximas seções, serão apresentados breves detalhes sobre as abordagens propostas por Tumminello et al. [46] e Massara et al. [33]. Além disso, serão apresentados

procedimentos de transformação que possibilitam melhorias em soluções heurísticas obtidas para o MWPSP.

2.2 PMFG

O PMFG (*Planar Maximally Filtered Graph*), proposto por Tumminello et al. [46], é um algoritmo do tipo *bottom-up* (de baixo para cima) para o problema do MWPSP, onde cada iteração é feita uma escolha gulosa (escolha ótima local com a esperança de encontrar uma solução ótima global), adicionando aresta por aresta à solução.

Em alguns domínios, a restrição de planaridade é uma consequência direta de uma geometria planar bidimensional de um problema, enquanto em outros ela surge como uma alternativa para restringir a complexidade de um grafo, reduzindo o grau de entrelaçamento.

Dado um grafo completo $G = (V, E)$, a primeira etapa do PMFG consiste em encontrar a árvore geradora máxima de G. A árvore geradora máxima é uma versão invertida da árvore geradora mínima onde, ao invés de gerar a árvore de menor peso, obtém-se a árvore de maior peso. Em seguida, ordena-se as arestas restantes de G de forma decrescente, de acordo com os pesos. Para cada aresta, é feita uma tentativa de inserção no subgrafo, descartando as arestas que violarem a restrição de planaridade. Para isso, um teste de planaridade é realizado para cada aresta inserida (muito caro computacionalmente), sendo necessário n^2 testes de planaridade para G. Assumindo que o teste de planaridade tenha complexidade de tempo $O(n)$, o algoritmo como um todo possui complexidade $O(n^3)$.

2.3 TMFG

O TMFG (*Triangulated Maximally Filtered Graph*) é um algoritmo proposto por Massara et al. [33] baseado em um método de filtragem de rede. A ideia consiste em construir uma triangularização que maximiza uma função objetiva, chamada de *função de ranqueamento*, associada a uma quantidade de informação retida na rede, utilizando movimentos de transformação em sua topologia local. Os movimentos serão discutidos em seções adiante deste trabalho.

Dado um grafo completo $G = (V, E)$, o TMFG consiste em encontrar uma clique de ordem 4 (K_4) cuja soma das arestas seja a maior possível e adicionar os vértices restantes nas faces da solução, chamado de movimento T_2. A cada etapa, o algoritmo otimiza uma *função de pontuação* (e.g. a soma dos pesos das arestas). Este método não depende de uma ordem particular dos vértices e, por isso, a cada passo é calculada a pontuação que seria obtida ao adicionar qualquer um dos vértices restantes dentro de

uma possível face. O movimento T_2 é então aplicado ao par de vértice e face de maior pontuação, enquanto existirem vértices a serem inseridos.

De acordo com Massara et al. [33], uma implementação simples consistiria em avaliar a função de ganho para todo par composto por um vértice e face, resultando em $O(n^2)$ cálculos a cada etapa, resultando assim em uma complexidade de tempo final $O(n^3)$. Contudo, é possível manter e atualizar incrementalmente uma *cache* contendo a informação sobre o melhor par, atualizando somente os registros afetados pelo movimento. Assumindo que o cálculo do máximo requer $O(\backslash)$ operações para n vértices, o número total de operações é $O(n^2)$.

2.3.1 Paralelismo proposto por Massara et al.

Massara et al. [33] sugeriram em seu artigo que um possível uso de paralelismo no TMFG consistiria na construção de cada face triangular de um tetraedro inicial simultaneamente. Dessa forma, cada face evoluiria seus subgrafos de forma independente das demais faces. Embora essa abordagem possa diminuir o tempo de execução, ela não oferece qualquer garantia de melhoria na qualidade do resultado. Em um primeiro ponto, dado que o grafo precisa "crescer" utilizando os vértices que não foram inseridos na solução, seria necessário que estes fossem armazenados em uma lista compartilhada; caso contrário, o grafo gerado seria inconsistente. Para acesso à lista, seria preciso o uso de algum mecanismo de controle de concorrência, e.g. um *lock*, o que diminuiria o potencial de paralelismo neste contexto.

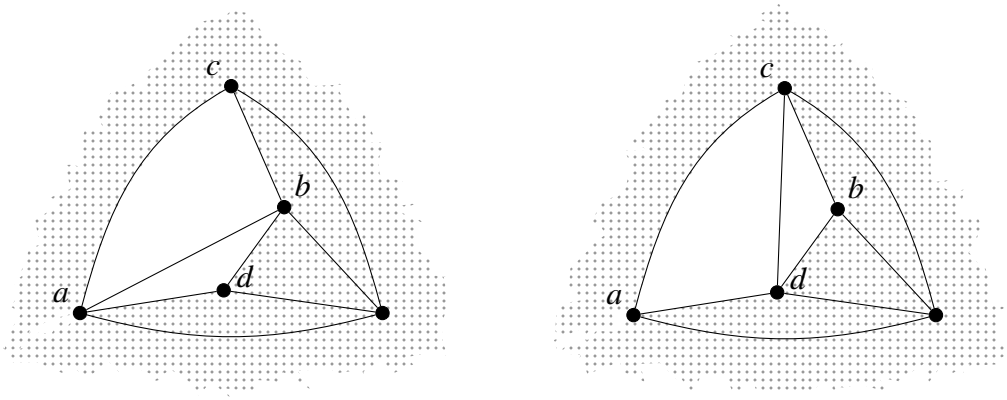
Em um segundo ponto, o algoritmo de [33] já é rápido mesmo em instâncias grandes, executando em cerca de 90 segundos uma instância de 10.000 vértices. Portanto, o uso de paralelismo não seria muito vantajoso. Por último, o objetivo do problema consiste em obter uma solução planar para um grafo, ou seja, que seja de fácil compreensão quando desenhada no plano. Para grafos grandes (e.g. 10.000 vértices), este desenho pouco ajudaria no entendimento da rede subjacente.

Na próxima seção, é feita uma descrição dos procedimentos de transformação, apresentando possíveis casos de uso. Os procedimentos listados visam melhorar soluções existentes através da substituição de arestas e realocação de vértices.

2.4 Procedimentos de melhoria

Um subgrafo planar maximal pode ser transformado em outro através de diversos movimentos (ou transformações) topológicos locais. Entre estes procedimentos, temos a substituição de aresta, onde uma aresta é removida e substituída por outra na diagonal do C_4 (ciclo com 4 vértices) induzido, realizando um procedimento de altera-

ção de corda. (Este procedimento é chamado de movimento T_1 por [33]). É importante ressaltar que a aresta utilizada na substituição esteja presente no grafo. O movimento de substituição de aresta é ilustrado pela Figura 2.1. Seguindo a figura, seja $G = (V,E,F)$ um subgrafo planar maximal com $n \geqslant 5$ e $\{a,b,c\},\{a,b,d\} \in F$. No primeiro caso, se $\{c,d\} \notin E$, o movimento substitui a aresta $\{a,b\}$ por $\{c,d\}$ transformando G no subgrafo planar maximal $G' = (V',E',F')$ onde $V' = V$, $E' = (E \setminus \{\{a,b\}\}) \cup \{\{c,d\}\}$, $F' = (F \setminus \{\{a,b,c\},\{a,b,d\}\}) \cup \{\{a,c,d\},\{b,c,d\}\}$. Neste primeiro caso, o movimento envolve a substituição de $\{a,b\}$, aresta diagonal do $C_4(a,c,b,d)$, por outra aresta diagonal, $\{c,d\}$.



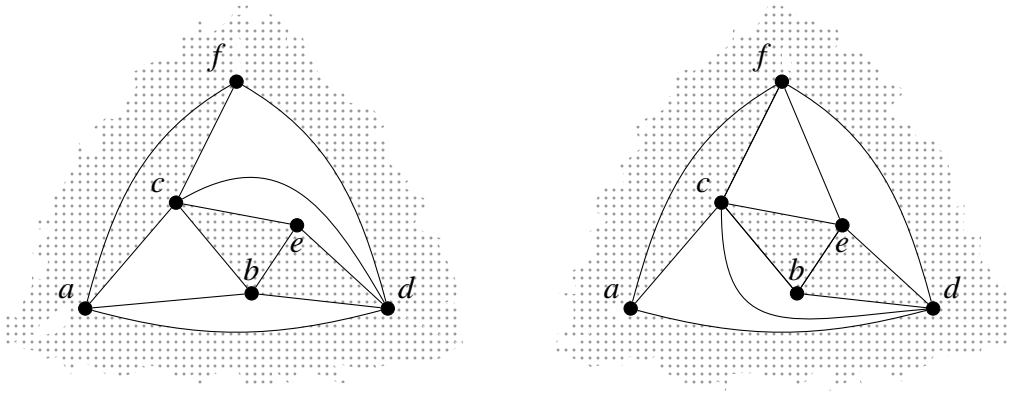
(a) *Grafo G antes do movimento* T_1.

(b) *Movimento* T_1 *onde a aresta* $\{a,b\}$ *em G é substituída por* $\{c,d\}$.

Figura 2.1: *Movimento de substituição de aresta (primeiro caso).*

Contudo, na tentativa de fazer o movimento de substituição de aresta é possível que $\{a,b\},\{c,d\} \in E$ com G planar (maximal) e $n \geqslant 5$. Este segundo caso é ilustrado pela Figura 2.2. Seguindo a ilustração, suponha que as faces $\{a,b,c\},\{a,b,d\},\{c,d,e\},\{c,d,f\} \in F$. Tanto a, b, e e f são distintos ou a ou b coincide com e ou f. Em ambos os casos, o movimento T_1 substitui a aresta $\{a,b\}$ pela aresta $\{e,f\}$. Note que $\{e,f\} \notin E$, caso contrário a, b, c, d, e e f são vértices do grafo bipartido completo (não-planar) a, d e e com b, c e f. Substituindo $\{a,b\}$ por $\{e,f\}$, transforma G no subgrafo planar maximal $G' = (V',E',F')$ onde $V' = V$, $E' = (E \setminus \{\{a,b\}\}) \cup \{\{e,f\}\}$, $F' = (F \setminus \{\{a,b,c\},\{a,b,d\},\{c,d,e\},\{c,d,f\}\}) \cup \{\{a,c,d\},\{b,c,d\},\{c,e,f\},\{d,e,f\}\}$. Note que se $\{a,b\} = \{e,f\}$ então $n = 4$, não exisitindo movimento T_1.

Outro movimento possível é chamado de *realocação de vértice* (ou T_3), onde um vértice de grau 3, se existir, é removido e realocado em outra face, como mostra a Figura 2.3. Por exemplo, seja $G = (V,E,F)$ um subgrafo planar maximal com $n \geqslant 5$, um vértice u com grau 3 e faces distintas $\{a,b,u\},\{b,c,u\},\{a,c,u\},\{p,q,r\} \in F$. O movimento T_3 remove u, criando a face $\{a,b,c\}$ e inserindo u na face $\{p,q,r\}$, transformando G no subgrafo planar maximal $G' = (V',E',F')$ onde $V' = V$, $E' = (E \setminus$

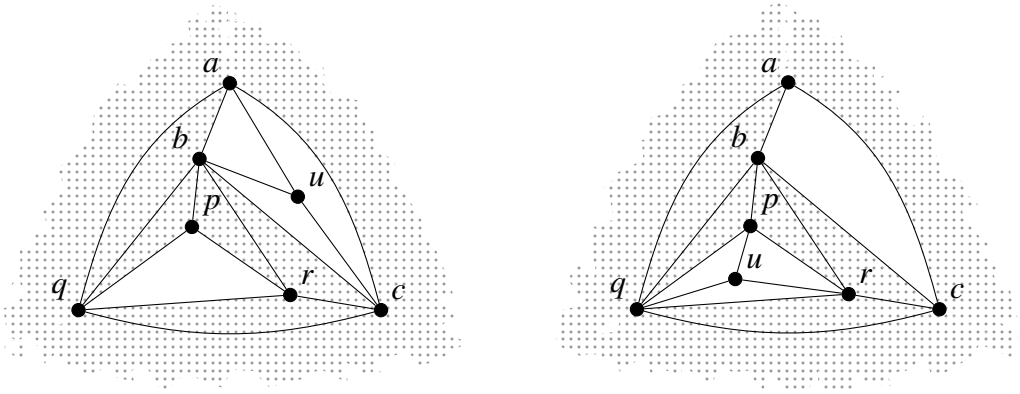


(a) *Grafo G antes do movimento* T_1.

(b) *O movimento* T_1 *onde cada aresta* $\{a,b\}$ *e G é substituída pela aresta* $\{e,f\}$.

Figura 2.2: *Movimento de substituição de aresta (segundo caso).*

$\{\{a,u\},\{b,u\},\{c,u\}\}) \cup \{\{p,u\},\{q,u\},\{r,u\}\},\ F' = (F \setminus \{\{a,b,u\},\{b,c,u\},\{a,c,u\},$ $\{p,q,r\}\}) \cup \{\{a,b,c\},\{p,q,u\},\{q,r,u\},\{p,r,u\}\}$.



(a) *Grafo G antes do movimento* T_3 *com o vértice u na face* $\{a,b,c\}$ *em G.*

(b) *O movimento* T_3 *com relocação do vértice u na face* $\{p,q,r\}$.

Figura 2.3: *O movimento T3.*

A seguir, um modelo de programação inteira para o problema, proposto por Ahmadi et al. [1], é apresentado. O modelo foi escolhido por ser uma proposta no estado da arte e de fácil compreensão, facilitando o desenvolvimento de um código testável.

2.5 Um algoritmo exato para o MWPSP

O algoritmo exato para o MWPSP proposto por Ahmadi et al. [1] consiste em uma abordagem iterativa de planos de corte (*cutting plane*) para um modelo de PLI. O *cutting plane* é um método de programação linear que busca iterativamente refinar um conjunto viável ou função objetiva por meio de inequações lineares, chamadas cortes. Com base em três propriedades de subgrafos planares maximais com n vértices [4]:

i) existem $3 \cdot (n-2)$ arestas ou, de forma equivalente, $2 \cdot (n-2)$ faces triangulares na representação planar do grafo;
ii) pelo menos três arestas são incidentes a cada vértice, ou seja, o grau de cada vértice é no miníno três; e
iii) exatamente duas faces triangulares estão associadas a cada aresta.

O modelo utiliza dois conjuntos de variáveis binárias para identificar subgrafos planares maximais com pesos nas arestas de um dado grafo $G = (V, E)$. Este conjuntos são definidos a seguir. Seja $f_{ijk} = 1$, se a tripla $\langle i, j, k \rangle$ de vértices mutualmente conectados por arestas forma uma face no subgrafo; $f_{ijk} = 0$, caso contrário, com $i, j, k \in V$ e $i < j < k$. Seja $x_{ij} = 1$, se os vértices i e j são conectados por uma aresta no subgrafo; $x_{ij} = 0$, caso contrário, com $i, j \in V$ e $i < j$. O parâmetro r_{ij} no modelo é um número não-negativo representando o grau de importância de que $i, j \in V$ sejam adjacentes. O modelo é apresentado a seguir.

Modelo de Ahmadi et al. [1]:

$$\max \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} r_{ij} x_{ij} \tag{2-2}$$

sujeito a:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} x_{ij} = 3 \cdot (n-2) \tag{2-3}$$

$$\sum_{i,j \in V:\, j<i} x_{ji} + \sum_{i,j \in V:\, j>i} x_{ij} \geqslant 3, \qquad i \in V \tag{2-4}$$

$$\sum_{k \in V:\, k<i} f_{kij} + \sum_{k \in V:\, i<k<j} f_{ikj} + \sum_{k \in V:\, k>j} f_{ijk} = 2 \cdot x_{ij}, \qquad i, j \in V,\ i < j \tag{2-5}$$

$$f_{ijk} < x_{ij}, \qquad i, j, k \in V,\ i < j < k \tag{2-6}$$

$$f_{ijk} < x_{jk}, \qquad i, j, k \in V,\ i < j < k \tag{2-7}$$

$$f_{ijk} < x_{ki}, \qquad i, j, k \in V,\ i < j < k \tag{2-8}$$

$$\sum_{i,j,k \in S:\, i<j<k} f_{ijk} \leqslant 2 \cdot (|S| - 2) - 1, \qquad S \subset V,\ |S| \geqslant 4 \tag{2-9}$$

$$x_{ij}, f_{ijk} \in \{0, 1\}, \qquad i, j, k \in V,\ i < j < k \tag{2-10}$$

A função objetiva (2-2) representa a soma total dos pesos das arestas selecionadas na solução para o MWPSP. As restrições (2-3)–(2-5) garantem, respectivamente, que a solução satisfaça as propriedades i)–iii). As restrições (2-6)–(2-8) afirmam que se uma face $\langle i, j, k \rangle$ é selecionada, as arestas (i, j), (j, k) e (i, k) devem ser também selecionadas. O conjunto de restrições (2-9) elimina a possibilidade de uma face interna ser considerada uma face externa. Por fim, a restrição (2-10) define o domínio das variáveis.

2.6 Resumo

Neste capítulo, foram apresentados trabalhos relacionados a esta dissertação, descrevendo possíveis aplicações para o problema. Dentre os trabalhos relacionados, foram destacadas as propostas heurísticas de Tumminello et al. [46] e Massara et al. [33] pela mais recente relevância na literatura. Ambos propuseram soluções capazes de encontrar boas soluções e de forma rápida, sem o uso de algortimos ou estruturas de dados muito complexas. Em seguida, procedimentos de transformação foram descritos, possibilitando a melhoria destas soluções. Por fim, um modelo de programação inteira, proposto por Ahmadi et al. [1], foi descrito, apresentando seu mapeamento para o MWPSP. No próximo capítulo, detalhes sobre o modelo arquitetural utilizado para o ambiente de testes será discutido em detalhes.

CAPÍTULO 3

Arquitetura Multicore e Manycore

Neste capítulo, serão abordados aspectos sobre as arquiteturas utilizadas para desenvolvimento das aplicações descritas nesta dissertação. Em uma primeira etapa, será apresentada uma visão geral sobre a arquitetura *multicore* e sobre o modelo de programação OpenMP. Em seguida, detalhes sobre a arquitetura *manycore* serão abordados, focando no funcionamento do *hardware* da GPU e no modelo de programação CUDA.

3.1 Arquitetura multicore e OpenMP

O termo *multicore* refere-se a uma arquitetura na qual um único processador físico incorpora a lógica de mais de um processador [44]. Um simples circuito integrado é usado para agrupar esses processadores. Em arquiteturas *multicore*, múltiplos núcleos de processamento são agrupados em um único processador físico com o objetivo de criar um sistema capaz de executar mais tarefas ao mesmo tempo, obtendo um melhor desempenho. O conceito de arquitetura *multicore* está centralizado na possibilidade de processamento paralelo, aumentando significamente a eficiência e desempenho através do uso de duas ou mais Unidades Centrais de Processamento (*Central Unit Processing*, CPU) em apenas um *chip*.

Um dos grandes motivos para o desenvolvimento da arquitetura *multicore* foi a dificuldade de aumentar o *clock* (frequência) dos processadores [44]. À medida que a frequência do *clock* aumenta, problemas de superaquecimento surgem, além do alto custo de energia para resfriamento das máquinas.

Um processador *multicore* requer menos espaço de circuito impresso (*Printed Circuit Board*, PCB) do que modelos do tipo SMP (*Symmetric Multiprocessor*). No SMP, cada processador dispõe da mesma fatia de tempo para acesso a um endereço de memória compartilhado, cada um sendo tratado da mesma forma pelo sistema operacional. Além disso, um processador *multicore* com dois núcleos utiliza menos energia do que dois processadores *SMP*, uma vez que a quantidade de energia necessária para comunicação com dispositivos externos ao *chip* diminui. O custo com resfriamento também diminui, dado que a frequência dos processadores diminui.

O modelo SMT (*Simultaneous Multi-Threading*) permite que múltiplas *threads* independentes possam executar simultaneamente no mesmo *core*. Cada *thread* consiste em um código de um programa que está sendo executado, com valores de variáveis e estruturas de dados. Se uma *thread* está bloqueada esperando uma operação de ponto flutuante concluir, outra *thread* pode executar operações com inteiros. Por outro lado, o modelo SMT não é verdadeiramente paralelo, pois o *chip* possui apenas uma cópia de cada recurso para todos os *cores*, ao contrário do modelo *multicore*, onde cada *core* tem sua própria cópia dos recursos.

3.1.1 Hierarquia de memória

Quanto à hierarquia de memória, todas as memórias *caches* são compartilhadas no modelo SMT. Já em *chips multicore*, a *cache* L1 é privada e a *cache* L2 pode ser privada ou compartilhada. Na arquitetura *multicore*, a memória do sistema é sempre compartilhada, como mostra a Figura 3.1. A memória privada tem a vantagem de ser mais próxima a CPU, portanto é muito rápida. Contudo, é uma memória muito pequena. Por outro lado, a memória compartilhada é maior, permitindo que *threads* em diferentes *cores* compartilhem o mesmo espaço de memória.

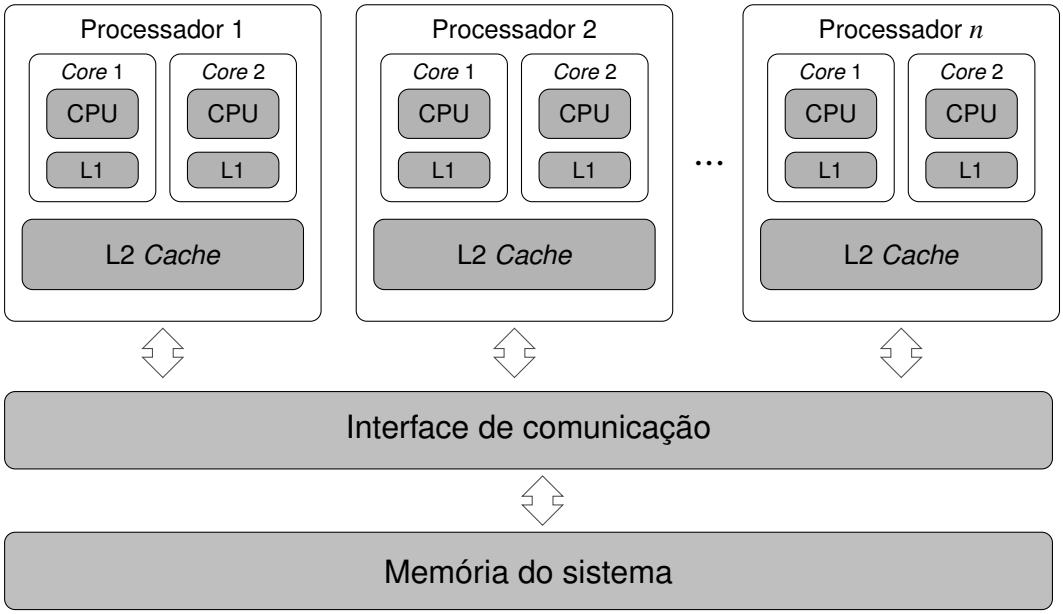


Figura 3.1: *Modelo de uma arquitetura multicore.*

Apesar da memória L1 ser muito rápida, existe um problema em manter a consistência dos dados entre as demais *caches* em cada núcleo, surgindo a necessidade de algoritmos eficientes e protocolos de consistência. Uma simples solução consiste em combinar o uso do *protocolo de invalidação* com o algoritmo de *snooping*. O protocolo de invalidação estabelece que, se um *core* realiza uma operação de escrita, todas as demais cópias nas outras *caches* são invalidadas. Já o algoritmo de *snooping* consiste em

"bisbilhotar" a interface de comunicação entre os *cores*, realizando um monitoramento na interface. Outros protocolos mais sofisticados como MSI (*Modified, Shared, Invalid*) e MESI (*Modified, Exclusive, Shared, Invalid*) representam os estados exclusivos que uma linha do *cache* se encontra, utilizando dois *bits* adicionais codificados na linha.

3.1.2 Modelo de programação OpenMP

OpenMP é uma API (*Application Programming Interface*) para desenvolvimento de aplicações *multi-thread*, consistindo em um conjunto de diretivas e rotinas passadas ao compilador pelo programador. O OpenMP oferece portabilidade e escalabilidade, fornecendo um modelo de programação paralela com memória compartilhada de forma simples e flexível [44]. O modelo falicitou o desenvolvimento de aplicações paralelas em Fortran, C e C++, padronizando práticas do modelo de programação SMP e *multicore*.

O modelo oferece também suporte para desenvolvimento de aplicações paralelas em sistemas embarcados e dispositivos aceleradores. Atualmente, o OpenMP ARB (*Architecture Review Board*) é a organização responsável por definir as especificações, produzindo e aprovando novas versões para a API. O ARB é um consórcio de organizações interessadas em padronizar o modelo, composto por gigantes como NVidia, Intel, AMD, IBM entre outras.

A maioria dos construtores aplicam um padrão de blocos estruturados (bloco com uma ou mais declarações com um ponto de entrada e outro de saída). A maioria das diretivas de OpenMP seguem o padrão `#pragma omp construct [clause [clause]...]`.

O OpenMP contém uma série de rotinas disponíveis para chamadas em tempo de execução. É possível modificar o número de *threads* através do comando `omp_set_num_threads()` e ao mesmo tempo perguntar quantas *threads* estão sendo utilizadas por meio do comando `omp_get_num_threads()`. Operações como `omp_get_thread_num()` (identifica o índice de uma *thread* em um laço paralelo), `omp_get_max_threads()` (devolve a quantidade máxima de *threads* disponíveis), `omp_in_parallel()` (verificar se o bloco em execução está em uma região paralela) e `omp_num_procs` (indica a quantidade de processadores no sistema) também estão disponíveis para uso.

No OpenMP, as *threads* se comunicam através de variáveis compartilhadas, sendo passível de alterações nos resultados quando as *threads* são escalonadas em diferente ordem. Por isso, é necessário que exista algum tipo de sincronização para evitar conflitos no acesso a um conteúdo compartilhado. No entanto, a sincronização é um procedimento caro, pois força que algumas *threads* fiquem ociosas enquanto esperam

as demais concluirem suas operações. As operações de sincronização disponíveis no OpenMP são: `critical`, `atomic` e `barrier`.

Na exclusão mútua (`critical`), apenas uma *thread* por vez pode entrar na região crítica de um código. A operação atômica (`atomic`) fornece uma exclusão mútua para operações de atualização em uma região de memória compartilhada. Na barreira (`barrier`), cada *thread* na região paralela deve esperar as demais, até que todas concluam seu processamento. É possível adicionar uma diretiva chamada `nowait` para burlar esse comportamento. O Código 3.1 ilustra um exemplo utilizando a operação `critical`.

No exemplo, a variável `res` é compartilhada por todo bloco parelelo. Dentro do bloco, a variável `idx` contém o índice da *thread* executando em um determinado instante. Cada *thread* executa `niters` operações dentro do laço e, uma por vez, armazena o resultado de `consume` levando em conta o valor de `B` obtido em `big_job`. Sem a diretiva `critical` na Linha 7, todas as *threads* efetuariam uma tentativa de escrita em `res` ao mesmo tempo, causando uma inconsitência no resultado. Neste exemplo, a operação `critical` poderia ser substituída por `atomic`, surtindo o mesmo efeito.

Código 3.1 Exemplo de uso de `critical`

```
1      float res;
2      #pragma omp parallel
3      {
4          int idx = omp_get_thread_num();
5          for (int i = 0; i < niters; i++){
6              float B = big_job(idx);
7              #pragma omp critical
8                  res += consume(B);
9          }
10     }
```

Para construir loops paralelos, basta adicionar a diretiva `#pragma omp parallel` no bloco desejado. Com isso, todas as variáveis declaradas dentro do bloco serão compartilhadas por todas as *threads*. Um exemplo contendo um laço dentro de uma região paralela é mostrado no Código 3.2. Neste caso, o simples fato de adicionar o laço a um bloco não produzirá o resultado correto, pois a variável `i` é compartilhada por todas as *threads*.

Para que o código funcione corretamente, é necessário adicionar a diretiva `#pragma omp for` para isolar o contexto de execução de cada *thread*. As diretivas pode ser combinadas para simplesmente `#pragma omp parallel for`. Caso existam laços aninhados, a diretiva recebe um parâmetro extra, transformando em `#pragma omp parallel for collapse(n)`, com *n* sendo a quantidade de laços aninhados.

Em alguns casos, a dependência das iterações dos laços não são trivialmente removíveis, dificultando a paralelização direta. Por exemplo, paralelizar a soma dos valores contidos em um *array* e armazenar em uma variável acumulável, também conhecida como *operação de redução*. O modelo fornece facilmente uma solução para este problema através do uso da diretiva `#pragma omp parallel for reduction(op:var)`, onde `op` representa o tipo da operação desejada (`+`, `-`, `*`, `/`) e `var` uma variável compartilhada.

Código 3.2 Um paralelismo com resultado inesperado

```
1       #pragma omp parallel
2       {
3           int i;
4           for (int i = 0; i < n; i++)
5               a[i] = a[i] + b[i];
6       }
```

3.2 GPU e a plataforma CUDA

A Unidade de Processamento Gráfico (*Graphics Processing Unit*, GPU) é um microprocessador especializado em computação gráfica. Normalmente, localiza-se em placas gráficas ou, em formas mais simples, integradas nos *chipsets* da placa-mãe. Inicialmente, a GPU foi desenvolvida com propósitos de processamento gráfico. No entanto, houve uma tendência de evolução para uma computação aplicada a propósitos gerais.

3.2.1 Conceitos iniciais

A programação paralela consiste em um processamento simultâneo de várias tarefas e dados, sendo estas tarefas executadas por *threads*. Tanto na Unidade Central de Processamento (*Central Processing Unit*, CPU) quanto na GPU, o processamento de uma *thread* é sequencial. No entanto, a CPU é projetada para executar uma pequena quantidade de tarefas bastante complexas, enquanto a GPU é projetada para o processamento de uma grande quantidade de tarefas simples. Consequentemente, o suporte à *thread* é muito diferente na CPU e GPU. Por exemplo, o número de registradores por núcleo é diferente para o suporte de troca de contexto das *threads*, sendo um único banco de registradores na CPU e muitos na GPU [9, 47].

A GPU trabalha em conjunto com a CPU, uma vez que os dados inicialmente estão localizados na memória RAM do computador e o recebimento de dados que serão armazenados na memória da GPU é realizado pela CPU. A Figura 3.2 ilustra, de maneira

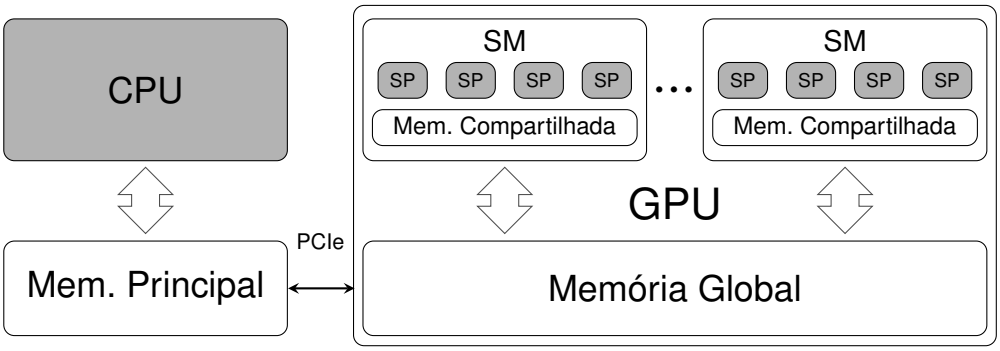


Figura 3.2: *Modelo arquitetural de uma GPU.*

geral, a conexão entre a GPU e a CPU. Observe que a conexão entre elas é realizada por meio do barramento PCI Express (PCIe). Esse barramento oferece 16 GB/s de taxa de transferência de dados, na versão 2.0. Já na versão 3.0, este valor chega a ser 32 GB/s [47]. Mais recentemente, a NVidia dispinibilizou no mercado uma tecnologia chamada de NVLink, capaz de diminuir o gargalo existente no barramento PCIe. NVLink é uma conexão com alta largura de banda entre a GPU e CPU, apresentando taxas de transferências de pelo menos 80 GB/s. A tecnologia foi disponibilizada no mercado juntamente o lançamento da arquitetura Pascal, em 2016.

A NVidia é uma das responsáveis por disponibilizar uma arquitetura baseada em GPUs, conhecida como CUDA (*Compute Unified Device Architecture*). A arquitetura CUDA pode ser dividida entre *hardware* e *software*. Com relação ao *hardware*, a arquitetura baseia-se em uma hierarquia tanto de processadores quanto de memória. Com relação à parte de *software*, é disponibilizado um modelo de programação com intuito de abstrair os detalhes do *hardware*.

3.2.2 Hardware CUDA

A arquitetura CUDA consite de três conceitos principais: processadores de fluxo (*Streaming Processors*, SP), multiprocessadores de fluxo (*Streaming Multiprocessors*, SM) e uma hierarquia de memória (registradores, compartilhada, global, constante e textura) [9, 47].

Os SPs, conhecidos também como CUDA *cores*, são definidos pela NVidia como núcleos de processamento e são constituídos de unidades lógicas e aritméticas (*Arithmetic Logic Units*, ALU) e unidades de ponto flutuante (*Floating Point Units*, FPU). As operações de ponto flutuante são executadas de acordo com o padrão IEEE 754-2008 [8], que fornece suporte à instrução *Fused Multiply-Add* (FMA) tanto para precisão simples (32 bits) quanto para precisão dupla (64 bits).

Ela é responsável por melhorar a instrução *Multiply-Add* (MAD), sendo mais precisa do que as operações de adição e multiplicação executadas separadamente. Tais

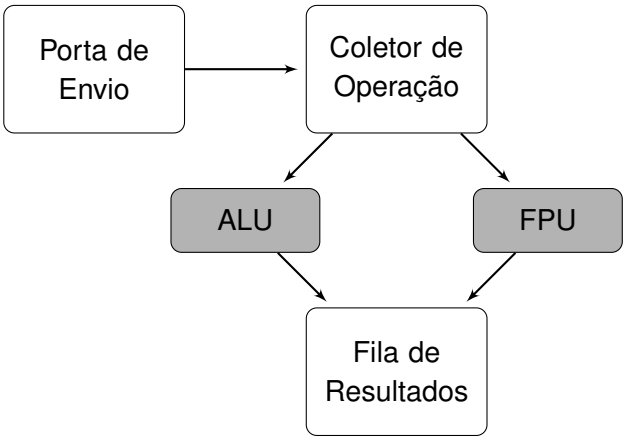


Figura 3.3: *Estrutura do processador de fluxo (SP).*

instruções, FMA e MAD, são frequentemente utilizadas em computação gráfica, álgebra linear e em aplicações científicas. As unidades de funções especiais (*Special Function Units*, SFU), localizadas nos SMs, são responsáveis por executar instruções especiais implementadas em *hardware*, como operações seno, cosseno, raiz quadrada e expoente. A Figura 3.3 ilustra, de forma simplificada, a arquitetura do SP contendo uma ALU, FPU e registradores.

Os SPs são agrupados em estruturas que compartilham vários recursos do *hardware*. Essas estrututas são chamadas de multiprocessadores de fluxo (SM), onde cada uma é responsável por uma grande quantidade de *threads*. Embora todos os SMs não trabalhem sincronizadamente, os SPs contidos em único SM processam os dados de maneira síncrona.

A quantidade de unidades de processamento em todos os SMs em uma GPU varia de geração para geração. De acordo com [9], essa quantidade é um aspecto chave da arquitetura, pois dentre outros fatores, permite escalabilidade da GPU. Com essa escalabilidade, o desempenho do *hardware* pode ser aumentado simplesmente combinando o número de SMs com o número de CUDA *cores* dentro de cada SM.

A presença de centenas de registradores dentro dos SMs permite que as trocas de contextos entre as *threads* sejam mais eficientes, maximizando o desempenho de processamento dos SMs. Isso possibilita que a GPU tenha uma solução eficaz em relação à latência contida no acesso à memória, no qual é gasto centenas de ciclos de *clock* para buscar e armazenar os dados [47].

A latência é o atraso ou espera que aumenta o tempo de resposta real ou percebido para além do tempo de resposta desejado. Quando um grupo de 32 *threads* executando em um SM fica ocioso por acessar a memória ou por alcançar uma barreira de sincronização, outro grupo de *threads* é escalonado instantaneamente para o SM. Isso é realizado pelos escalonadores de *warps* contidos nos SMs. Uma *warp* é um conjunto de *threads* que compartilham o mesmo código, executam a mesma instrução, tendo mínima

divergência, e que "param" no mesmo lugar.

A arquitetura CUDA tem alta vazão graças aos milhares de CUDA *cores* disponíveis em cada GPU, possibilitando um processamento massivamente paralelo, além de uma hierarquia de memória disponível para todos os *cores*. Essa hierarquia faz com que as unidades de processamento acessem com menor frequência as memórias com maior latência. Quanto mais alto é o nível da memória nessa hierarquia, maior é a velocidade de acesso.

No topo, encontra-se a memória privada dos SPs, que é composta por um conjunto de registradores de tamanho 32 bits, localizado dentro dos SMs. Esses registradores, semelhantes aos das CPUs, estão próximos aos SPs e possuem um tempo de acesso de mesma ordem de grandeza dos processadores de fluxo, sendo a mais rápida dentre as demais memórias.

Logo abaixo da memória privada, encontra-se a memória compartilhada, também localizada dentro dos SMs. Contudo, ela é acessível a todos os SPs contidos em um mesmo SM, o que não acontece na memória privada. Cada memória compartilhada é visível a um único SM. Normalmente utilizada para troca de dados entre os SPs do SMs, ela pode servir também como memória *cache* que, diferente da CPU, é inteiramente controlada pelo programador. Quanto a velocidade, a memória compartilhada é geralmente 10 vezes mais lenta do que a dos registradores [9]. Além disso, a quantidade de memória compartilhada disponível por SM é pouca, sendo de 64KB na arquitetura mais recente, Pascal.

Na base da hierarquia de memória, encontra-se as acessíveis por todos os SMs: global, constante e textura. A global é a principal memória da GPU e aceita operações de escrita e leitura. A memória constante somente aceita operações de leitura, sendo ela otimizada para *multibroadcast* envolvendo múltiplos SPs. A memória de textura é uma memória do tipo *read-only* que pode melhorar a performance e reduzir o tempo acesso aos dados.

Assim como a memória constante, a memória de textura é "cacheada" no chip, proporcionando em algumas situações uma maior largura de banda e, assim, reduzindo requisições de acesso à memória global. Embora tenha sido originalmente criada para aplicações gráficas, ela pode ser utilizada em outras aplicações em GPU.

3.2.3 Modelo de programação CUDA

O modelo de programação CUDA é uma extensão da linguagem C e C++, criada em 2006 pela NVidia Corporation. Esse projeto teve como principal objetivo permitir o desenvolvimento de aplicações massivamente paralelas e altamente escaláveis nas GPUs. A partir de sua criação, houve uma explosão no desenvolvimento de aplicações

sobre plataformas GPUs, principalmente de aplicações não-gráficas. Isso ocorreu devido à facilidade de desenvolver programas com pouco ou nenhum conhecimento em APIs gráficas. Assim, programas paralelos foram desenvolvidos para diversas áreas, como química, matemática, física e atividades gerais de busca e classificação [26].

A abstração dos detalhes da arquitetura é um ponto forte desse modelo, pois torna as aplicações sobre a plataforma da GPU altamente escaláveis e fáceis de desenvolver. Escalável porque o programa fica independente do modelo do *hardware* que está sendo compilado; e fácil de desenvolver por ter os detalhes do *hardware* CUDA encapsulados nas características do modelo programação. Uma aplicação desenvolvida no modelo de programação CUDA não é totalmente paralelizada pois requer duas fases: uma sequencial e uma paralela. Elas se alternam entre si uma ou mais vezes em uma única execução.

A CPU, também conhecida como *host*, inicia a execução, sendo ela sequencial. A GPU, chamada de *device*, é responsável pela fase paralela. Essa fase é realizada por funções, chamadas de *kernels*. Quando a execução de um *kernel* é concluída, o controle é redirecionado para a fase sequencial na CPU [26]. Na fase paralela, milhares de *threads* podem ser lançadas na GPU. A quantidade de *threads* é determinada no momento do lançamento do *kernel*.

As *threads* são organizadas em grades e blocos de *threads*. Cada bloco de *threads* da grade possui identificadores em relação à sua posição nos eixos x e y na grade e toda *thread* de um bloco também possui um identificador de sua posição nos eixos x, y e z em relação ao bloco.

Uma grade possui um único *kernel* e, consequentemente, os blocos de *threads* da grade contêm o mesmo código. Entretanto, não é possível determinar se todos os blocos processam a mesma instrução em um mesmo instante. Por esse motivo que, em nível de programação, o modelo de programação CUDA é classificado como *Single-Program Multiple-Data* (SPMD), pois cada bloco de *threads* é uma cópia das instruções do programa.

As *threads* que estão dentro de uma grade são organizadas em uma lista de blocos de *threads*. Essa lista pode ser de uma ou duas dimensões. Todos os blocos de uma mesma grade possuem o mesmo tamanho. Esse tamanho é definido pela quantidade de *threads*, sendo que a quantidade máxima pode variar de acordo com as gerações das arquiteturas.

Por exemplo, nas arquiteturas Fermi e Kepler a quantidade máxima é de 1024 *threads*. Todo o bloco de *threads* é processado por um único SM. Contudo, um SM pode processar concorrentemente vários blocos de *threads*. O limite de processamento simultâneo do SM também é definido de acordo com a arquitetura da GPU. Visto que o bloco é processado por um SM, um único fluxo de instruções é dado para todas as *threads* dentro dele.

Os blocos de *threads* são compostos por conjuntos de 32 *threads*, conhecidos por

warps. O SM particiona os blocos em *warps* de forma idêntica em todas as execuções. Ou seja, cada *warp* possui *threads* consecutivas, aumentando simplesmente os identificadores. Ele cria, gerencia e executa *threads* de uma *warp* em uma única vez. As *threads* da *warp* iniciam juntas o mesmo endereço do programa, embora cada uma possua seus próprios contadores de programa e registradores de estado [9]. Uma *warp* executa uma mesma instrução a cada momento. Isso implica que, no caso de divergência de algumas *threads* da *warp* em uma instrução condicional, a *warp* serializará os ramos de execução. Ou seja, partes das *threads* que não entraram no ramo em questão, ficarão ociosas. Vale ressaltar que o tratamento de divergências em instruções condicionais acontecem somente dentro de uma *warp*; em diferentes *warps*, a execução é independente.

Em relação à programação, os SMs são considerados arquiteturas *Single-Instruction Multiple-Threads* (SIMT), ou seja, único fluxo de instrução e múltiplas *threads*. Essa arquitetura, criada pela NVidia, especifica o comportamento de uma única *thread*, habilitando os programadores a desenvolverem códigos paralelos com *threads* independentes. Isso faz com que eles não sejam mais obrigados a escrever códigos para cada *thread* que segue o mesmo caminho de execução, similar ao modelo SPMD.

Threads podem divergir e, em seguida, convergir em algum ponto mais tarde. Porém, cada caminho deve ser executado sequencialmente até que o fluxo de controle se junte novamente. O modelo de programação não obriga os programadores a ter conhecimento dos detalhes desse fluxo, porém é importante que se conheça para que sejam obtidos bons desempenhos em aplicações paralelas [9, 26].

A linguagem CUDA é uma extensão da linguagem C e C++, possuindo algumas palavras reservadas próprias. Por exemplo, `__device__`, `__const__` e `__shared__` servem para identificar a localidade de variáveis declaradas na memória global, constante e compartilhada, respectivamente.

Código 3.3 Um exemplo de código em CUDA

```
1   __device__ void callMe(int matrix[], int size, int idx){
2       for (int i = 0; i < size; i++)
3           matrix[i] = idx + i;
4   }
5
6   __global__ void sample(){
7       int idx = blockDim.x * blockIdx.x + threadIdx.x;
8       __shared__ int matrix[10];
9       callMe(matrix, 10, idx);
10  }
```

Um exemplo em CUDA é mostrado no Código 3.3. A função `callMe` é invocada pelo *kernel* passando um *array* declarado na memória compartilhada da GPU, identificado

pela palavra reservada __shared__, o tamanho do *array* e o índice da *thread* que irá executar a função. A função *callMe* soma o valor do índice *i* do *array* com o índice *idx* da *thread* atual e armazena em $matrix[i]$.

A palavra reservada __global__ serve para indicar que a função é um *kernel*, sendo que toda função *kernel* tem o retorno do tipo `void`. A palavra reservada `threadIdx` é uma variável de três dimensões que identifica uma *thread* dentro do bloco de *threads*.

De forma semelhante, a variável `blockIdx` indica a identificação do bloco de *threads* e a variável `blockDim` contém a dimensão em números de *threads* do bloco. Assim, a variável `idx` recebe a identificação global da *thread*, ou seja, da *thread* na grade.

CAPÍTULO 4

Algoritmos

Dada a necessidade de melhorar as técnicas já existentes para o MWPSP, este trabalho segue com propostas heurísticas capazes de resolver instâncias maiores do problema de forma rápida e de qualidade no mínimo igual ou superior às propostas no estado da arte. A técnica consiste em paralelizar soluções iniciais distintas que, através de operações iterativas, constroem um subgrafo planar maximal.

A metodologia utilizada segue a linha de pensamento de [33]. Em relação ao algoritmo de [33], nossa proposta difere em três aspectos. Primeiramente, ao invés de escolher apenas uma semente, todas as possíveis combinações são utilizadas. Dessa forma, dado que diferentes sementes resultam em diferentes soluções, amplia-se o espaço de busca, possibilitando novos e melhores resultados. Segundo, evitamos uma busca linear para escolha dos pares utilizando uma fila de prioridades (*heap*). Por último, fazemos uso de uma estrutura de dados para armazenar o conjunto de faces, provendo rápido acesso as informações. Uma descrição mais detalhada das propostas é apresentada a seguir.

4.1 Face dimpling

A primeira proposta, chamada de *face dimpling* (movimento T2 por [33]), é descrita pelo Algoritmo 4.1. A solução tem inicialmente vários tetraedros gerados a partir de $\binom{n}{4}$ combinações de vértices. Para cada semente, cria-se uma lista $\mathcal{S}$ contendo todos os vértices que não estão na semente (Linha 4), uma matriz de adjacências $\mathcal{E}$ com a solução atual (Linha 5) e $\mathcal{M}'$ com a solução final, uma lista F contendo o conjunto de faces da semente (Linha 6) e uma estrutura de *heap* $\mathcal{H}$ contendo o par de vértice e face de maior ganho (Linha 7).

Em seguida, para cada semente é calculada a soma dos pesos de suas arestas e este valor é armazenado em $TmpMax$ (Linha 9). A cada passo é extraído de $\mathcal{H}$ o par de vértice e face de maior pontuação (Linha 10). O movimento de *face dimpling* é então aplicado ao par de vértice e face escolhido. O vértice escolhido é removido de $\mathcal{S}$ e a face é removida de F, dando lugar a 3 novas faces (Linha 14).

Algoritmo 4.1: Face dimpling

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacência $M_{n \times n}$, com arestas de pesos não-negativos.

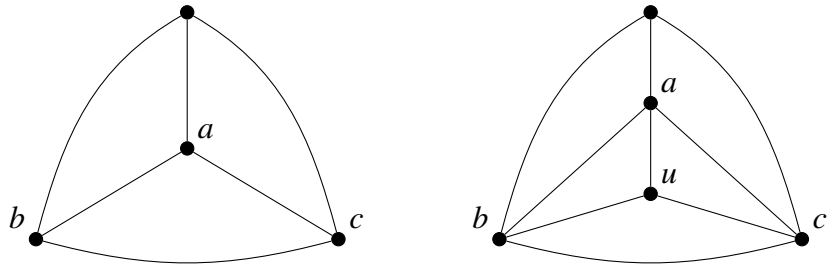
Saída: Grafo planar maximal representado por $\mathcal{M}'$, filtrado a partir de M.

1 $\mathcal{C} \leftarrow$ Conjunto de possíveis tetraedros $\{v_1, v_2, v_3, v_4\}$, com $v_i \in V$;
2 $MaxWeight \leftarrow 0$;
3 **para** $(k \leftarrow 1 \leq |\mathcal{C}|)$ **faça**
4 $\quad \mathcal{S} \leftarrow V \setminus \mathcal{C}_k$;
5 $\quad \mathcal{E} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
6 $\quad F \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
7 $\quad \mathcal{H} \leftarrow \forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s, f\}, \forall f \in F$;
8 $\quad TmpMax \leftarrow \sum w_e, \forall e \in \mathcal{E}$;
9 $\quad$ **enquanto** $(\mathcal{S} \neq \varnothing)$ **faça**
10 $\quad\quad \{v_i, f_i\} \leftarrow \max\{\mathcal{H}\}$, $v_i \in \mathcal{S}$ e $f_i \in F$;
11 $\quad\quad \mathcal{H} \leftarrow \mathcal{H} \setminus \max\{\mathcal{H}\}$;
12 $\quad\quad$ Execute o *dimpling*;
13 $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{v_i\}$;
14 $\quad\quad F \leftarrow (F \setminus \{f_i\}) \cup \{f_a, f_b, f_c\}$;
15 $\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{\{v_i, v_x\}, \{v_i, v_y\}, \{v_i, v_z\}\}$, com $v_x, v_y, v_z \in f_i$;
16 $\quad\quad TmpMax \leftarrow TmpMax + w_{\{v_i, v_x\}} + w_{\{v_i, v_y\}} + w_{\{v_i, v_z\}}$;
17 $\quad\quad \mathcal{H} \leftarrow \mathcal{H} \cup \{\forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s, f_a\}, \{s, f_b\}, \{s, f_c\}\}$;
18 $\quad$ **se** $(TmpMax \geq MaxWeight)$ **então**
19 $\quad\quad MaxWeight \leftarrow TmpMax$;
20 $\quad\quad \mathcal{M}' \leftarrow \mathcal{E}$;
21 **retorna** $\mathcal{M}'$.

O processo de *face dimpling* é ilustrado na Figura 4.1. Em seguida, é adicionado a $TmpMax$ o valor das 3 arestas inseridas na solução (Linha 16). É necessário que $\mathcal{H}$ seja atualizado para a próxima iteração, calculando-se o ganho de cada um dos vértices restantes quando inseridos nas 3 novas faces (Linha 17). Este processo é repetido enquanto $\mathcal{S}$ não estiver vazio. Concluída a etapa anterior, é então verificado se, para o grafo gerado, a soma do pesos de suas arestas é maior que o valor armazenado em $MaxWeight$. Em caso afirmativo, o valor é atualizado e a matriz de adjacências $\mathcal{E}$ que gerou a resposta é então armazenada em $\mathcal{M}'$ (Linhas 19 e 20).

4.2 Edge dimpling

A segunda proposta, chamada de *edge dimpling* (movimento A por [33]), é descrita pelo Algoritmo 4.2. De forma análoga ao Algoritmo 4.1, a solução tem inicialmente vários tetraedros gerados a partir de $\binom{n}{4}$ combinações de vértices. Para cada semente, cria-se uma lista $\mathcal{S}$ contendo todos os vértices que não estão na semente (Linha 4), uma matriz de adjacências $\mathcal{E}$ com a solução atual (Linha 5) e $\mathcal{M}'$ com a solução final, uma lista F



(a) *Grafo G antes do face dimpling.* (b) *Vértice u é inserido na face* $\{a,b,c\}$ *em G.*

Figura 4.1: *Inserção de um vértice em uma face, resulta em 3 novas faces.*

contendo o conjunto de faces da semente (Linha 6) e uma estrutura de *heap* $\mathcal{H}$ contendo o par de vértice e aresta de maior ganho (Linha 7). Em seguida, é calculada a soma dos pesos das arestas de cada semente e este valor é armazenado em $TmpMax$ (Linha 8).

Algoritmo 4.2: Edge dimpling

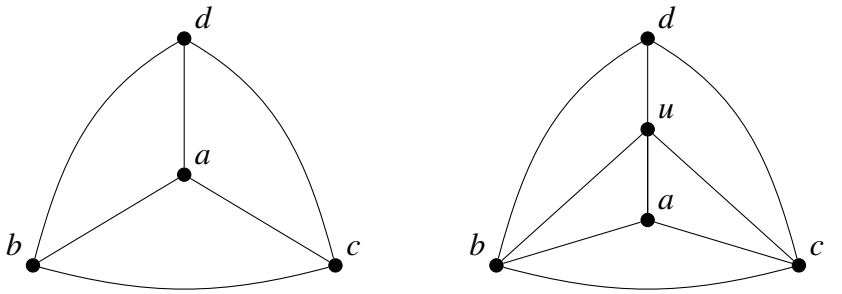
Entrada: Grafo $G = (V,E)$, representado pela matriz de adjacência $M_{n\times n}$, com arestas de pesos não-negativos.

Saída: Grafo planar maximal representado por $\mathcal{M}'$, filtrado a partir de M.

1 $\mathcal{C} \leftarrow$ Conjunto de possíveis tetraedros $\{v_1,v_2,v_3,v_4\}$, com $v_i \in V$;
2 $MaxWeight \leftarrow 0$;
3 **para** $(k \leftarrow 1 \leq |\mathcal{C}|)$ **faça**
4 $\mathcal{S} \leftarrow V \setminus \mathcal{C}_k$;
5 $\mathcal{E} \leftarrow \{\{v_1,v_2\},\{v_1,v_3\},\{v_1,v_4\},\{v_2,v_3\},\{v_2,v_4\},\{v_3,v_4\}\}$;
6 $F \leftarrow \{\{v_1,v_2,v_3\},\{v_1,v_2,v_4\},\{v_1,v_3,v_4\},\{v_2,v_3,v_4\}\}$;
7 $\mathcal{H} \leftarrow \forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s,e\}, \forall e \in \mathcal{E}$;
8 $TmpMax \leftarrow \sum w_e, \forall e \in \mathcal{E}$;
9 **enquanto** $(\mathcal{S} \neq \varnothing)$ **faça**
10 $\{v_i,e_i\} \leftarrow \max\{\mathcal{H}\}$, $v_i \in \mathcal{S}$ e $e_i \in E$;
11 $\mathcal{H} \leftarrow \mathcal{H} \setminus \max\{\mathcal{H}\}$;
12 Execute o *dimpling*;
13 $\mathcal{F} \leftarrow$ Faces que e_i pertence;
14 $\mathcal{R} \leftarrow$ Todos os vértices de $\mathcal{F}$;
15 $\mathcal{S} \leftarrow \mathcal{S} \setminus \{v_i\}$;
16 $F \leftarrow F \setminus \mathcal{F}$;
17 $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{e_i\}) \cup \{\{v_i,v_x\},\{v_i,v_y\},\{v_i,v_w\},\{v_i,v_z\}\}$, com $v_x,v_y,v_w,v_z \in \mathcal{R}$;
18 $F \leftarrow F \cup \{f_a,f_b,f_c,f_d\}$;
19 $TmpMax \leftarrow TmpMax - w_{\{e_i\}} + w_{\{v_i,v_x\}} + w_{\{v_i,v_y\}} + w_{\{v_i,v_w\}} + w_{\{v_i,v_z\}}$;
20 $\mathcal{H} \leftarrow \mathcal{H} \cup \{\forall s \in \mathcal{S}$, calcule o ganho máximo de s para cada aresta adicionada$\}$;
21 **se** $(TmpMax \geq MaxWeight)$ **então**
22 $MaxWeight \leftarrow TmpMax$;
23 $\mathcal{M}' \leftarrow \mathcal{E}$;
24 **retorna** $\mathcal{M}'$.

A cada passo é extraído de $\mathcal{H}$ o par de vértice e aresta de maior pontuação (Linha 10). Diferente do *face dimpling* que remove a face que resultou a maior pontuação, este remove a aresta. O movimento de *edge dimpling* é então aplicado ao par de vértice e aresta escolhido. O processo é um pouco mais trabalhoso neste algoritmo, pois além de remover o vértice escolhido de $\mathcal{S}$, é necessário desvincular a aresta escolhida e de todas as faces que ela pertence, pois ela será removida da solução. É preciso listar as faces que e pertence em $\mathcal{R}$ (Linha 13) e então obter todos os vértices que compõem estas faces (Linha 14). A aresta escolhida é removida de $\mathcal{E}$ e suas faces são removidas de F, dando lugar a 4 novas faces (Linha 17), conectando o vértice inserido aos vértices extraídos das faces removidas.

O processo de *edge dimpling* é ilustrado na Figura 4.2. Em seguida, é adicionado a $TmpMax$ o valor das 4 arestas inseridas na solução, subtraindo o valor da aresta removida (Linha 19). É necessário que $\mathcal{H}$ seja atualizado para a próxima iteração, calculando-se o ganho de cada um dos vértices restantes quando inseridos nas 4 novas arestas (Linha 20). Este processo é repetido enquanto $\mathcal{S}$ não estiver vazio. Concluída a etapa anterior, é então verificado se, para o grafo gerado, a soma do pesos de suas arestas é maior que o valor armazenado em $MaxWeight$. Em caso afirmativo, o valor é atualizado e o grafo $\mathcal{E}$ que gerou a resposta é então armazenado em $\mathcal{M}'$ (Linhas 22 e 23).



(a) *Grafo G antes do edge dimpling.*

(b) *Vértice u é inserido na aresta* $\{a,d\}$ *em G.*

Figura 4.2: *Inserção de um vértice em uma aresta, resultando em 4 novas faces.*

4.3 Face-edge dimpling

A terceira proposta, chamada de *face-edge dimpling*, nada mais é que a combinação dos Algoritmos 4.1 e 4.2, resultando no Algoritmo 4.3. A solução tem inicialmente vários tetraedros gerados a partir de $\binom{n}{4}$ combinações de vértices. Para cada semente, cria-se uma lista $\mathcal{S}$ contendo todos os vértices que não estão na semente (Linha 4), uma matriz de adjacências $\mathcal{E}$ com a solução atual (Linha 5), uma lista F contendo o conjunto de faces

Algoritmo 4.3: Face-edge dimpling

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacência $M_{n \times n}$, com arestas de pesos não-negativos.

Saída: Grafo planar maximal representado por $\mathcal{M}'$, filtrado a partir de M.

1 $\mathcal{C} \leftarrow$ Conjunto de possíveis tetraedros $\{v_1, v_2, v_3, v_4\}$, com $v_i \in V$;
2 $MaxWeight \leftarrow 0$;
3 **para** $(k \leftarrow 1 \leq |\mathcal{C}|)$ **faça**
4 $\mathcal{S} \leftarrow V \setminus \mathcal{C}_k$;
5 $\mathcal{E} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
6 $F \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
7 $\mathcal{H}_f \leftarrow \forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s, f\}, \forall f \in \mathcal{F}$;
8 $\mathcal{H}_e \leftarrow \forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s, e\}, \forall e \in \mathcal{E}$;
9 $TmpMax \leftarrow \sum w_e, \forall e \in \mathcal{E}$;
10 **enquanto** $(\mathcal{S} \neq \varnothing)$ **faça**
11 **se** $(\mathcal{H}_f \geq \mathcal{H}_e)$ **então**
12 Siga as etapas do *face dimpling*;
13 **senão**
14 Siga as etapas do *edge dimpling*;
15 $\mathcal{H}_f \leftarrow \mathcal{H}_f \cup \{\forall s \in \mathcal{S}$, calcule o ganho máximo de s para cada face adicionada$\}$;
16 $\mathcal{H}_e \leftarrow \mathcal{H}_e \cup \{\forall s \in \mathcal{S}$, calcule o ganho máximo de s para cada aresta adicionada$\}$;
17 **se** $(TmpMax \geq MaxWeight)$ **então**
18 $MaxWeight \leftarrow TmpMax$;
19 $\mathcal{M}' \leftarrow \mathcal{E}$;
20 **retorna** $\mathcal{M}'$.

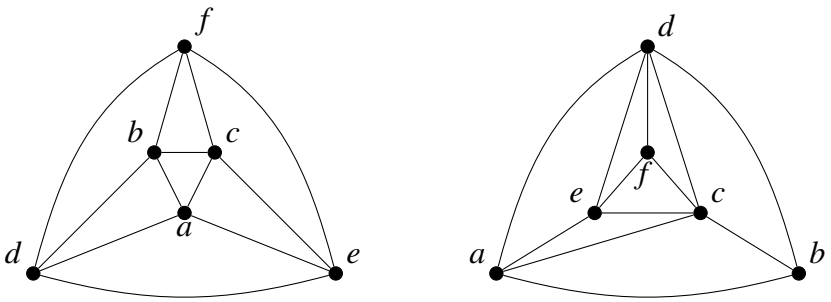
da semente (Linha 6) e uma estrutura de *heap* $\mathcal{H}_f$ contendo o par de vértice e face de maior ganho (Linha 7) e $\mathcal{H}_e$ contendo o par de vértice e aresta de maior ganho (Linha 8). Em seguida, para cada semente é calculada a soma dos pesos de suas arestas e este valor é armazenado em $TmpMax$ (Linha 9).

A cada passo verifica-se qual a melhor estratégia a seguir, fazendo isso de forma gulosa. Caso o ganho do *face dimpling* seja melhor ou igual ao *edge dimpling*, o primeiro é escolhido, seguindo as etapas da Linha 10 à 16. O *edge dimpling* é escolhido, caso contrário, seguindo as etapas da Linha 10 à 18. Em seguida, é necessário atualizar as *heaps* para a próxima iteração (Linhas 15 e 16). Este processo é repetido enquanto $\mathcal{S}$ não estiver vazio. Concluída a etapa anterior, é então verificado se, para o grafo gerado, a soma do pesos de suas arestas é maior que o valor armazenado em $MaxWeight$. Em caso afirmativo, o valor é atualizado e o grafo $\mathcal{E}$ que gerou a resposta é então armazenado em $\mathcal{M}'$ (Linhas 18 e 19).

Um detalhe importante relacionado à solução encontrada pelo Algoritmo 4.1 e de outros baseados em *dimpling*, é o fato de não garantirem que uma solução ótima seja obtida. Isso acontece porque, dada a natureza do algoritmo, o *face dimpling* gera apenas um subconjunto restrito de subgrafos planares maximais, chamados de Redes

-	b	c	d	e	f
a	2	2	2	2	1
b		2	2	1	2
c			1	2	2
d				2	2
e					2

Tabela 4.1: *Matriz de adjacências de um grafo K_6.*



(a) (b)

Figura 4.3: *(a) solução ótima correspondente ao grafo da Tabela 4.1 e (b) solução obtida pelo Algoritmo 4.1.*

Apolonianas [2]. Ou seja, esse procedimento sempre gera soluções com pelo menos um vértice de grau 3. Este fato pode ser visto na Tabela 4.1, que corresponde a matriz de adjacências de um grafo K_6. O Algoritmo 4.1 gera a solução da Figura 4.3(b), com pelo menos um vértice de grau 3. A soma do peso de suas arestas é igual a 23. No entanto, a solução ótima corresponde a Figura 4.3(a), onde todos os vértices possuem grau 4. A soma do peso de suas arestas é igual a 24.

No entanto, para este grafo em particular, o mesmo não ocorre com o Algoritmo 4.2, uma vez que sua estratégia sempre insere vértices de grau 4, sendo assim possível obter a solução da Figura 4.3(a). Cada abordagem tem suas vantagens, e estas irão depender da distribuição do pesos das arestas. Com isso, surgiu a ideia de combinar ambas estratégias através do Algoritmo 4.3, possibilitando uma nova diversidade de soluções, tendo o melhor dos dois mundos. A qualidade das soluções, i.e. peso do grafo solução, será no mínimo igual ou superior aos algoritmos executados separadamente.

4.4 Algoritmos paralelos

Como os algoritmos sequenciais processam os tetraedros de maneira totalmente independente, e com alto custo computacional resultante, nós estendemos estes algoritmos para um ambiente paralelo com p processadores e memória compartilhada. Embora direta, esta extensão requer um gerenciamento das sementes e dos resultados produzidos por

Algoritmo 4.4: Face dimpling paralelo

Entrada: Grafo completo $G = (V, E)$, representado pela matriz de adjacência $M_{n \times n}$, com arestas de pesos não-negativos.

Saída: Subgrafo planar maximal e o peso máximo obtido.

```
1  para todo id representando uma semente faça em paralelo
2      Obtenha os vértices que não estão na semente;
3      Construa o conjunto de faces e a fila de prioridade;
4      enquanto existirem vértices a serem inseridos faça
5          Encontre o vértice e face de ganho máximo;
6          Execute o face dimpling;
7          Remova o vértice escolhido da lista;
8          Adicione o peso das arestas inseridas ao peso máximo da instância;
9          Atualize a fila de prioridade;
10     Atualize o grafo planar maximal e o valor do peso máximo se, atráves de uma
       operação atômica, o valor obtido na instância for maior que o peso máximo;
11 1 retorna Subgrafo planar maximal e peso máximo.
```

cada instância. Além disso, existe o desafio de se trabalhar com uma estrutura irregular (grafo) em uma abordagem altamente paralela onde os custos de comunicação (acesso à memória compartilhada) costumam ser altos para acessos não aglutinados. Por conta disso, nós estruturamos os dados de faces e vértices como um vetor de estruturas para garantir acesso aglutinado (*coalesced*) por vários processadores.

O algoritmo paralelo resultante para o *face dimpling* é descrito pelo Algoritmo 4.4. Cada semente terá um identificador paralelo, com uma lista de vértices, um conjunto de faces e uma estrutura de fila de prioridade. Dada uma lista dos vértices que não pertencem à semente, um vértice é escolhido de forma que a sua inserção em uma face produza a maior pontuação a cada iteração (Linha 5). O vértice e a face são removidos de suas listas e dão origem a 3 novas faces a partir do vértice inserido (Linha 6). Ao final da iteração, cada identificador atualizará o valor do peso máximo caso este seja maior que o valor mais atual (Linha 10). Esta operação é crítica e, por isso, apenas um identificador deve executá-la por vez.

A seguir, o Algoritmo 4.5 descreve a versão paralela do *edge dimpling*. Cada semente terá um identificador paralelo, com uma lista de vértices, um conjunto de faces e uma estrutura de fila de prioridade. Dada uma lista dos vértices que não pertencem à semente, um vértice é escolhido de forma que a sua inserção em uma aresta produza a maior pontuação a cada iteração (Linha 5). O vértice e aresta são removidos de suas listas e dão origem a 4 novas faces a partir do vértice inserido (Linha 6). Ao final da iteração, cada identificador atualiza o valor do peso máximo caso este seja maior que o valor atual (Linha 10). Esta operação é crítica e, por isso, apenas um identificador deve executá-la por vez.

Algoritmo 4.5: Edge dimpling paralelo

Entrada: Grafo completo $G = (V, E)$, representado pela matriz de adjacência $M_{n \times n}$, com arestas de pesos não-negativos.
Saída: Subgrafo planar maximal e o peso máximo obtido.

```
1  para todo id representando uma semente faça em paralelo
2      Obtenha os vértices que não estão na semente;
3      Construa o conjunto de faces e a fila de prioridade;
4      enquanto existirem vértices a serem inseridos faça
5          Encontre o vértice e aresta de ganho máximo;
6          Execute o edge dimpling;
7          Remova o vértice escolhido da lista;
8          Adicione o peso das arestas inseridas ao peso máximo da instância;
9          Atualize a fila de prioridade;
10     Atualize o grafo planar maximal e o valor do peso máximo se, atráves de uma
         operação atômica, o valor obtido na instância for maior que o peso máximo;
11 retorna Subgrafo planar maximal e peso máximo.
```

Algoritmo 4.6: Face-edge dimpling paralelo

Entrada: Grafo completo $G = (V, E)$, representado por uma matriz de adjacência $M_{n \times n}$, com arestas de pesos não-negativos.
Saída: Subgrafo planar maximal e o peso máximo obtido.

```
1  para todo id representando uma semente faça em paralelo
2      Obtenha os vértices que não estão na semente;
3      Construa o conjunto de faces e as filas de prioridade;
4      enquanto existirem vértices a serem inseridos faça
5          se Ganho do face dimpling for maior ou igual ao do edge dimpling então
6              Siga as etapas do face dimpling;
7          senão
8              Siga as etapas do edge dimpling;
9          Remova o vértice escolhido da lista;
10         Adicione o peso das arestas inseridas ao peso máximo da instância;
11         Atualize as filas de prioridade;
12     Atualize o grafo planar maximal e o valor do peso máximo se, atráves de uma
         operação atômica, o valor obtido na instância for maior que o peso máximo;
13 retorna Subgrafo planar maximal e peso máximo.
```

Por último, o Algoritmo 4.6 descreve a versão paralela do *face-edge dimpling*. Cada semente terá um identificador paralelo, com uma lista de vértices, um conjunto de faces e uma estrutura de fila de prioridade para o *face dimpling* e outra para o *edge dimpling*. Dada uma lista dos vértices que não pertencem à semente, um vértice é escolhido de forma que a sua inserção em uma face ou em uma aresta produza a maior pontuação a cada iteração. Se o ganho de inserção na face for maior ou igual ao ganho de inserção na aresta, o primeiro é escolhido (Linha 5). O *face dimpling* é escolhido no caso

de empate devido sua iteração ser mais rápida.

No *face dimpling*, o vértice e a face são removidos de suas listas e dão origem a 3 novas faces a partir do vértice inserido. No *edge dimpling*, o vértice e a aresta são removidos de suas listas e dão origem a 4 novas faces a partir do vértice inserido. Ao final da iteração, cada identificador atualiza o valor do peso máximo caso este seja maior que o valor atual (Linha 12). Esta operação é crítica e, por isso, apenas um identificador deve executá-la por vez.

CAPÍTULO 5

Implementações

As implementações descritas a seguir foram construídas com base no modelo de Ahmadi et al. [1], descrito na Seção 2.5, e nos algoritmos propostos no Capítulo 4. Detalhes sobre a implementação do modelo de programação linear utilizando CPLEX [25] são apresentados, explicando como foi feita a escolha dos cortes da restrição 2-9 do Modelo de [1] com uso de OpenMP. Além disso, são fornecidos detalhes sobre a implementação dos algoritmos propostos, comentando as dificuldades encontradas em abstrair uma estrutura não-regular (grafo) para um ambiente paralelo em CUDA.

5.1 Programação linear para a solução exata

Dado que a quantidade de restrições (2-9) podem ser exponencialmente grande, [1] propôs uma maneira alternativa de *cutting plane* para seu modelo. As restrições (2-9) são inicialmente removidas, e então é feita uma etapa de relaxação no modelo resultante, levando-o à otimalidade. Se a solução ótima obtida não representa um subgrafo planar maximal, um subconjunto de restrições que foram violadas na solução atual são adicionados ao modelo. Então, uma nova rodada de otimização é executada com os cortes adicionados.

No entanto, [1] não construiu um código para esse procedimento, gerando os cortes manualmente em conjuntos de grafos com não mais que 24 vértices. Neste trabalho, um método de geração de cortes foi implementado para o modelo descrito. O método apresentou-se muito eficaz, proporcionando uma solução ótima para grafos com até 40 vértices em uma quantidade moderada de tempo, mesmo dependendo de um teste de planaridade. Existem vários testes de planaridade na literatura, tais como [6, 24, 38]. Para este trabalho, utilizamos a proposta de [6], por sua fácil implementação.

A implementação começa estabelecendo todas as restrições de 2-2 a 2-8 e 2-10 de acordo com o conjunto de entrada. Então, após a otimização do modelo, é verificado se a solução inteira produzida representa um subgrafo planar maximal através de um teste de planaridade. Em caso afirmativo, a solução é dada pelas x variáveis utilizadas no teste de planaridade. Caso contrário, é necessário adicionar uma restrição ao modelo, chamada

de corte. O corte é dado pelo menor subrafo planar maximal da solução inteira, que não é planar.

Para adicionar um corte, é necessário gerar combinações de vértices, começando a partir de $n = 4$. Um grafo induzido dos vértices combinados é construído, adicionando somente as arestas presentes no grafo não-planar da solução inteira obtida, e então um teste de planaridade é executado para esse grafo. Caso o subgrafo não tenha sido utilizado anteriormente como uma restrição e seja planar maximal, este é escolhido como corte. Caso contrário, outro subgrafo com n vértices é gerado e o processo se repete até que um subgrafo seja encontrado. Este processo é feito de forma paralela para cada combinação, utilizando OpenMP e uma técnica chamada de *combinadic* [27] para gerar as combinações.

Se todas as combinações possíveis com n vértices foram testadas, o valor de n é incrementado em 1 para gerar combinações com $n+1$ vértices, até que um corte utilizável seja encontrado. Após a adição de um corte, o modelo é submetido a uma nova etapa de otimização, produzindo outra solução inteira. Este processo é então repetido até que a solução inteira encontrada represente um subgrafo planar maximal.

Por curiosidade, foi feito um teste onde o corte gerado era submetido a um critério diferente de escolha. Além de ser um subgrafo planar maximal, este deveria ser o menor subrafo planar máximo. Ou seja, o menor grafo que não seja subgrafo de nenhum outro que seja um candidato a corte. Para isso, foi necessário gerar todos os subgrafos de $n = 4$ a $|V| - 1$, com $|V|$ sendo a quantidade de vértices do grafo de entrada, que sejam possíveis cortes. Essa abordagem gerou uma quantidade menor de iterações, contudo, e mostrou-se muito lento para $n > 25$ porque muitos subgrafos eram gerados em cada iteração.

5.2 Face dimpling

O Algoritmo 4.4 foi mapeado para executar em uma arquitetura *manycore* representada por uma GPU. A preparação dos dados e o recebimento do resultado são descritos na Implementação 5.1, chamada de *Pré e Pós-processamento*. É necessário alocar uma quantidade de memória para os seguintes dados: grafo $\mathcal{M}$, F' que armazenará o índice da semente que resultou a solução, $MaxWeight$ que contém o maior peso obtido dentre os subgrafos gerados, e uma estrutura $\mathcal{T}$ responsável por armazenar as instâncias de F e $\mathcal{S}$ para cada semente em $\mathcal{C}$. Em seguida, copia-se o grafo para a memória alocada na GPU. A chamada ao *kernel* é então realizada, passando todos os parâmetros alocados anteriormente. Conforme experimentos realizados, os melhores tempos de execução foram obtidos quando a quantidade de *threads* por bloco foi de $2^{10} = 1024$ e a quantidade de blocos foi igual a $\lceil \frac{|\mathcal{C}|}{2^{10}} \rceil$.

Em versões iniciais de implemetação, as sementes eram geradas na CPU e depois eram todas copiadas para a GPU, gerando um *overhead* adicional na cópia e desperdício de memória, pois cada semente seria acessada uma única vez na inicialização. Com isso, outras alternativas foram buscadas. A saída encontrada foi gerar cada semente na própria GPU através da técnica de *combinadic*, utilizada nas implementações descritas neste capítulo.

Na primeira versão proposta, descrita pela Implementação 5.2, cada *thread* é responsável pelo crescimento de uma semente de forma paralela. Note que as etapas descritas nas Linhas 3 a 15 correspondem a um mapeamento do Algoritmo 4.1, adicionando-se o índice da *thread* correspondente a execução. Concluída a etapa anterior, cada *thread* ficará retida em uma barreira de sincronização (Linha 16), esperando as demais de seu bloco concluírem a execução. Em seguida, uma operação atômica para encontrar o valor máximo é realizada, comparando os valores armazenados em $TmpMax$ e $MaxWeight$, de forma a garantir que somente uma *thread* realize a escrita em $MaxWeight$ por vez (Linhas 17 a 19).

A matriz $\mathcal{E}$, presente no algoritmo, foi removida da implementação para economizar memória, sendo necessário apenas o valor da soma das arestas que compõem a semente. Por isso, caso o valor de $MaxWeight$ seja atualizado, o índice da *thread* que realizou a atualização é armazenado em F'. Dessa forma, é possível reconstruir o grafo resultante a partir do índice da *thread* responsável por construir a melhor solução.

Uma das otimizações realizadas, descrita pela Implementação 5.3 consiste em utilizar um espaço alocado na memória compartilhada da GPU para armazenar o grafo. Esta memória, embora pequena, é 100 vezes mais rápida do que a memória global. Dessa forma, ao invés de acessar a memória global, o acesso é realizado na memória

Implementação 5.1: Pré e Pós-processamento - Passo da CPU

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n\times n}$ com arestas de pesos não-negativos.

Saída: Conjunto de faces F', representando um grafo planar maximal e $MaxWeight$ com o peso máximo obtido.

1 Aloca memória para $MaxWeight$ na GPU;
2 Aloca memória para F' na GPU;
3 Aloca memória para a matriz $\mathcal{M}_{n\times n}$ na GPU;
4 Aloca memória para uma estrutura $\mathcal{T}$, contendo F e $\mathcal{S}$ para cada instância de $\mathcal{C}$, na GPU;
5 Copia a matriz $\mathcal{M}_{n\times n}$ para a GPU;
6 $dimBloco \leftarrow 2^{10}$;
7 $dimGrid \leftarrow \lceil \frac{|\mathcal{C}|}{dimBloco} \rceil$;
8 Invoca o kernel $< dimGrid, dimBloco >(\mathcal{M}_{n\times n}, \mathcal{T}, MaxWeight, F')$;
9 Copia F' e $MaxWeight$ da GPU para a CPU;
10 Libera o espaço da memória da GPU ocupado por $\mathcal{M}_{n\times n}$, F', $\mathcal{T}$ e $MaxWeight$;
11 **retorna** F' e $MaxWeight$.

Implementação 5.2: Kernel - Face dimpling paralelo

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n\times n}$ com arestas de pesos não-negativos e um conjunto de sementes $\mathcal{C}$.

Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e *MaxWeight*, com o peso máximo obtido.

```
1  idx ← Índice de uma thread;
2  para todo idx ≤ |C| faça
3      T_{S_idx} ← V \ C_idx;
4      E ← {{v1,v2},{v1,v3},{v1,v4},{v2,v3},{v2,v4},{v3,v4}};
5      T_{F_idx} ← {{v1,v2,v3},{v1,v2,v4},{v1,v3,v4},{v2,v3,v4}};
6      T_{H_idx} ← ∀s ∈ T_{S_idx}, calcule o ganho máximo de {s,f}, ∀f ∈ T_{F_idx};
7      T_{TmpMax_idx} ← Σ w_e, ∀e ∈ E;
8      enquanto T_{S_idx} ≠ ∅ faça
9          {v_i, f_i} ← max{T_{H_idx}}, v_i ∈ T_{S_idx} e f_i ∈ T_{F_idx};
10         T_{H_idx} ← T_{H_idx} \ max{T_{H_idx}};
11         Execute o dimpling;
12         T_{S_idx} ← T_{S_idx} \ {v_i};
13         T_{F_idx} ← (T_{F_idx} \ {f_i}) ∪ {f_a, f_b, f_c};
14         T_{TmpMax_idx} ← T_{TmpMax_idx} + w_{G_{v_i,v_x}} + w_{G_{v_i,v_y}} + w_{G_{v_i,v_z}}, com v_x, v_y, v_z ∈ f_i;
15         T_{H_idx} ← T_{H_idx} ∪ {∀s ∈ T_{S_idx}, calcule o ganho máximo de {s,f_a},{s,f_b},{s,f_c}};
16     Execute a barreira de sincronização;
17     se T_{TmpMax_idx} ≥ MaxWeight então
18         MaxWeight ← T_{TmpMax_idx};
19         F' ← idx;
20 retorna F' e MaxWeight.
```

compartilhada, acelerando o trabalho de todas as *threads*.

Outro detalhe está na presença de um acesso aglutinado no conjunto de faces, uma vez que este está armazenado na GPU e será utilizado com muita frequência. Tal organização de dados consiste em colocar todo um conjunto de faces de um determinado índice sequencialmente. Isto é, em cada instrução de *warp*, todas as *threads* buscarão o mesmo índice do conjunto de faces. Dessa forma, diminui-se a latência de acesso à memória global.

A última melhoria, correspondendo a uma terceira implementação, consiste em dividir a tarefa entre várias GPUs. Cada GPU fica responsável por uma fatia das sementes, de forma que quanto mais GPUs estejam disponíveis, maior a distribuição e mais rápida a execução. Para isso, é necessário calcular a quantidade de memória que será gasta em cada execução de *kernel*, dividir fatias de sementes para cada GPU e passar o *offset* de cada fatia para suas respectivas GPUs.

Após concluída a etapa de execução do *kernel*, os valores de F' e *MaxWeight* são copiados para a memória principal da CPU e cada valor de *MaxWeight* é comparado com os obtidos em cada GPU, sendo o maior deles a resposta. A matriz resultante é então reconstruída atráves do F' que resultou no maior *MaxWeight*.

Implementação 5.3: Kernel - Face dimpling paralelo com Memória Compartilhada

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não-negativos e um conjunto de sementes $\mathcal{C}$.
Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e $MaxWeight$, com o peso máximo obtido.

1 $idx \leftarrow$ Índice de uma *thread*;
2 Aloca G' na memória compartilhada;
3 $G' \leftarrow G$;
4 Execute a barreira de sincronização;
5 **para todo** $idx \leq |C|$ **faça**
6 $\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow V \setminus \mathcal{C}_{idx}$;
7 $\quad \mathcal{E} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
8 $\quad \mathcal{T}_{F_{idx}} \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
9 $\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, f\}, \forall f \in \mathcal{T}_{F_{idx}}$;
10 $\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \sum w_e, \forall e \in \mathcal{E}$;
11 $\quad$ **enquanto** $\mathcal{T}_{\mathcal{S}_{idx}} \neq \varnothing$ **faça**
12 $\quad\quad \{v_i, f_i\} \leftarrow \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$, $v_i \in \mathcal{T}_{\mathcal{S}_{idx}}$ e $f_i \in \mathcal{T}_{F_{idx}}$;
13 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \setminus \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$;
14 $\quad\quad$ Execute o *dimpling*;
15 $\quad\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow \mathcal{T}_{\mathcal{S}_{idx}} \setminus \{v_i\}$;
16 $\quad\quad \mathcal{T}_{F_{idx}} \leftarrow (\mathcal{T}_{F_{idx}} \setminus \{f_i\}) \cup \{f_a, f_b, f_c\}$;
17 $\quad\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \mathcal{T}_{TmpMax_{idx}} + w_{G'_{v_i,v_x}} + w_{G'_{v_i,v_y}} + w_{G'_{v_i,v_z}}$, com $v_x, v_y, v_z \in f_i$;
18 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \cup \{\forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, f_a\}, \{s, f_b\}, \{s, f_c\}\}$;
19 $\quad$ Execute a barreira de sincronização;
20 $\quad$ **se** $\mathcal{T}_{TmpMax_{idx}} \geq MaxWeight$ **então**
21 $\quad\quad MaxWeight \leftarrow \mathcal{T}_{TmpMax_{idx}}$;
22 $\quad\quad F' \leftarrow idx$;
23 **retorna** F' e $MaxWeight$.

5.3 Edge dimpling

De forma similar, o Algoritmo 4.5 foi mapeado para executar em uma arquitetura *manycore*. A principal diferença consiste na forma como o *edge dimpling* foi implementado. O desafio foi maior, pois diversas estruturas presentes na biblioteca padrão do C++ (STL) tiveram que ser reimplementadas na GPU. Além dos elementos já conhecidos, outros dois são incluídos para um controle adicional das etapas presentes no *edge dimpling*: $\mathcal{E}$ para armazenar as arestas presentes na solução e $\mathcal{L}$ para armazenar as faces que cada aresta pertence. Por isso, precisou-se de uma etapa de preparação e recebimento dos dados diferente do *face dimpling*, descrita na Implementação 5.4, chamada de *Pré e Pós Processamento*.

Primeiramente, é necessário alocar uma quantidade de memória para os seguintes dados: grafo $\mathcal{M}$, F' que armazenará o índice da semente que resultou a solução, $MaxWeight$ que contém o maior peso obtido dentre os grafos gerados e uma estrutura $\mathcal{T}$ responsável por armazenar as instâncias de F, $\mathcal{S}$, $\mathcal{E}$ e $\mathcal{L}$ para cada semente. Em seguida, copia-se o grafo para a memória alocada na GPU. A chamada ao *kernel* é então

Implementação 5.4: Pré e Pós Processamento - Passo da CPU

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não-negativos.

Saída: Conjunto de faces F', representando um grafo planar maximal e $MaxWeight$ com o peso máximo obtido.

1 Aloca memória para $MaxWeight$ na GPU;
2 Aloca memória para F' na GPU;
3 Aloca memória para a matriz $\mathcal{M}_{n \times n}$ na GPU;
4 Aloca memória para uma estrutura $\mathcal{T}$, contendo F, $\mathcal{S}$, $\mathcal{E}$ e $\mathcal{L}$ para cada instância de $\mathcal{C}$, na GPU;
5 Copia a matriz $\mathcal{M}_{n \times n}$ para a GPU;
6 $dimBloco \leftarrow 2^{10}$;
7 $dimGrid \leftarrow \lceil \frac{|\mathcal{C}|}{dimBloco} \rceil$;
8 Invoca o kernel $< dimGrid, dimBloco >(\mathcal{M}_{n \times n}, \mathcal{T}, MaxWeight, F')$;
9 Copia F' e $MaxWeight$ da GPU para a CPU;
10 Libera o espaço da memória da GPU ocupado por $\mathcal{M}_{n \times n}$, F', $\mathcal{T}$ e $MaxWeight$;
11 **retorna** F' e $MaxWeight$.

realizada, passando todos os parâmetros alocados anteriormente. De forma similar, os melhores tempos de execução foram obtidos quando a quantidade de *threads* por bloco foi de $2^{10} = 1024$ e a quantidade de blocos foi igual a $\lceil \frac{|\mathcal{C}|}{2^{10}} \rceil$.

Na primeira versão proposta, descrita pela Implementação 5.5, cada *thread* é responsável pelo crescimento de uma semente de forma paralela. Note que as etapas descritas nas Linhas 11 a 19 correspondem a um mapeamento do Algoritmo 4.2, adicionando-se o índice da *thread* correspondente a execução e estruturas para auxiliar no processo de *edge dimpling*. Aqui, foi codificada uma estrutura de aresta para auxiliar no mapeamento e uma estrutura de *hash* em *bucket* $\mathcal{B}$, funcionando como um *set* do C++ STL.

Uma vez que cada aresta pertence a exatamente duas faces, no momento da remoção de uma aresta, 4 vértices ficam órfãos de uma face. No momento que se insere um novo vértice, é necessário identificar quem são esses 4 vértices. Uma das funções do *hash* consiste em filtrar os vértices do conjunto de faces que a aresta e_i pertence (Linha 17). A outra função do *hash* consiste em verificar se um vértice já foi utilizado no *dimpling*, impedindo que ele pertença a número de faces diferente de 2. Concluída a etapa anterior, cada *thread* ficará retida em uma barreira de sincronização (Linha 20), esperando as demais de seu bloco concluírem a execução.

Em seguida, uma operação atômica para encontrar o valor máximo é realizada, comparando os valores armazenados em $TmpMax$ e $MaxWeight$, de forma a garantir que somente uma *thread* realize a escrita em $MaxWeight$ por vez (Linhas 21 a 23). Ao contrário das Implementações 5.2 e 5.3, a matriz $\mathcal{E}$ foi mantida aqui. Por isso, caso o valor de $MaxWeight$ seja atualizado, o índice da *thread* que realizou a atualização é armazenado em F'. Dessa forma, é possível reconstruir o grafo resultante a partir do índice da *thread*

responsável por construir a melhor solução.

De forma similar à Implementação 5.3, também foi feita uma otimização que consiste em utilizar um espaço alocado na memória compartilhada da GPU para armazenar o grafo, descrita pela Implementação 5.6. O acesso aglutinado neste não foi implementado, pois em cada iteração é necessário atualizar o conjunto de faces e arestas, o que torna os acessos a memória mais caros e difíceis de mapear à nível de *warp*.

A última melhoria, correspondendo a uma terceira implementação, consiste em dividir a tarefa entre várias GPUs. Cada GPU fica responsável por uma fatia das sementes, de forma que quanto mais GPUs estejam disponíveis, maior a distribuição e mais rápida a execução. Para isso, é necessário calcular a quantidade de memória que será gasta em cada execução de *kernel*, dividir fatias de sementes para cada GPU e passar o *offset* de cada fatia para suas respectivas GPUs.

Implementação 5.5: Kernel - Edge dimpling paralelo

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não-negativos e um conjunto de sementes $\mathcal{C}$.

Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e $MaxWeight$, com o peso máximo obtido.

1 $idx \leftarrow$ Índice de uma *thread*;
2 **para todo** $idx \leq |C|$ **faça**
3 $\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow V \setminus \mathcal{C}_{idx}$;
4 $\quad \mathcal{T}_{\mathcal{E}_{idx}} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
5 $\quad \mathcal{T}_{F_{idx}} \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
6 $\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, e\}, \forall e \in \mathcal{T}_{\mathcal{E}_{idx}}$;
7 $\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \sum w_e, \forall e \in \mathcal{T}_{\mathcal{E}_{idx}}$;
8 $\quad$ **enquanto** $\mathcal{T}_{\mathcal{S}_{idx}} \neq \varnothing$ **faça**
9 $\quad\quad \{v_i, e_i\} \leftarrow \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$, $v_i \in \mathcal{T}_{\mathcal{S}_{idx}}$ e $e_i \in \mathcal{T}_{E_{idx}}$;
10 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \setminus \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$;
11 $\quad\quad$ Execute o *dimpling*;
12 $\quad\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow \mathcal{T}_{\mathcal{S}_{idx}} \setminus \{v_i\}$;
13 $\quad\quad \mathcal{F} \leftarrow \mathcal{T}_{\mathcal{L}_{idx}}[e_i]$;
14 $\quad\quad \forall f \in \mathcal{F}$ e $f = \{u, v, w\}$, insere u, v e w em $\mathcal{B}$;
15 $\quad\quad \mathcal{T}_{F_{idx}} \leftarrow \mathcal{T}_{F_{idx}} \setminus \mathcal{F}$;
16 $\quad\quad \mathcal{T}_{\mathcal{E}_{idx}} \leftarrow (\mathcal{T}_{\mathcal{E}_{idx}} \setminus \{e_i\}) \cup \{\{v_i, v_x\}, \{v_i, v_y\}, \{v_i, v_w\}, \{v_i, v_z\}\}$, com $v_x, v_y, v_w, v_z \in \mathcal{B}$;
17 $\quad\quad \mathcal{T}_{F_{idx}} \leftarrow \mathcal{T}_{F_{idx}} \cup \{f_a, f_b, f_c, f_d\}$;
18 $\quad\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \mathcal{T}_{TmpMax_{idx}} - w_{e_i} + w_{G_{v_i,v_x}} + w_{G_{v_i,v_y}} + w_{G_{v_i,v_w}} + w_{G_{v_i,v_z}}$, com $v_x, v_y, v_w, v_z \in \mathcal{B}$;
19 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \cup \{\forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, \{v_i, v_x\}\}, \{s, \{v_i, v_y\}\}, \{s, \{v_i, v_w\}\}, \{s, \{v_i, v_z\}\}\}$;
20 $\quad$ Execute a barreira de sincronização;
21 $\quad$ **se** $\mathcal{T}_{TmpMax_{idx}} \geq MaxWeight$ **então**
22 $\quad\quad MaxWeight \leftarrow \mathcal{T}_{TmpMax_{idx}}$;
23 $\quad\quad F' \leftarrow idx$;
24 **retorna** F' e $MaxWeight$.

Implementação 5.6: Kernel - Edge dimpling paralelo com Memória Compartilhada

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não-negativos e um conjunto de sementes $\mathcal{C}$.
Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e *MaxWeight*, com o peso máximo obtido.

1 $idx \leftarrow$ Índice de uma *thread*;
2 Aloca G' na memória compartilhada;
3 $G' \leftarrow G$;
4 Executa a barreira de sincronização;
5 **para todo** $idx \leq |C|$ **faça**
6 $\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow V \setminus \mathcal{C}_{idx}$;
7 $\quad \mathcal{T}_{\mathcal{E}_{idx}} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
8 $\quad \mathcal{T}_{\mathcal{F}_{idx}} \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
9 $\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, e\}, \forall e \in \mathcal{T}_{\mathcal{E}_{idx}}$;
10 $\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \sum w_e, \forall e \in \mathcal{T}_{\mathcal{E}_{idx}}$;
11 $\quad$ **enquanto** $\mathcal{T}_{\mathcal{S}_{idx}} \neq \varnothing$ **faça**
12 $\quad\quad \{v_i, e_i\} \leftarrow \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$, $v_i \in \mathcal{T}_{\mathcal{S}_{idx}}$ e $e_i \in \mathcal{T}_{E_{idx}}$;
13 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \setminus \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$;
14 $\quad\quad$ Execute o *dimpling*;
15 $\quad\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow \mathcal{T}_{\mathcal{S}_{idx}} \setminus \{v_i\}$;
16 $\quad\quad \mathcal{F} \leftarrow \mathcal{T}_{\mathcal{L}_{idx}}[e_i]$;
17 $\quad\quad \forall f \in \mathcal{F}$ e $f = \{u, v, w\}$, insere u, v e w em $\mathcal{B}$;
18 $\quad\quad \mathcal{T}_{\mathcal{F}_{idx}} \leftarrow \mathcal{T}_{\mathcal{F}_{idx}} \setminus \mathcal{F}$;
19 $\quad\quad \mathcal{T}_{\mathcal{E}_{idx}} \leftarrow (\mathcal{T}_{\mathcal{E}_{idx}} \setminus \{e_i\}) \cup \{\{v_i, v_x\}, \{v_i, v_y\}, \{v_i, v_w\}, \{v_i, v_z\}\}$, com $v_x, v_y, v_w, v_z \in \mathcal{B}$;
20 $\quad\quad \mathcal{T}_{\mathcal{F}_{idx}} \leftarrow \mathcal{T}_{\mathcal{F}_{idx}} \cup \{f_a, f_b, f_c, f_d\}$;
21 $\quad\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \mathcal{T}_{TmpMax_{idx}} - w_{e_i} + w_{G'_{v_i,v_x}} + w_{G'_{v_i,v_y}} + w_{G'_{v_i,v_w}} + w_{G'_{v_i,v_z}}$, com $v_x, v_y, v_w, v_z \in \mathcal{B}$;
22 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \cup \{\forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, \{v_i, v_x\}\}, \{s, \{v_i, v_y\}\}, \{s, \{v_i, v_w\}\}, \{s, \{v_i, v_z\}\}\}$;
23 $\quad$ Execute a barreira de sincronização;
24 $\quad$ **se** $\mathcal{T}_{TmpMax_{idx}} \geq MaxWeight$ **então**
25 $\quad\quad MaxWeight \leftarrow \mathcal{T}_{TmpMax_{idx}}$;
26 $\quad\quad F' \leftarrow idx$;
27 **retorna** F' e *MaxWeight*.

Após concluída a etapa de execução do *kernel*, os valores de F' e *MaxWeight* são copiados para a memória principal da CPU e cada valor de *MaxWeight* é comparado com os obtidos em cada GPU, sendo o maior deles a resposta. A matriz resultante é então reconstruída atráves do F' que resultou no maior *MaxWeight*.

5.4 Face-edge dimpling

Por último, foi feita uma implementação do Algoritmo 4.6, descrita pela Implementação 5.7. Diferente das implementações anteriores, esta foi mais direta, uma vez com o *face* e *edge dimpling* implementados, bastou combiná-los em um único código. A Implementação 5.4, correspondendo ao Pré e Pós-processamento, foi aproveitada para o *face-edge dimpling*, pois este detinha de todas as estruturas necessárias pelas Implemen-

Implementação 5.7: Kernel - Face-edge dimpling paralelo

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não-negativos e um conjunto de sementes $\mathcal{C}$.

Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e *MaxWeight*, com o peso máximo obtido.

```
1  idx ← Índice de uma thread;
2  para todo idx ≤ |C| faça
3      T_{S_idx} ← V \ C_idx;
4      T_{E_idx} ← {{v1,v2},{v1,v3},{v1,v4},{v2,v3},{v2,v4},{v3,v4}};
5      T_{F_idx} ← {{v1,v2,v3},{v1,v2,v4},{v1,v3,v4},{v2,v3,v4}};
6      T_{Hf_idx} ← ∀s ∈ T_{S_idx}, calcule o ganho máximo de {s,f}, ∀f ∈ T_{F_idx};
7      T_{He_idx} ← ∀s ∈ T_{S_idx}, calcule o ganho máximo de {s,e}, ∀e ∈ T_{E_idx};
8      T_{TmpMax_idx} ← Σ w_e, ∀e ∈ T_{E_idx};
9      enquanto T_{S_idx} ≠ ∅ faça
10         se (max{T_{Hf_idx}} ≥ max{T_{He_idx}}) então
11             Siga as etapas do face dimpling;
12         senão
13             Siga as etapas do edge dimpling;
14         T_{Hf_idx} ← T_{Hf_idx} ∪ {∀s ∈ T_{S_idx}, calcule o ganho máximo de s para cada face
             adicionada};
15         T_{He_idx} ← T_{He_idx} ∪ {∀s ∈ T_{S_idx}, calcule o ganho máximo de s para cada aresta
             adicionada};
16     Execute a barreira de sincronização;
17     se T_{TmpMax_idx} ≥ MaxWeight então
18         MaxWeight ← T_{TmpMax_idx};
19         F' ← idx;
20 retorna F' e MaxWeight.
```

tações 5.2 e 5.5. A quantidade de *threads* por bloco foi mantida em $2^{10} = 1024$, sendo a quantidade de blocos igual a $\lceil \frac{|\mathcal{C}|}{2^{10}} \rceil$.

De forma similar à Implementação 5.3, também foi feita uma otimização que consiste em utilizar um espaço alocado na memória compartilhada da GPU para armazenar o grafo. O acesso aglutinado neste não foi implementado, pois em cada iteração é necessário atualizar o conjunto de faces e arestas, o que torna os acessos a memória mais caros e difíceis de mapear à nível de *warp*.

A última melhoria, correspondendo a uma terceira implementação, consiste em dividir a tarefa entre várias GPUs. Cada GPU fica responsável por uma fatia das sementes, de forma que quanto mais GPUs estejam disponíveis, maior a distribuição e mais rápida a execução. Para isso, é necessário calcular a quantidade de memória que será gasta em cada execução de *kernel*, dividir fatias de sementes para cada GPU e passar o *offset* de cada fatia para suas respectivas GPUs.

CAPÍTULO 6

Experimentos realizados

Neste capítulo, serão discutidos os resultados obtidos a partir das implementações descritas no Capítulo 5. Detalhes sobre o ambiente utilizado para os testes, bem como sobre as instâncias utilizadas são abordados. Além disso, é feita uma análise comparativa, apresentando as vantagens e desvantagens da solução proposta frente os resultados obtidos por Tumminello et al. [46] e Massara et al. [33].

6.1 Configuração dos experimentos

Os experimentos foram conduzidos em um computador com processador Intel Xeon E5-2620 2GHz, 16GB de ECC RAM, sistema operacional CentOS 7.2.1511 64-bits, e quatro (4) GeForce Zotac Nvidia GTX Titan Black, com 6GB de RAM e 2.880 CUDA *cores* cada. Foram usados os compiladores GCC 4.9.2 e CUDA Toolkit 7.5 para os códigos da CPU e GPU, respectivamente, com a *flag* $O3$. Os experimentos foram realizados em caráter de dedicação total, ou seja, sem interferência de outros usuários.

Os tempos calculados levam em consideração a execução do programa como um todo, incluindo entrada e saída, e não apenas o algoritmo. Cada instância foi executada 10 vezes e delas foi calculado o tempo médio de execução.

6.2 Instâncias sintéticas

Para os testes, um total de 19 grafos completos foram gerados, variando entre 10 e 100 vértices, em múltiplos de cinco. Para cada aresta, foi atribuído um peso gerado aleatoriamente entre os valores $[0, 199]$. Os experimentos apresentados a seguir visam o *face* e o *face-edge dimpling*, uma vez que o uso exclusivo do *edge dimpling* não apresentou resultados significativos em relação a tempo e valor da solução.

6.2.1 Face dimpling

A Tabela 6.1 apresenta os tempos de execução para a implementação sequencial do Algoritmo 4.1 e da Implementação 5.3, com apenas uma GPU e outra com quatro GPUs. Podemos perceber que as versões paralelas começam a apresentar ganho de desempenho frente à versão sequencial a partir de 25 e 35 vértices, respectivamente. Além disso, ambas versões levam menos de 2 segundos para processar instâncias com até 55 vértices. A medida que as instâncias crescem, nota-se o quão escalável a versão multi-GPU (com 4 GPUs) é frente a versão com somente uma GPU, sendo cerca de quatro vezes mais rápida na instância com 100 vértices. Este fato pode ser observado na Figura 6.1, onde um comparativo de tempo entre as implementações paralelas é ilustrado.

É notável o desempenho da implementação multi-GPU, que apresenta um aumento de poucos segundos de uma instância para outra, enquanto a versão mais simples (sequencial) tem um aumento de quase 30 minutos de 95 a 100 vértices. A partir de 90 vértices, a quantidade de memória necessária para a instância ultrapassa os 6GB disponíveis na GPU. Sendo assim, foi necessário dividir a execução em lotes (*batch*), adicionando uma comunicação extra com a CPU, ocasionando um aumento significativo no tempo de execução.

A Tabela 6.1 apresenta também o *speedup* obtido pelas versões paralelas quando

	Sequencial	**Paralelo (1 GPU)**		**Paralelo (4 GPUs)**		**1 GPU**
n	**Tempo** (s)	**Tempo** (s)	**Speedup**	**Tempo** (s)	**Speedup**	**vs. 4 GPUs**
10	0,001	0,368	0,003	1,624	0,001	0,245
15	0,015	0,362	0,041	1,639	0,009	0,246
20	0,092	0,365	0,252	1,607	0,057	0,257
25	0,387	0,370	1,047	1,622	0,239	0,250
30	1,199	0,375	3,197	1,616	0,742	0,256
35	3,394	0,412	8,240	1,631	2,081	0,285
40	8,580	0,489	17,553	1,645	5,216	0,334
45	19,597	0,663	29,567	1,715	11,424	0,419
50	41,327	0,995	41,543	1,676	24,654	0,611
55	80,842	1,698	47,610	1,872	43,194	0,944
60	149,241	2,801	53,274	2,145	69,576	1,364
65	263,207	4,309	**61,080**	2,631	100,025	1,700
70	443,737	7,844	56,571	3,415	129,934	2,313
75	723,145	12,733	56,794	4,649	155,562	2,816
80	1.140,939	20,062	56,872	6,335	180,101	3,197
85	1.750,306	30,921	56,606	8,897	196,719	3,513
90	2.617,484	47,276	55,367	12,724	205,712	3,753
95	3.835,536	69,420	55,251	18,165	211,146	3,907
100	5.514,923	100,670	54,782	25,770	**214,005**	**4,006**

Tabela 6.1: *Tempos de execução e speedups do face dimpling.*

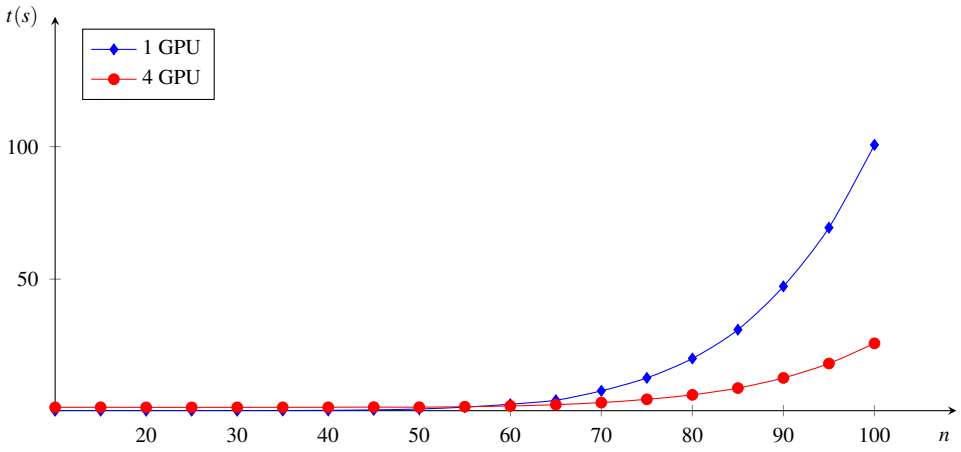


Figura 6.1: *Comparativo de tempo entre as implementações paralelas do face dimpling.*

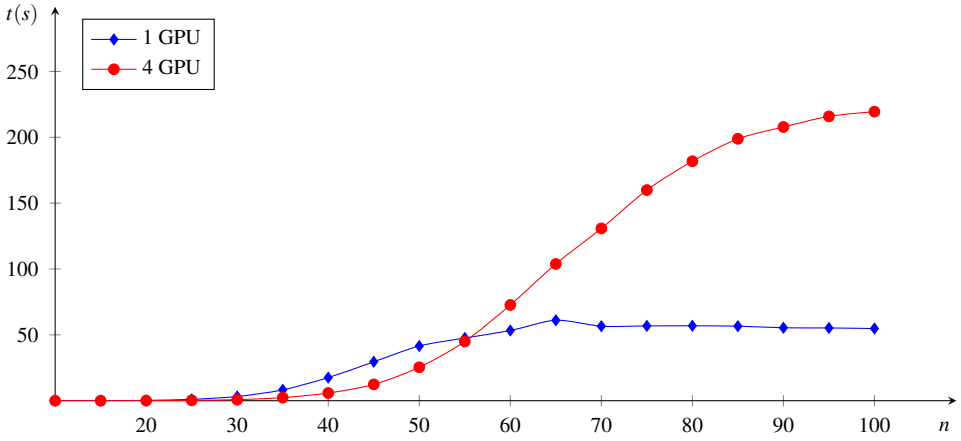


Figura 6.2: *Speedup das implementações paralelas do face dimpling em relação a versão sequencial.*

comparadas a proposta sequencial. A implementação com uma GPU obteve seu melhor *speedup* para a instância de 65 vértices, sendo 61 vezes mais rápida. Além disso, é 54-56 vezes mais rápida para instâncias onde $70 \leqslant n \leqslant 100$. Por outro lado, o *speedup* da versão multi-GPU somente cresceu, sendo 214 vezes mais rápida para $n = 100$. A partir de 60 vértices, a versão multi-GPU começa a obter um melhor desempenho. Esta afirmação fica clara analisando a Figura 6.2.

Um fato curioso é mostrado na Tabela 6.1: o *speedup* da versão multi-GPU frente à versão com uma GPU. Era de se esperar que, com o recurso computacional multiplicado em 4 (de uma para quatro GPUs), o *speedup* seria proporcional. No entanto, isso só acontece na instância onde $n = 100$ (através de *speedup* superlinear), uma vez que o custo de comunicação entre os processadores (cada um dos quatro disparando um *kernel*) torna-se "desprezível" em instâncias maiores.

6.2.2 Face-edge dimpling

Em seguida, a Tabela 6.2 apresenta os tempos de execução e *speedups* para a implementação sequencial do Algoritmo 4.3 e da Implementação 5.7 com memória compartilhada, com apenas uma GPU e outra com quatro GPUs. Neste caso, as implementações paralelas começam a sobressair-se perante a versão sequencial em instâncias menores, 15 e 20 vértices respectivamente. Ambas versões levam menos de 2 segundos para processar instâncias com até 25 vértices, valor menor que o *face dimpling*. Na Figura 6.3, mesmo utilizando uma escala diferente, é possível observar uma similaridade com a Figura 6.1. A tendência é que a curva representada pela versão multi-GPU tenha um crescimento menor quando comparada a versão com uma GPU.

O *speedup* obtido foi pouco se comparado ao *face dimpling*. Isso acontece porque, com apenas 35 vértices, a quantidade de memória necessária ultrapassa os 6GB disponíveis em uma GPU. A GPU fica muito ociosa, não utilizando toda sua capacidade de processamento. Além disso, a quantidade de acessos a memória global é maior, uma vez que a quantidade de estruturas necessárias para o processamento aumenta, limitando o *speedup* máximo em 30,582 e 112,126 vezes, para uma e quatro GPUs respectivamente. Por isso, a versão multi-GPU obtém melhor desempenho em instâncias maiores de 30 vértices, como mostra a Figura 6.4. Em instâncias a partir de 50 vértices, a versão multi-

	Sequencial	**Paralelo (1 GPU)**		**Paralelo (4 GPUs)**		**1 GPU**
n	**Tempo (*s*)**	**Tempo (*s*)**	**Speedup**	**Tempo (*s*)**	**Speedup**	**vs. 4 GPUs**
10	0,054	0,306	0,176	1,469	0,043	0,208
15	0,665	0,373	1,783	1,476	0,522	0,253
20	4,824	0,479	10,072	1,631	3,358	0,294
25	24,789	1,126	22,015	1,814	15,850	0,621
30	95,365	3,508	27,185	2,300	45,272	1,525
35	309,696	10,379	29,839	3,969	80,409	2,615
40	812,179	26,738	30,375	8,206	97,407	3,258
45	1.963,768	64,214	**30,582**	19,204	100,840	3,344
50	4.033,059	137,230	29,389	35,964	109,310	3,816
55	7.989,628	292,506	27,314	70,245	110,045	4,164
60	14.959,323	517,651	28,899	131,062	**112,126**	3,950
65	26.741,648	987,017	27,093	233,739	110,911	4,223
70	45.665,963	1.783,155	25,610	397,367	106,879	**4,487**
75	76.638,723	2.949,968	25,980	663,359	111,430	4,447
80	119.131,496	4.647,913	25,631	1.042,568	106,187	4,458
85	188.712,574	7.172,943	26,309	1.782,565	100,100	4,024
90	286.887,185	10.802,614	26,557	2.681,164	101,978	4,029
95	407.860,326	15.941,909	25,584	3.955,415	98,304	4,030
100	614.516,857	22.913,464	26,819	5.696,968	100,074	4,022

Tabela 6.2: *Tempos de execução e speedups para o face-edge.*

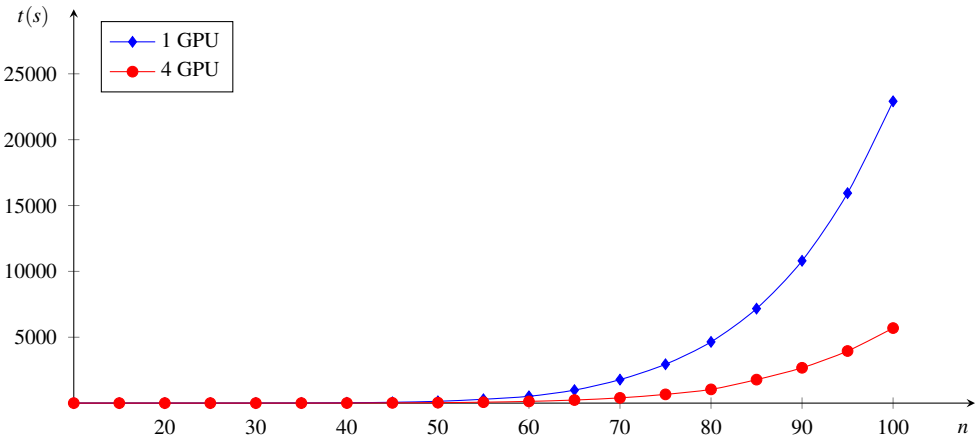


Figura 6.3: *Comparativo de tempo entre as implementações paralelas do face-edge dimpling.*

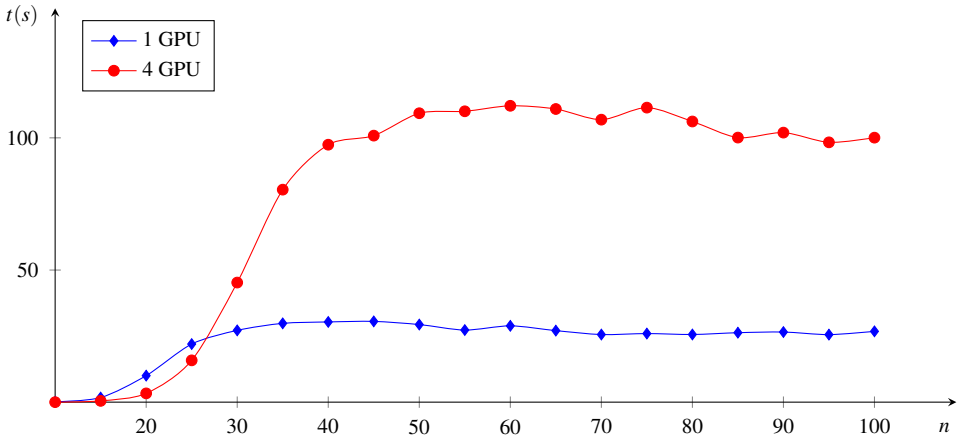


Figura 6.4: *Speedup das implementações paralelas do face-edge dimpling em relação a versão sequencial.*

GPU é em média 4 vezes mais rápida, identificando um *speedup* superlinear de até 4.4 vezes em relação a uma única GPU, ou seja, o *speedup* é superior a quantidade GPUs utilizadas.

6.2.3 Qualidade das soluções

Para avaliar a qualidade das soluções apresentadas pelas Implementações 5.2, 5.5 e 5.7, foram realizados testes com instâncias de até 40 vértices que foram resolvidas à otimalidade a partir da implementação descrita na Seção 5.1. Os resultados obtidos são descritos na Tabela 6.3. No geral, o *face dimpling* apresenta seu pior desempenho para a instância de 20 vértices, com distância de otimalidade de 4,423% diante a solução exata. Em alguns cenários, o *edge dimpling* conseguiu resultados melhores que o *face dimpling*.

No entanto, a qualidade da solução foi melhorada quando executada com o *face-edge dimpling*, obtendo seu pior desempenho para a instância de 40 vértices, com

uma distância de otimalidade de 2,054%. Em todos os casos apresentados, o *face-edge dimpling* obteve uma solução com qualidade no mínimo igual ao *face* e *edge dimpling*, além de obter a solução ótima para as instâncias com 10 e 15 vértices. Embora este, no geral, produza soluções de melhor qualidade, o *face-edege dimpling* requer uma quantidade de tempo significativamente maior quando comparado ao *face dimpling* para instâncias médias a grandes. Por isso, mesmo que as soluções produzidas sejam melhores, o *face dimpling* é o mais indicado para soluções com prioridade de tempo.

	Solução	**Face**		**Edge**		**Face-edge**	
n	**Ótima**	**Solução**	**Dist. (%)**	**Solução**	**Dist. (%)**	**Solução**	**Dist. (%)**
10	2.107	2.107	0,000	2.007	**4,746**	2.017	0,000
15	5.772	5.743	0,502	5.708	1,109	5.772	0,000
20	8.434	8.061	**4,423**	8.320	1,351	8.321	1,340
25	11.594	11.291	2,613	11.310	2,449	11.431	1,406
30	13.770	13.435	2,433	13.238	3,863	13.644	0,915
35	17.177	16.833	2,003	16.628	3,196	17.028	0,867
40	19.862	19.232	3,172	19.270	2,980	19.454	**2,054**

Tabela 6.3: *Distâncias de otimalidade para o face, edge e face-edge dimpling.*

6.3 Instâncias da literatura

Foram conduzidos testes com instâncias da literatura, incluindo a instância fornecida por [1], que resolvemos até à otimalidade através do algoritmo proposto e pelo método exato de [1] que foi implementado. A outra instância analisada foi a de 100 vértices, fornecida por [46], baseada em um estudo empírico sobre dados financeiros derivados da bolsa de valores de mercado dos Estados Unidos. Esta instância foi resolvida pelos algoritmos mencionados neste trabalho, como mostra a Tabela 6.4. Podemos ver que a solução produzida pelo PMFG [46] apresenta o menor peso dos quatro métodos.

Método	**Valor da solução**
PMFG	115,568
TMFG	116,224
Face dimpling	116,553
Face-edge dimpling	116,897
Upper-Bound	121,835

Tabela 6.4: *Valores obtidos para a instância de [46].*

Embora o TMFG [33] tenha produzido uma melhoria substancial, tanto o *face dimpling* quanto o *face-edge dimpling* foram capazes de obter soluções melhores (de

Instância	Upper Bound	Face dimpling		TMFG	
		Solução	Dist. (%)	Solução	Dist. (%)
$\alpha 0,5 - \beta 3,0$	462,959	373,880	19,241	367,250	20,673
$\alpha 3,0 - \beta 0,5$	1.185,648	1.181,402	0,358	1.179,143	0,549
Pareto α no. 1	3,136	3,135	0,040	3,134	0,064
Pareto α no. 2	14,306	13,642	4,642	13,596	4,963
Corr with 20 factors	253,806	222,784	12,223	216,951	14,521
Corr with 50 factors	100,132	83,692	16,418	81,827	18,281
Corr with 100 factors	48,429	40,738	15,881	40,022	17,359
Uniform $[0,1]$	1.082,540	1.013,646	6,364	996,526	7,946

Tabela 6.5: *Comparativo entre o TMFG e face dimpling.*

maior peso). Um limite superior de 121,835, obtido através da solução ótima, é apresentado na útlima linha. Este valor foi descoberto através de uma execução interrompida (com três meses de duração) da solução exata, utilizando as técnicas propostas por [1].

O *face dimpling* foi aplicado também para instâncias maiores da literatura [33], comparando os resultados obtidos com os do TMFG, como mostra a Tabela 6.5. (Os resultados para *edge* e *face-edge dimpling* não foram apresentados devido à grande quantidade de tempo necessária para calcular as soluções). Como não foi possível obter a solução ótima para estas instâncias, foram calculados os limites superiores para cada um através da técnicas propostas por [1]. Como pode ser visto na tabela, desconsiderando a instância *Pareto* α *no. 1*, onde ambos métodos produziram soluções muito próximas à otimalidade, o *face dimpling* obteve melhores resultados em todas as execuções.

Assim como foi mencionado anteriormente, a proposta apresentada neste trabalho garante um resultado melhor ou igual se comparado ao método de [33]. Isso pode ser observado pela instância Pareto α no. 1, onde ambas obtiveram valores extremamente próximos. Mesmo com os valores obtidos por *Pareto* α *no. 2* e *Corr with 100 factors* sendo pouco significativos, pode-se afirmar que o algoritmo proposto é, em geral, melhor.

CAPÍTULO 7

Conclusão

Este trabalho propôs estratégias de extração de um subgrafo planar de peso máximo a partir de um grafo completo com o objetivo de analisar seu uso em aplicações já existentes na literatura. Foram propostos novos algoritmos sequenciais e paralelos para o problema em questão, além de experimentos relacionados a suas implementações, contextualizando exemplos de cenários da vida real.

Além das implementações sequenciais e paralelas, foram apresentadas otimizações importantes que contribuíram para uma solução escalável em uma plataforma multi-GPU. As propostas apresentadas consideram várias sementes (tetraedros) como solução inicial, permitindo assim que o espaço de busca, considerando a heurística utilizada, seja totalmente analisado.

A implementação de uma versão exata, tendo como base a proposta de [1], desenvolveu um papel importante quanto à análise qualitativa dos experimentos realizados. Instâncias de até 40 vértices foram resolvidas em um tempo computacional razoável, apresentando um avanço significativo se comparado à instância de 24 vértices anteriormente resolvida por [1].

A parte crucial dessa implementação foi a implementação de uma função capaz de gerar os cortes que alimentam a restrição 2-9 do modelo. Além disso, foi adicionado nesta função um nível de paralelismo em CPU, uma vez que a medida que o problema crescia, a geração de cortes tornava-se o gargalo da solução. Graças a esses avanços, foi possível comparar e estimar a proximidade das soluções propostas frente a uma solução exata.

Testes realizados com grafos fornecidos por [1, 33, 46] demonstraram-se promissores. A solução obtida por este trabalho foi no mínimo idêntica às propostas citadas, apresentando, no geral, uma solução de melhor qualidade. Além disso, o uso de paralelismo permitiu que instâncias maiores fossem processadas também em um tempo razoável.

Além disso, vale ressaltar que a implementação paralela do algoritmo proposto possui a mesma qualidade da implementação sequencial, com a vantagem de ser 200 vezes mais rápida para instâncias de até 100 vértices, levando apenas 25 segundos para concluir

a execução. A versão multi-GPU demonstrou-se altamente escalável, apresentando maior eficiência em problemas a partir de 65 vértices e uma curva de *speedup* em constante crescimento.

7.1 Trabalhos futuros

Um possível projeto futuro seria comparar a proposta deste trabalho com outros trabalhos mencionados na Seção 2.1. Além disso, visto que a proposta baseia-se em um implementação embaraçosamente paralela, outro possível trabalho seria estender o atual algoritmo, aplicando paralelismo em um segundo nível, ou seja, no processamento de cada semente.

Referências Bibliográficas

[1] AHMADI-JAVID, A.; ARDESTANI-JAAFARI, A.; FOULDS, L. R.; HOJABRI, H.; FARAHANI, R. Z. **An algorithm and upper bounds for the weighted maximal planar graph problem**. *Journal of the Operational Research Society*, 66(8):1399–1412, 2015.

[2] ANDRADE, J. S.; HERRMANN, H. J.; ANDRADE, R. F. S.; DA SILVA, L. R. **Apollonian Networks: Simultaneously Scale-Free, Small World, Euclidean, Space Filling, and with Matching Graphs**. *Physical Review Letters*, 94:018702–, Jan 2005.

[3] ASTE, T.; DI MATTEO, T.; HYDE, S. **Complex networks on hyperbolic surfaces**. *Physica A: Statistical Mechanics and its Applications*, 346(1):20–26, 2005.

[4] BONDY, J.-A.; MURTY, U. S. R. **Graph Theory**, volume 244 de **Graduate Texts in Mathematics**. Springer-Verlag, London, 2010.

[5] BOSWELL, S. G. **TESSA – A new greedy heuristic for facilities layout planning**. *International Journal of Production Research*, 30(8):1957–1968, 1992.

[6] BOYER, J. M.; MYRVOLD, W. **Simplified $O(n)$ planarity algorithms**, 2001.

[7] COELHO, V. S.; MARTINS, W. S.; FOULDS, L. R.; DIAS, E. S.; CASTONGUAY, D.; LONGO, H. J. **Uma proposta de solução aproximada para o problema do subgrafo planar de peso máximo**. In: *XVII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2016)*, p. 16–27, 2016.

[8] COMMITTEE, I. S.; OTHERS. **754-2008 IEEE standard for floating-point arithmetic**. *IEEE Computer Society Std*, 2008, 2008.

[9] COOK, S. **CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of Gpu Computing)**. Morgan Kaufmann, 2012.

[10] DI BATTISTA, G.; TAMASSIA, R. **Incremental planarity testing**. *30th Annual Symposium on Foundations of Computer Science*, (v):436–441, 1989.

[11] DYER, M. E.; FOULDS, L. R.; FRIEZE, A. M. **Analysis of heuristics for finding a maximum weight planar subgraph**. *European Journal of Operational Research*, 20(1):102–114, 1985.

[12] EADES, P.; FOULDS, L.; GIFFIN, J. **An efficient heuristic for identifying a maximum weight planar subgraph**. In: Billington, E. J.; Oates-Williams, S.; Street, A. P., editors, *Combinatorial Mathematics IX: Proceedings of the Ninth Australian Conference on Combinatorial Mathematics Held at the University of Queensland, Brisbane, Australia, August 24–28, 1981*, volume 952 de **Lecture Notes in Mathematics**, p. 239–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[13] EASLEY, D.; KLEINBERG, J. **Networks, Crowds, and Markets: Reasoning About a Highly Connected World**. Cambridge University Press, Cambridge, UK, 2010.

[14] ELSHAFEI, A. N. **Hospital layout as a quadratic assignment problem**. *Operational Research Quarterly*, p. 167–179, 1977.

[15] FIEDOR, P. **Networks in financial markets based on the mutual information rate**. *Physical Review E*, 89(5):1–7, 2014.

[16] FOULDS, L. R.; GIBBONS, P. B.; GIFFIN, J. W. **Facilities layout adjacency determination: An experimental comparison of three graph theoretic heuristics**. *Operations Research*, 33(5):1091–1106, 1985.

[17] FOULDS, L. R.; ROBINSON, D. F. **A strategy for solving the plant layout problem**. *Journal of the Operational Research Society*, 27(4):845–855, 1976.

[18] FOULDS, L. R.; ROBINSON, D. F. **Graph theoretic heuristics for the plant layout problem**. *International Journal of Production Research*, 16(1):27–37, 1978.

[19] FOULDS, L. R.; ROBINSON, D. F. **Construction properties of combinatorial deltahedra**. *Discrete Applied Mathematics*, 1(1):75–87, 1979.

[20] FOULDS, L. R. **Graph Theory Applications**. Universitext. Springer, New York, 1992.

[21] GIFFIN, J. W. **Graph theoretic techniques for facilities layout**. PhD thesis, University of Canterbury, Christchurch, New Zealand, 1984.

[22] GREEN, R. H.; AL-HAKIM, L. A. R. **A heuristic for facilities layout planning**. *Omega*, 13(5):469–474, 1985.

[23] HERAGU, S. S.; KUSIAK, A. **Efficient models for the facility layout problem**. *European Journal of Operational Research*, 53(1):1–13, 1991.

[24] HOPCROFT, J.; TARJAN, R. **Efficient planarity testing**. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[25] **IBM ILOG CPLEX Optimizer**. https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer.

[26] KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: A hands-on approach**. Morgan Kaufmann, San Francisco, 2016.

[27] KNUTH, D. E. **The Art of Computer Programming, Volume 4: Generating all Combinations and Partitions, Fascicle 3**. Addison-Wesley, July, 2005.

[28] KOOPMANS, T. C.; BECKMANN, M. **Assignment problems and the location of economic activities**. *Econometrica: Journal of the Econometric ...*, 25(1):53–76, 1957.

[29] KUSIAK, A.; HERAGU, S. S. **The facility layout problem**. *European Journal of Operational Research*, 29(3):229–251, 1987.

[30] LENGAUER, T. **Combinatorial algorithms for integrated circuit layout**. Springer Science & Business Media, 2012.

[31] LEUNG, J. **A new graph-theoretic heuristic for facility layout**. *Management Science*, 38(4):594–605, 1992.

[32] LIEBERS, A. **Planarizing graphs – A survey and annotated bibliography**. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001.

[33] MASSARA, G. P.; DI MATTEO, T.; ASTE, T. **Network filtering for big data: Triangulated maximally filtered graph**. *Journal of Complex Networks*, 5(2):1–18, 2017.

[34] MELLER, R. D.; GAU, K.-Y. **The facility layout problem: Recent and emerging trends and perspectives**. *Journal of Manufacturing Systems*, 15(5):351–366, 1996.

[35] MOORE, J. **Plant Layout and Design**. MacMillan, 1969.

[36] MUSMECI, N.; ASTE, T.; DI MATTEO, T. **Relation between financial market structure and the real economy: Comparison between clustering methods**. *PloS one*, 10(3):1–24, 2015.

[37] MUSMECI, N.; ASTE, T.; DI MATTEO, T. **Risk diversification: A study of persistence with a filtered correlation-network approach**. *The Journal of Network Theory in Finance*, 1:77–98, 2015.

[38] NAGAMOCHI, H.; SUZUKI, T.; ISHII, T. **A simple recognition of maximal planar graphs**. *Information Processing Letters*, 89(5):223–226, 2004.

[39] OSMAN, I. H. **A tabu search procedure based on a random roulette diversification for the weighted maximal planar graph problem**. *Computers & Operations Research*, 33(9):2526–2546, 2006.

[40] PESCH, E. **Efficient facility layout planning in a maximally planar graph model**. *International Journal of Production Research*, 37(2):263–283, 1999.

[41] POZZI, F.; DI MATTEO, T.; ASTE, T. **Spread of risk across financial markets: better to invest in the peripheries**. *Scientific Reports*, 3(1665):1–7, 2013.

[42] SEPPÄNEN, J.; MOORE, J. M. **Facilities planning with graph theory**. *Management Science*, 17(4):B–242–B–253, 1970.

[43] SEPPÄNEN, J.; MOORE, J. M. **String processing algorithms for plant layout problems**. *International Journal of Production Research*, 13(3):239–254, 1975.

[44] SOLIHIN, Y. **Fundamentals of Parallel Multicore Architecture**. CRC Press, 2015.

[45] SONG, W.-M.; ASTE, T.; DI MATTEO, T. **Correlation-based biological networks**. In: *Proceedings of International Symposium on Microelectronics, MEMS and Nanotechnology*, volume 6802, p. 680212–1–680212–11, Canberra, Australia, 2007.

[46] TUMMINELLO, M.; ASTE, T.; DI MATTEO, T.; MANTEGNA, R. N. **A tool for filtering information in complex systems**. *Proceedings of the National Academy of Sciences of the United States of America*, 102(30):10421–10426, 2005.

[47] WILT, N. **The CUDA Handbook: A Comprehensive Guide to GPU Programming**. Addison-Wesley Professional, 2013.

APÊNDICE A

Lista de siglas

ALU - Arithmetic Logic Units
API - Application Programming Interface
ARB - Architecture Review Board
CPU - Central Processing Units
CUDA - Compute Unified Device Architecture
FMA - Fused Multiply-Add
FPU - Floating Point Units
GPU - Graphic Processing Units
GRASP - Greedy Random Adaptive Search
IP - Integer Programming
MAD - Multiply-Add
MESI - Modified, Exclusive, Shared, Invalid
MPG - Maximal Planar Subgraph
MSI - Modified, Shared, Invalid
MWPSP - Maximum Weight Planar Subgraph Problem
OpenMP - Open Multi-Processing
PCB - Printed Circuit Boards
PLI - Programação Linear Inteira
PMFG - Planar Maximally Filtered Graph
QAP - Quadratic Assignment Problem
SFU - Special Function Units
SIMT - Single-Instruction Multiple-Threads
SM - Streaming Multiprocessors
SMP - Symmetric Multiprocessor
SMT - Simultaneous Multi-Threading
SP - Streaming Processors
SPMD - Single-Program Multiple-Data
STL - Standard Template Library
TMFG - Triangular Maximally Filtered Graph