CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Gabriel Felipe Paiva Pereira

ESTUDO DA APLICAÇÃO DE MODELOS DE COMUNICAÇÃO ENTRE GPUS NA SOLUÇÃO DE PROBLEMAS PELO MÉTODO DOS GRADIENTES CONJUGADOS

Timóteo

Dezembro/2019

Gabriel Felipe Paiva Pereira

ESTUDO DA APLICAÇÃO DE MODELOS DE COMUNICAÇÃO ENTRE GPUS NA SOLUÇÃO DE PROBLEMAS PELO MÉTODO DOS GRADIENTES CONJUGADOS

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Lucas Pantuza Amorim

Timóteo

Dezembro/2019

Gabriel Felipe Paiva Pereira

Estudo da Aplicação de Modelos de Comunicação entre GPUs na solução de Problemas pelo Método dos Gradientes Conjugados

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

Trabalho aprovado. Timóteo, 12 de dezembro de 2019:

Prof. Me. Lucas Pantuza Amorim
Orientador

Prof. Dr. Rodrigo Gaiba de Oliveira
Professor Convidado

Prof. Dr. Bruno Rodrigues Silva
Professor Convidado

Timóteo
2019

Dedico a todos que acreditaram em mim
e contribuíram para minha educação.

Agradecimentos

Agradeço à Deus e meus familiares pelo apoio incondicional nestes anos. Em especial à minha mãe Walquíria, meu pai José Mário e minha irmã Rebecca.

Agradeço à Victória, por sua paciência e presença nos momentos de dificuldade.

Agradeço ao Thiago Goveia e ao professor Lucas Pantuza, peças muito importantes na realização deste trabalho.

Agradeço a todos mestres e amigos que fiz nesta jornada. Eles deram sentido ao caminho.

"Eu sou a mensagem"
Markus Zusak

Resumo

Diversos fenômenos de ciência e engenharia são modelados a partir de equações diferenciais parciais. Essas últimas podem ser resolvidas, resguardada a precisão, a partir de técnicas numéricas de aproximação como o Método dos Elementos Finitos, que tem como produto um sistema linear. Para alcançar um bom desempenho na resolução desse sistema, são utilizadas estratégias que envolvem o uso e cooperação de placas gráficas. A comunicação de dados entre dispositivos instalados em um mesmo *host* é um fator relevante à esse ambiente. A fim de comparar os modelos de comunicação entre GPUs definidos por Kraus e Stephan (2017), a saber, *Multi-Threaded Copy* e *Multi-Threaded P2P*, é realizado neste trabalho uma implementação do método dos elementos finitos elemento a elemento integrado ao Gradiente Conjugado para resolver o sistema supracitado, utilizando a coloração da malha e estratégias de separação de tarefas entre as GPUs. Os resultados obtidos foram comparados e na etapa de mapeamento, que possui duas operações de comunicação entre GPUs, o modelo *Multi-Threaded P2P* superou o *Multi-Threaded Copy* independente da malha ou quantidade de placas.

Palavras-chave: Comunicação entre Placas Gráficas, Computação Paralela, Método dos Elementos Finitos, Método dos Gradientes Conjugados.

Abstract

Several phenomena of science and engineering are modeled from partial differential equations. The latter can be solved, safeguarding precision, using numerical approximation techniques such as the Finite Element Method, which has as its product a linear system. To achieve a good performance in the resolution of this system, strategies involving the use and cooperation of graphics cards are used. Data communication between devices installed on the same host is a relevant factor in this environment. To compare the communication models between GPUs defined by Kraus e Stephan (2017), namely *Multi-Threaded Copy* and *Multi-Threaded P2P*, an implementation of the Element-to-Element Finite Element method integrated with the Conjugate Gradient is performed, using mesh coloration and task separation strategies between GPUs. The obtained results were compared and in the mapping stage, which has two communication operations between GPUs, the *Multi-Threaded P2P* model surpassed the *Multi-Threaded Copy* regardless of the mesh or number of devices.

Keywords: Communication Between GPUs, Parallel Computing, Finite Element Method, Conjugate Gradient Method.

Lista de ilustrações

Lista de abreviaturas e siglas

FEM	Método dos Elementos Finitos
EBE	Elemento por Elemento
EBE-FEM	Método dos Elementos Finitos de Elemento por Elemento
GPGPU	Unidade de Processamento Gráfico de Propósito Geral
GPU	Unidade de Processamento Gráfico
CG	Gradiente Conjugado
CPU	Unidade central de processamento
MT-Copy	Multi-Threaded Copy
MT-P2P	Multi-Threaded P2P
P2P	Peer to Peer
PD	Positivo-Definida
FEA	Análise dos Elementos Finitos
PVI	Problema de Valor Inicial
PVC	Problema de Valor de Contorno

Sumário

1 Introdução

"The number one reason most people don't get what they want is that they don't know what they want"
T Hard Eker

Diversos problemas de engenharia como a dilatação ou contração de cabos de rede elétrica (MIOTTO; CARGNELUTTI; MACHADO, 2013) e formulação de circuitos elétricos com resistências, indutores e capacitores (SILVA, 2014), podem ser escritos como equações diferenciais (PEREIRA, 2005). A fim de obter uma resposta às equações diferenciais utilizando ferramentas computacionais atuais, surgem soluções baseadas em aproximação numérica do resultado da equação diferencial. Dentre essas, há o Método dos Elementos Finitos - FEM (GALVIS; VERSIEUX, 2011). Esse método gera uma equação matricial da forma:

$$Ax = b \tag{1.1}$$

na qual A é uma matriz geralmente esparsa, isso é, com uma grande quantidade de zeros, b é um vetor resposta e x é uma possível solução. A partir desse método, surgiram algumas variações no intuito de maximizar desempenho, dentre essas, o método dos elementos finitos de elemento por elemento - EbE-FEM, idealizado inicialmente para processadores com baixa quantidade de memória (CAREY et al., 1988). Diferentemente do método original, que computa uma grande matriz A do sistema a ser resolvido, essa modificação trabalha com cada elemento da matriz separadamente, gerando uma equação de elemento, dada por:

$$A_e x_e = b_e \tag{1.2}$$

Entretanto, essa nova abordagem do método traz a necessidade de uma estratégia de coloração de malha para evitar a condição de corrida. Dessa forma, os elementos da malha são computados em grupos independentes que partilham uma mesma cor, sendo realizada a computação dos grupos juntamente com as iterações do *solver* até que seja alcançada a precisão desejada (AMORIM et al., 2018, p.1). Quando a perspectiva passa aos elementos, deve-se assegurar que os cálculos simultâneos são independentes uns dos outros. Portanto, a separação de elementos em cores, isto é, grupos de execução, garante a solução para a condição de corrida.

No contexto de *solvers* para o sistema matricial, existem aqueles iterativos, que partem de um valor inicial para o vetor x e atualizam esse valor até um limite de erro tolerável. Tais sistemas são uma estratégia para aproximar o valor de resposta sem a necessidade calculos mais complexos sobre a matriz A (como faria um *solver* direto). Além disso, são beneficiados pela possibilidade de se decompor em operações mais simples, como em produtos de matriz por vetor e produtos internos de vetores, tornando viável a implementação em uma metodologia de elemento por elemento (KISS et al., 2012, p.1).

Dentre esses *solvers* iterativos, um bastante utilizado principalmente junto ao FEM é o Gradiente Conjugado (CG), pois resultados revelam que as aproximações de elementos finitos geram sistemas lineares bem condicionados que possibilitam a rápida convergência do CG. (SCHWARZ, 1979, p.343).

Essas ferramentas, atreladas ao surgimento e evolução das Unidades de Processamento Gráfico de Propósito Geral (GPGPU), foram temas de muitos trabalhos que usufruíam desses dispositivos no intuito de melhorar o desempenho dos algoritmos, (e.g, Dziekonski et al. (2012) e Kiss et al. (2012)).Especialmente no que diz respeito ao método dos elementos finitos elemento por elemento, ele é bastante usado em trabalhos que propõem utilizar GPU para ganhos significativos de desempenho computacional (AMORIM et al., 2018, p.2).

Uma característica das Unidades de Processamento Gráfico é a facilidade de trabalhar com paralelização, pois o dispositivo possui inúmeros blocos internos e cada um deles possui inúmeras *threads* que executam a instrução dada à priori. Dessa forma, é possível gerenciar cada *thread* para se preocupar apenas com uma parte independente da atividade que será realizada. Neste contexto, em um problema facilmente decomposto em partes menores independentes, trabalhar com múltiplas placas gráficas pode trazer um grande beneficio no sentido de ampliar o número de *threads*, o que torna possível resolver problemas ainda maiores. Para este fim, muitos trabalhos foram produzidos inclusive com o *solver* CG, e.g, Zaspel e Griebel (2010) e Cevahir, Nukada e Matsuoka (2009).

Entretanto, ao considerar múltiplas GPUs, existem variáveis que podem impactar no desempenho do algoritmo como: tamanho e forma de domínio, tipo de particionamento de dados, número de GPUs, largura das bordas de troca, núcleos a serem usados e tipo de sincronização entre os contextos das GPUs (SPAMPINATO, 2009, p.iii). Além dessas, uma variável muito relevante se tratando de recursos distribuídos é a forma de comunicação entre esses dispositivos gráficos (SPAMPINATO, 2009, p.2).

Em uma das conferências de tecnologia de GPU da NVIDIA de 2017, Kraus, em seu trabalho Multi GPU Programming Models, discorre acerca de uma implementação do método das diferenças finitas com o *solver* Jacobi utilizando uma abordagem de oito placas e analisando o desempenho de diferentes modelos de comunicação, dentre eles *"multi-threaded copy"* e *"multi-threaded p2p"* (KRAUS; STEPHAN, 2017).

A fim de avaliar os impactos dos modelos de comunicação citados por Kraus, alterando-se o *solver* e utilizando o EbE-FEM, a problemática desse trabalho consiste em analisar a viabilidade e desempenho obtido nos modelos citados anteriormente em uma implementação de método dos elementos finitos de elemento por elemento em um contexto de múltiplas placas gráficas com o *solver* de Gradiente Conjugado, haja vista sua relevância em trabalhos correlatos. Para essa análise será utilizada como medida o tempo de processamento.

1.1 Justificativa

Já foi mencionado que, como uma abordagem com viés de economizar memória em detrimento à computação, a estratégia de elemento por elemento no método dos elementos

finitos pode ser aplicada. Ademais, utilizar *solvers* iterativos (em especial o Gradiente Conjugado, amplamente usado pela comunidade) com essa estratégia é plausível, pois as matrizes de elementos não são armazenadas (como os métodos tradicionais fazem com a matriz global), mas são recalculadas em cada iteração. Isso é possível pois os *solvers* iterativos não afetam a matriz do sistema durante a solução, diferentemente dos diretos (KISS et al., 2012, p.1).

Neste cenário, do método dos elementos finitos elemento por elemento, no qual depende-se de uma alta capacidade de computação, unidades gráficas de processamento podem ser utilizadas com muita vantagem (KISS et al., 2012, p.1).

A medida que aumenta o número de dispositivos de unidade gráfica, é possível resolver problemas maiores e mais rapidamente, devido à facilidade de paralelização. Entretanto surge um conjunto de variáveis que impactam o desempenho da aplicação, dentre essas, o modelo de comunicação entre as placas é um aspecto relevante (SPAMPINATO, 2009).

Torna-se importante realizar o trabalho à medida que há benefícios de se utilizar um *solver* iterativo simples (CG) atrelado ao método dos elementos finitos elemento por elemento, percebe-se a boa aplicabilidade deste último à realidade das placas gráficas e entende-se que os modelos de comunicação entre placas são impactantes neste cenário. Portanto, compreender como cada modelo de comunicação afeta o desempenho do ambiente proposto é relevante.

1.2 Objetivos

O objetivo geral deste trabalho é analisar o desempenho obtido nos modelos citados por Kraus (KRAUS; STEPHAN, 2017), i.e., *multi-threaded copy* e *multi-threaded p2p* em uma implementação de método dos elementos finitos elemento por elemento com o *solver* de Gradiente Conjugado em um contexto de múltiplas Unidades de Processamento Gráfico, utilizando para essa análise a medida de tempo.

1.3 Estrutura do trabalho

Este trabalho está estruturado da seguinte forma:

- O Capítulo 2 objetiva-se abordar os principais trabalhos correlatos a este;
- O Capítulo 3 apresenta bases teóricas com os principais conceitos para melhor entendimento do trabalho, nele são abordados: Programação em GPU, Modelos de Comunicação, Método dos Elementos Finitos e Método dos Gradientes Conjugados;
- Os procedimentos metodológicos utilizados no trabalho são apresentados no Capítulo 4, sendo composto por três etapas;
- O quinto Capítulo descreve as particularidades do problema apresentado e decisões tomadas para a solução dele, bem como os resultados alcançados;

- Por fim, no Capítulo 6 são apresentadas conclusões a partir deste trabalho.

2 Trabalhos Relacionados

"Never confuse movement with action."
Ernest Hemingway

O método dos elementos finitos, especificamente na variação elemento por elemento, é utilizado neste trabalho e foi tema de outros estudos que tinham o intuito de melhorar desempenho, precisão, economizar memória ou aumentar a capacidade de processamento paralelo, como Kiss et al. (2012) e Hu, Quigley e Chan (2008).

Outro trabalho que usa o método dos elementos finitos elemento por elemento é o estudo de Amorim et al. (2018). Este aborda uma implementação que evita a necessidade da estratégia de coloração de malha, resolvendo o problema de corrida utilizando a memória do dispositivo gráfico.

Além da variação de elemento por elemento, este trabalho utiliza o *solver* Gradiente Conjugado, uma junção que é utilizada em vários trabalhos, como pode-se citar o trabalho de Goveia (2017) que aborda a implementação do método dos Gradientes Conjugados elemento a elemento utilizando a coloração. Ademais, existem mais estudos que utilizam a variação de elemento por elemento combinado com o Gradiente Conjugado pré-condicionado, como Xu et al. (2007).

Alguns trabalhos já implementam o gradiente conjugado no ambiente CUDA, uma API empregada em aplicações com computação paralela e heterogênea da NVIDIA que será utilizada neste trabalho, caso do estudo de Grisa (2010).

Aproximando-se ainda mais do viés desse trabalho, em sua dissertação, Bueno (2013) utiliza o método do gradiente conjugado em múltiplas GPUs, no ambiente *OpenCL*, visando resolver sistemas de equações lineares esparsos.

Existem implementações que exploram a programação com múltiplas placas gráficas especialmente no que diz respeito ao desempenho das aplicações, como Spampinato (2009) que estuda o impacto de determinadas variáveis, como tamanho e forma do domínio, número de dispositivos e formas de sincronização em um ambiente com vários dispositivos.

Camargos e Silva (2015) também faz uma análise de desempenho em uma abordagem utilizando métodos iterativos de Krylov, uma família específica de métodos, dos quais incluem o Gradiente Conjugado e mínimos resíduos genéricos por exemplo, que serão aplicados na análise de elementos finitos (FEA) de um fenômeno eletromagnético. Para serem comparados, são feitas implementações com múltiplas GPUs desses *solvers* sendo inspecionadas métricas de tempo e escalabilidade.

Outro importante estudo vínculado a este é um desenvolvido por Kraus e Stephan (2017) em uma conferência de tecnologia da NVIDIA que aborda modelos de comunicação entre as placas gráficas em uma aplicação do *solver* Jacobi. Esses tipos de comunicação

serão utilizados e comparados neste trabalho presente.

3 Fundamentação teórica

“Só aposte o que valer a pena”
Max Gunther

3.1 Programação em GPU

As unidades de processamento gráfico (GPUs) se diferem da unidade central de processamento (CPU), pois foram originalmente desenvolvidas para executar computações gráficas de forma paralela. Estas são unidades secundárias de processamento capazes de ampliar o poder computacional das máquinas e a gama de aplicações (SANDERS; KANDROT, 2011).

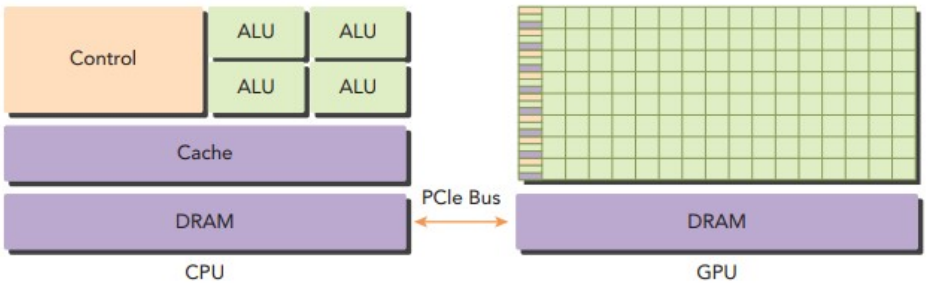
Com esses dois tipos de unidades atuando em soluções diferentes, tira-se bastante proveito de um sistema heterogêneo, com vantagens significativas comparados aos tradicionais sistemas de computação de alto desempenho (CHENG; GROSSMAN; MCKERCHER, 2014).

3.1.1 Arquitetura da Unidade de Processamento Gráfico

Placas gráficas são unidades secundárias, dessa forma não são plataformas independentes, mas utilizadas juntas à CPU. Por causa disso, a CPU geralmente é chamada de *Hospedeiro ou Host* e a GPU de *Dispositivo ou Device*.

A forma de conectar a GPU com a CPU é atraves de PCI-Express bus, um padrão de comunicação de dados, como mostra a Figura 1.

Figura 1 – Esboço da CPU e GPU conectadas pelo PCIe Bus



Fonte: Professional Cuda C Programming, 2014

Uma aplicação que utiliza tanto CPU quanto GPU pode ser chamada de heterogênea e mostra-se muito eficiente na medida que, se cada unidade tem suas vantagens para certos tipos de programas, utilizá-las da melhor forma pode otimizar a solução. A saber, CPU computa melhor as tarefas de controle intensivo e a GPU computa melhor as tarefas com intensiva paralelização de dados (CHENG; GROSSMAN; MCKERCHER, 2014, p. 12).

Como pode ser observado também na Figura 1, a GPU tem inúmeras unidades, chamadas de *threads*, que são dispostas em blocos e que podem executar computações em paralelo, por isso a facilidade da placa gráfica com tarefas de intenso paralelismo de dados.

Para tornar possível utilizar as placas de vídeo para computações de propósito geral, a NVIDIA construiu uma plataforma em que fosse possível criar algoritmos para serem executados em GPU, facilitando o que antes tinha que ser feito em OpenGL ou DirectX. Dessa forma, a linguagem, cujo nome é CUDA, tem sido utilizada em inúmeras aplicações como em dinâmica de fluidos e digitalização de imagens médicas, com otimizações exclusivas ao desempenho das placas (SANDERS; KANDROT, 2011).

3.1.2 CUDA

CUDA é uma plataforma para programação de aplicações de propósito geral que usa paralelismo, em unidades de processamento gráfico da *NVIDIA*. CUDA dispõe de muitas bibliotecas de otimização, interfaces e extensão para uso de linguagens como C, C++, Fortran e Python (CHENG; GROSSMAN; MCKERCHER, 2014).

Como apontado por Cheng, Grossman e McKercher, existe uma estrutura genérica que os programas heterogêneos em CUDA seguem (CHENG; GROSSMAN; MCKERCHER, 2014), que é:

1. Alocar a memória da GPU;
2. Copiar as informações da memória da CPU para a GPU;
3. Invocar o kernel do CUDA para realizar a computação paralela específica do programa;
4. Copiar a informação de volta da memória GPU para a CPU;
5. Liberar o espaço alocado na GPU

3.1.3 Programando em uma GPU

CUDA conta com inúmeras funções já desenvolvidas para tornar possível a comunicação da aplicação com a GPU, como por exemplo `cudaMalloc(void**devPtr, size_t size)` e `cudaMemcpy(void *dst, const void * src, size_t count, enum cudaMemcpyKind kind)` utilizadas respectivamente para alocar uma quantidade de memória em uma GPU específica e copiar a memória da unidade de origem para uma unidade de destino, seja ela GPU ou CPU.

Além dessas, CUDA conta com palavras chaves exclusivos como o caso do `___global___` que indica que uma função será executada na GPU, como exemplificado abaixo no Código 3.1

$$\_\_\_global\_\_\_\ function(int\ a,\ int\ b) \tag{3.1}$$

Ao utilizar `___global___`, a função será chamada como demonstrado abaixo no Código 3.2, especificando-se o número de blocos e o número de *threads* por bloco que executarão aquela função.

$$function <<< numBlocks, threadsPerBlock >>> (a, b); \tag{3.2}$$

Quando uma função é executada na placa de vídeo, todas as *threads* executam o mesmo trecho de código. Por isso, para que não seja feita computação repetida, isto é, a mesma coisa inúmeras vezes, o código a ser paralelizado deve conter tratativas que consideram qual *thread* está executando-a. Dessa forma, é possível fazer com que cada *thread* preocupe-se com uma parte pequena do problema de forma que, quando sincronizadas, representem a computação completa (SANDERS; KANDROT, 2011).

Como exemplo, o Código 3.1 demonstra a função *add* que adiciona dois vetores *a* e *b* e os coloca em *c*, todos de tamanho *N*.

```
1 ___global___ void add(int *a,int *b,int *c){
2     int index = threadIdx.x + blockIdx.x * blockDim.x;
3     while (index < N) {
4         c[index] = a[index] + b[index];
5         index += blockDim.x * gridDim.x;
6     }
7 }
```

Código fonte 3.1 – Código da função add em GPU

É interessante pensar que todas as *threads* da GPU estarão executando ao mesmo tempo essa função, por isso, a variável `index` na linha 2 representa qual seria o índice global daquela *thread* no contexto completo da placa gráfica.

A Figura 2 demonstra como se dá o arranjo de blocos e *threads* em duas dimensões.

Figura 2 – Arranjo de threads e blocos em duas dimensões

Fonte: Cuda By Example, 2011.

Ao comparar a imagem com o código, a variável `threadIdx.x` representa o índice da *thread* dentro do bloco, `blockIdx.x` o índice do bloco em que a *thread* está e `blockDim` um valor constante para todos blocos que representa o número de *threads* existentes na extensão de cada dimensão do bloco. Neste caso, o bloco é unidimensional, por isso é utilizado

`blockDim.x`, mas também é possível trabalhar com a dimensão `y` e `z` (SANDERS; KANDROT, 2011).

Para finalizar a computação, o resultado de todas as *threads* é relevante, portanto CUDA também conta com funções suporte, em especial `__syncthreads()`, que funciona como uma espécie de barreira em hardware para as *threads* de um dispositivo, que apenas prossegue a computação após todas chegarem a essa marca.

3.1.4 Programando em múltiplas GPUs

Aplicações heterogêneas podem dispor de mais de uma placa gráfica. Por causa disso, os programadores devem escolher a forma como será feita a divisão de tarefas de maneira a realizar o que se deseja.

A programação em múltiplas GPUs no ambiente CUDA herda todos os passos definidos para uma aplicação com apenas uma GPU. Entretanto, todos os passos referentes ao dispositivo, passam a ser feitos para cada GPU. Por isso, deve-se alocar memória para os dispositivos e invocar a função desejada para cada um deles (SANDERS; KANDROT, 2011).

CUDA também já possui funções próprias para lidar com GPUs, podem ser citadas `cudaGetDeviceCount(int* count)` e `cudaSetDevice(int device)` que retorna o número de dispositivos conectados na máquina e escolhe um dispositivo por índice respectivamente.

Além do CUDA já trabalhar com funções de suporte, existem bibliotecas que facilitam a paralelização entre as placas gráficas. No trabalho de Kraus e Stephan (2017), o autor explora o OpenMP, uma interface de programação de aplicações (API) focada em processamento paralelo em CPUs utilizando memória compartilhada.

Essa API é usada pois cada placa gráfica deve executar em uma *thread* diferente da CPU, de forma que consigam computar paralelamente.

3.1.4.1 OpenMP

O OpenMP surge da necessidade de desenvolver aplicações que usam processamento paralelo por meio de compartilhamento de memória, sendo nesses cenários um padrão bastante utilizado (SENA; COSTA, 2008).

OpenMP funciona como uma notação que pode ser adicionada em programas nas linguagens C, C++ e Fortran, que transforma aquele código sequencial, sendo possível descrever como o processamento será dividido e executado entre as *threads*, organizando o acesso à memória compartilhada (CHAPMAN; JOST; PAS, 2008).

No âmbito das placas gráficas, o *host* (CPU) deve fazer a chamada para a execução da função no dispositivo gráfico. Ao tratar de múltiplos dispositivos, é interessante que existam tantos *hosts* quanto dispositivos para que seja feita a chamada dos *kernels* de forma paralela, nisso surge a necessidade de trabalhar com *threads* na CPU, o que leva à escolha do OpenMP.

Em suma é criada uma *thread* no CPU para cada placa gráfica e em cada *thread* é atribuído o dispositivo relativo (com a função `cudaSetDevice`) a fim de ser executada a função

de interesse naquele dispositivo.

Com o intuito de trabalhar dessa maneira, Kraus e Stephan (2017) utiliza OpenMP para criar as *threads* no CPU e por fim executar tudo que se refere a um dispositivo gráfico, compartilhando entre as *threads* uma sequência de dados relevante à execução da função.

Como exemplo, a notação no Algoritmo 3.3 deve ser utilizada antes de um código que será paralelizado, dessa forma são criadas tantas *threads* quanto o número de dispositivos, sendo que estas possuem compartilhamento entre as posições de memória ocupadas pela variável `result` no exemplo.

```
#pragma omp parallel num_threads(num_devices) shared(result)
```
(3.3)

Além do OpenMP, outra API que pode ser explorada neste cenário de GPUs é a cuBLAS, utilizada pelo trabalho de Goveia (2017). Ela oferece suporte principalmente a operações algébricas em placas gráficas.

3.1.4.2 cuBLAS

A API cuBLAS é uma biblioteca do CUDA utilizada para computações em placas gráficas que possui a implementação de subprogramas de álgebra linear básica (BLAS). Seu uso torna-se interessante pois os métodos implementados são computações recorrentes e já estão padronizados, retornando até *feedback* de *status*.

Outra vantagem na aplicação dessas funções é que, como o cuBLAS faz parte do *kit* de ferramentas (*Toolkit*) do CUDA, possui o código compatível e facilmente integralizado com a plataforma, com as possíveis otimizações implementadas para usufruir das placas gráficas (NVIDIA, 2012).

Uma funcionalidade que pode ser citada é a `cublasSdot`, usada também em trabalhos correlatos como o de Goveia (2017), essa função realiza o produto escalar entre dois vetores *x* e *y* efetuando as computações na placa gráfica. Graças a API, essa operação não exige do programador a preocupação com a forma de paralelizar as tarefas entre as *threads* da GPU, uma vez que já está implementada a funcionalidade pela biblioteca.

Pode ser visto no Algoritmo 3.4, a forma como pode-se chamar essa função em C:

```
result = cublasSdot(handle, n, x, incx, y, incy)
```
(3.4)

em que são passados, respectivamente: o contexto do cuBLAS (criado a partir da função `cublasCreate()`, útil para estabelecer controle em um cenário de múltiplas *threads* no *host* e múltiplos dispositivos), o número de elementos nos vetores de entrada, o ponteiro para o vetor *x*, distância de memória entre os elementos de *x*, o ponteiro para o vetor *y* e a distância de memória entre os elementos de *y*. Por fim, essa função retorna um vetor que é o produto escalar entre *x* e *y*.

3.2 Modelos de Comunicação

3.2.1 GPUDirect

Um conjunto de soluções interessantes ao contexto de múltiplas placas gráficas é a *NVIDIA GPUDirect*. Ao trabalhar com comunicação entre as GPUs, os dados passam do dispositivo de origem, são copiados a CPU que posteriormente envia à GPU de destino. Isso gera cópias de memória do dispositivo de origem ao host e do host ao dispositivo de destino como demonstrado por Kraus em seu trabalho (KRAUS; STEPHAN, 2017).

Além do espaço de memória que deve ser reservado para alocar essa operação, há também o *overhead* que a CPU gasta realizando as cópias e transmissões entre os dispositivos. Por causa disso, foi desenvolvida pela *NVIDIA* a tecnologia GPUDirect com foco em otimizar o movimento dos dados entre GPUs ou entre GPUs e outros dispositivos.

GPUDirect já conta com algumas distribuições, como por exemplo a *"GPUDirect Peer-to-Peer"*, relevante ao trabalho.

3.2.2 GPUDirect Peer-to-Peer (P2P)

GPUDirect *Peer-to-Peer* foi implementada em 2011 e foi utilizada também no trabalho de Kraus e Stephan (2017) para otimizar o tempo de comunicação entre as GPUs.

Ao existir troca de informações entre duas placas gráficas, o CPU deve receber da placa origem e disponibilizar para a placa destino. Entretanto, essa distribuição possibilita que duas GPUs consigam se comunicar apenas pelo *PCI Express*, evitando que a CPU realize essas cópias desnecessárias (AGOSTINI; ROSSETTI; POTLURI, 2018).

Como essa solução é nativa das placas gráficas, basta apenas habilitar a comunicação deste tipo a nível de código para que seja possível realizar as computações.

Em 3.2 é demonstrado um exemplo de código responsável por habilitar o acesso à memória *Peer-to-Peer*.

```
1 cudaSetDevice(dev_id_1);
2 int canAcess = 0;
3 cudaDeviceCanAcessPeer(&canAcess,dev_id_1,dev_id_2);
4 if(canAcess)
5     cudaDeviceEnablePeerAcess(dev_id_2,0);
```

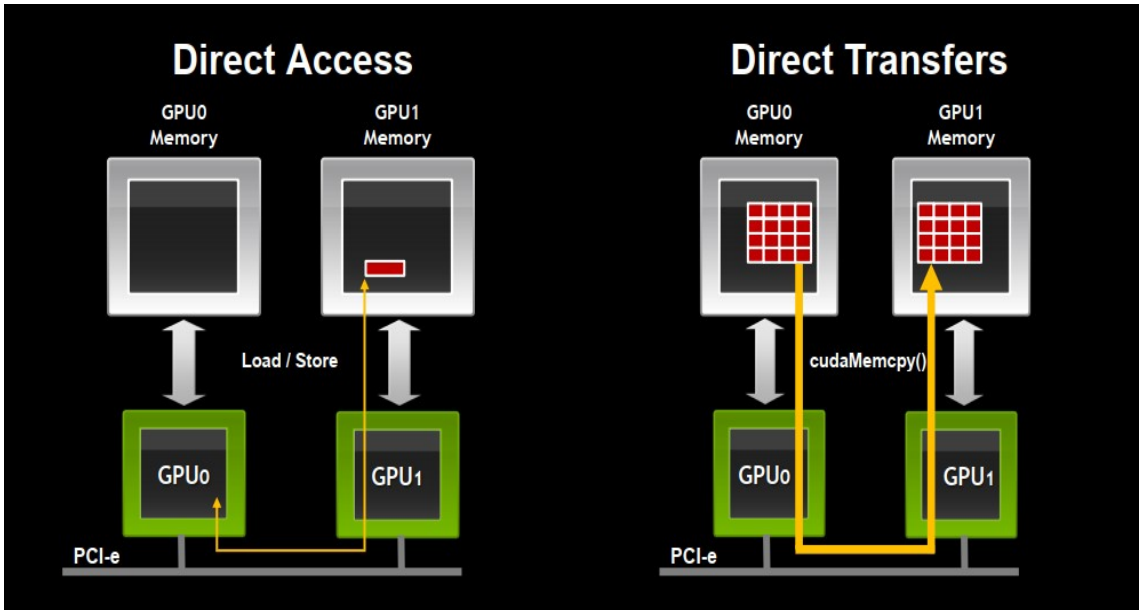
Código fonte 3.2 – Exemplo de código para habilitar acesso *Peer-to-Peer*

A função da linha 3 verifica se as placas gráficas podem implementar esse tipo de comunicação, isto é, se compartilham o mesmo PCI Express, enquanto que na linha 5 o dispositivo que está agora em uso (o `dev_id_1` ativado pelo `cudaSetDevice` da linha um), é habilitado a se comunicar com o `dev_id_2`.

A Figura 3 representa como se dá a conexão entre os dispositivos. São demonstradas operações diretas de acesso (esquerda) e transferência de arquivos (direita) entre as duas

placas gráficas, sendo que a transferência é feita utilizando o `cudaMemCpy` após a habilitação do *Peer to Peer*.

Figura 3 – Arquitetura com GPUDirect Peer-to-Peer



Fonte: NVIDIA Corporation, 2012

É importante frizar que essa comunicação só é possível entre placas gráficas que compartilhem o mesmo PCI Express. Caso estejam relacionadas de outra forma, não se garante a conexão direta.

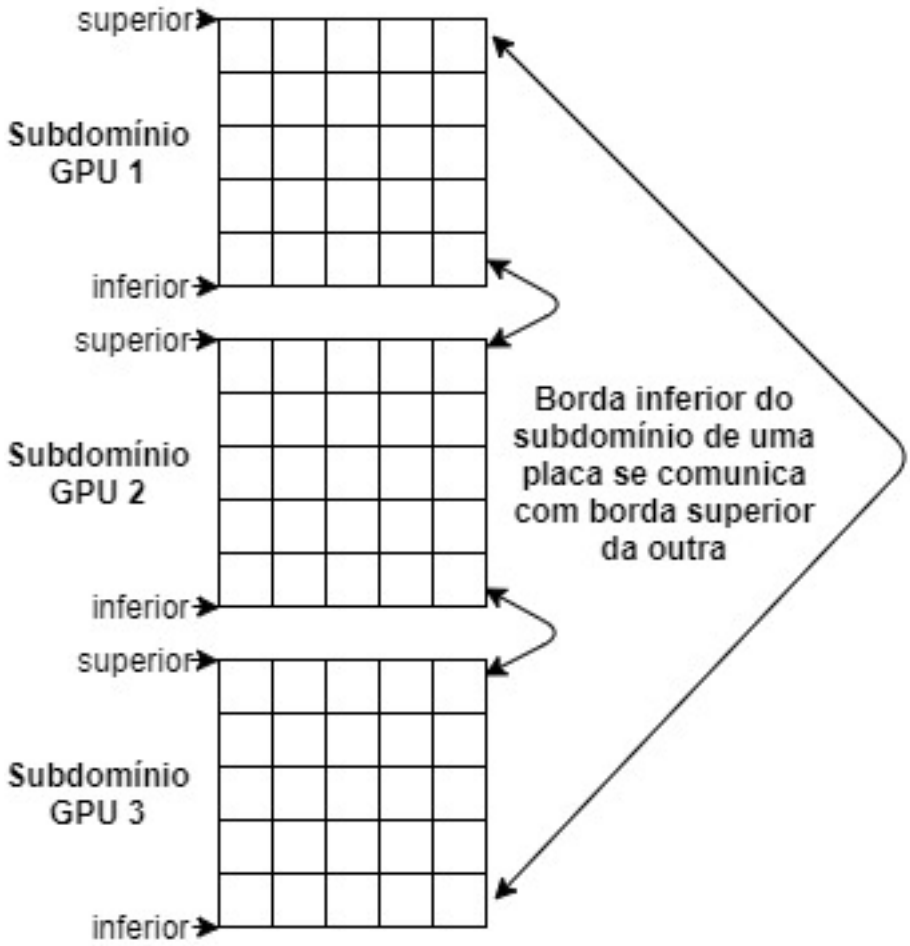
3.2.3 Modelos Abordados

Kraus e Stephan (2017) analisa de forma comparativa o uso de diferentes modelos de comunicação entre múltiplas placas gráficas na resolução do *solver* Jacobi levando em conta métricas de tempo e eficiência.

Já este trabalho, utiliza um *solver* diferente do abordado nesse trabalho. Outra diferença é que esse autor não adota a técnica do método dos elementos finitos elemento por elemento, portanto será feita uma adaptação dos modelos definidos por ele para este trabalho presente.

Na forma como Kraus define a divisão do domínio, cada GPU trabalha com uma parte das computações e se comunica com outras duas placas, sendo que uma delas compartilha a borda superior do sub-domínio com a borda inferior da outra, enquanto que a borda inferior da última placa é compartilhada com a borda superior da primeira placa de forma que todas as placas juntas representem o domínio todo. Como pode ser visto na Figura 4.

Figura 4 – Comunicação definida por Kraus e Stephan (2017) entre os subdomínios presentes nas placas



Fonte: Elaborado pelo autor

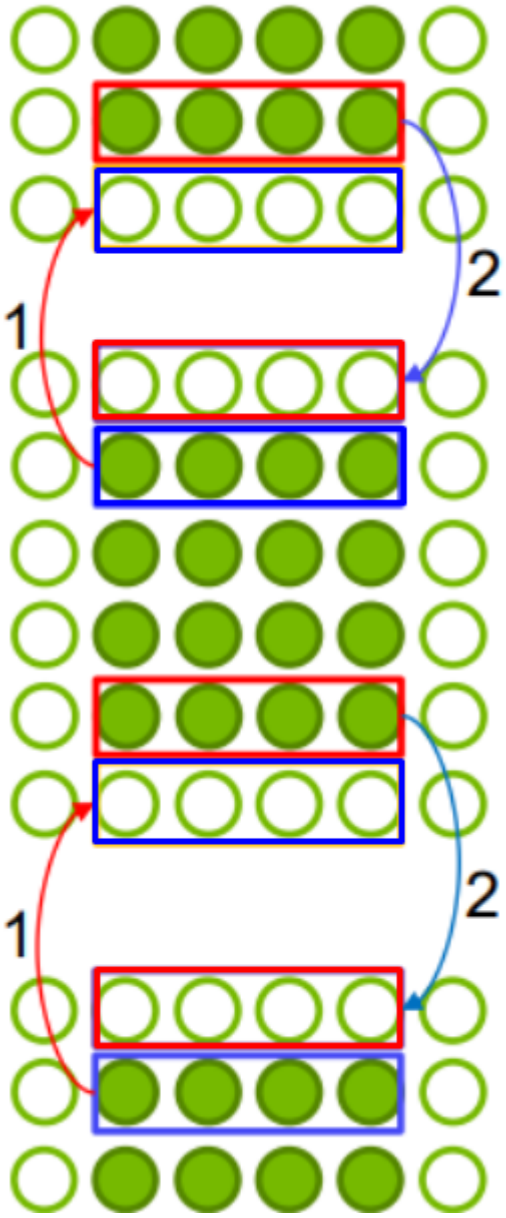
3.2.3.1 Multi-threaded copy

No modelo de comunicação definido por Kraus e Stephan (2017) como *"Multi-threaded copy"* são geradas tantas *threads* na CPU quanto dispositivos disponíveis a partir da tecnologia OpenMP. Dessa forma, cada *thread* executa a partição do problema de Kraus e Stephan (2017) que lhe foi definida em uma placa distinta.

Além disso, para serem feitas as comunicações entre as GPUs, (operação que ocorre após cada iteração completa do método Jacobi em que as placas precisam trocar valores obtidos nas bordas das separações do domínio), é utilizada a função `cudaMemCpy`, sendo possível também o uso da função `cudaMemCpyPeer`.

Conforme exemplificado pela Figura 5, Kraus e Stephan (2017) tem uma parte do domínio em cada placa gráfica (conjunto colorido em lote na Figura 5) que comunica sua borda superior com uma placa e a inferior com a outra, isso devido à divisão de domínio na horizontal. As trocas de dados necessárias são representadas pelos retângulos e setas indicando a direção de comunicação.

Figura 5 – Representação da comunicação entre placas de Kraus e Stephan (2017)



Fonte: NVIDIA Corporation, 2017

3.2.3.2 Multi-threaded p2p

No modelo intitulado *"Multi-threaded peer-to-peer"* também são geradas múltiplas *threads* para comportar os dispositivos a partir do OpenMP, entretanto, a comunicação entre as placas não tem mais o *overhead* da CPU, pois agora habilita-se a tecnologia *GPUDirect Peer-to-Peer*, na qual as GPUs podem compartilhar informações diretamente sem a necessidade da cópia feita pela CPU. Mesmo com essa habilitação, a comunicação entre placas continua ocorrendo pelas funções `cudaMemCpy` e `cudaMemCpyPeer`. Também são utilizadas as mesmas estratégias de divisão de subdomínio e comunicação supracitadas na Figura 4 e 5.

3.3 Método dos Elementos Finitos

O método dos elementos finitos é uma técnica computacional usada para aproximar soluções de problemas de valor de contorno. Este método trabalha subdividindo o domínio inicial analisado em várias partes, formando uma quantidade finita de subdomínios que são intitulados elementos, dessa forma dá-se o nome. A motivação para criação do método foi a análise de estruturas mecânicas, entretanto várias outras áreas da engenharia e da física encontraram aplicabilidade para o método (ZIENKIEWICZ; TAYLOR; ZHU, 2005).

3.3.1 Problema de Valor de Contorno

Existem equações diferenciais nas quais são definidas as condições iniciais do fenômeno, que são definidas sobre as variáveis no instante de tempo inicial t_0, os chamados problemas do valor inicial (PVI). Quando deseja-se aplicar condições a outros pontos, senão o inicial, como por exemplo x_i e x_f, há o caso do problema do valor de contorno (PVC).

Dependendo do problema modelado, podem existir limites específicos que dizem respeito à condições naturais da situação problema, um cenário onde o PVC pode ser aplicado. Outras restrições que podem ser utilizadas são as condições de Dirichlet e de Neumann, estabelecidas respectivamente sobre a variável dependente e suas derivadas.

Existem inúmeras técnicas diretas para resolução desses problemas de equações diferenciais, como uso da transformada de Laplace ou integração direta. Contudo, quando a complexidade da restrição ou do domínio são grandes, por exemplo em cenários reais da engenharia, a resolução direta é inviável, cenário onde os métodos númericos de aproximação são vantajosos e convergem para valores satisfatoriamente precisos. (BOYCE; DIPRIMA, 2010)

3.3.2 Implementação do FEM

Segundo Nikishkov, o método dos elementos finitos pode ser resumido em cinco etapas principais, são elas (NIKISHKOV, 2004):

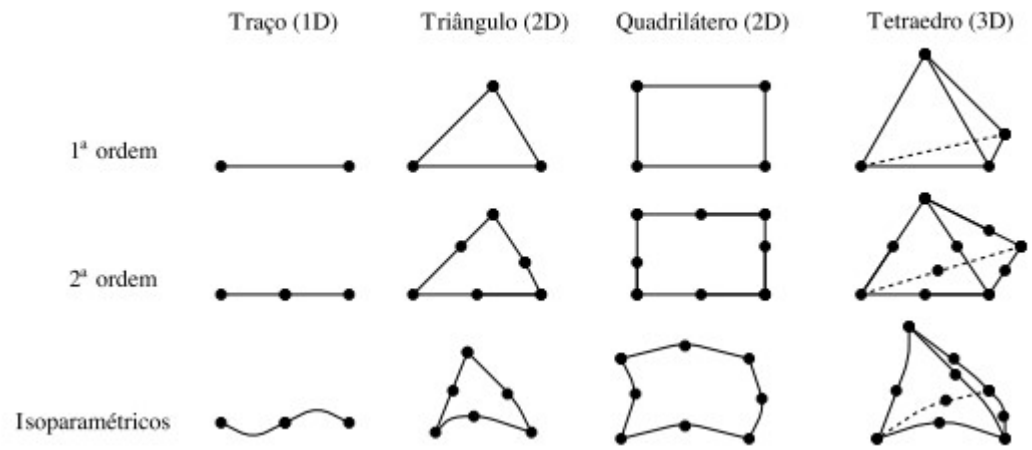
1. Discretização do domínio;
2. Seleção da função de interpolação;
3. Encontrar a propriedade dos elementos;
4. Montagem do sistema global;
5. Resolução do sistema de equações global.

3.3.2.1 Discretização do domínio

Nessa etapa, divide-se o domínio Ω em finitos elementos, que juntos compõem uma malha. Como os domínios podem ser de formatos variados, com diferentes graus de complexidade, é preciso definir a quantidade de elementos, a forma e o tamanho desses no intuito de obter uma aproximação precisa do original (GALVIS; VERSIEUX, 2011).

Em seu trabalho, Goveia (2017) demonstra possíveis formas assumidas pelos elementos que irão compor a malha. Como pode ser visto na Figura 6, a malha pode possuir elementos de vários formatos para se adequar da melhor forma ao domínio proposto.

Figura 6 – Exemplos dos elementos de uma malha

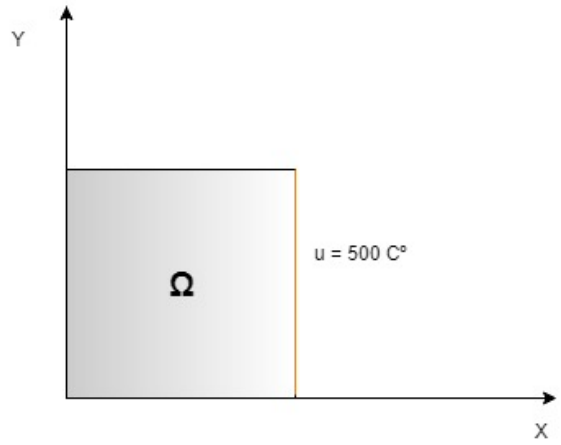


Fonte: Goveia, 2017

Enquanto elementos isoparamétricos podem ser melhores inseridos em domínios curvados, os elementos de ordem superior ampliam a precisão obtida na modelagem de problemas sem curvatura (GOVEIA, 2017).

Como exemplo, suponha o PVC de uma chapa de aço como mostrado na Figura 7.

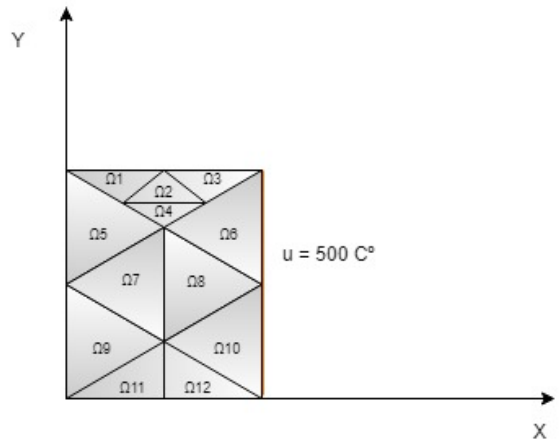
Figura 7 – O domínio Ω e sua condição de contorno



Fonte: Elaborado pelo autor

Uma possível discretização de domínio é apresentada na Figura 8

Figura 8 – O domínio Ω dividido em vários subdomínios



Fonte: Elaborado pelo autor

Cada elemento e vértice são identificados globalmente no contexto da malha e localmente no contexto dos elementos gerados, sendo utilizados para as próximas etapas.

3.3.2.2 Seleção da função de interpolação

Essa etapa consiste em definir a função que será utilizada para aproximar a solução no interior de cada elemento obtido na discretização (JIN, 2014). A função escolhida deve levar em conta as características dos elementos da malha, pois ela é utilizada para determinar a aproximação de cada nó *j* que compõe o elemento *e* com base em sua localização espacial (GOVEIA, 2017).

As funções de aproximação geralmente são polinômios de primeira ou segunda ordem, pela maior facilidade de manipulação, resultado satisfatório na aproximação e menor gasto na computação (NIKISHKOV, 2004). Entretanto, é possível utilizar ordens superiores para reduzir ainda mais o erro de aproximação (JIN, 2014).

A ordem do polinômio impacta diretamente na função escolhida (GOVEIA, 2017). Em outras palavras, o número de incógnitas da função de aproximação está relacionado ao número de nós dos elementos. Como exemplo, a Equação 3.5 (3 incógnitas) representa uma função de aproximação para um triângulo de primeira ordem, enquanto que a Equação 3.6 (6 incógnitas) pode representar um triângulo de segunda ordem.

$$f(x, y) = a + bx + cy \tag{3.5}$$

$$f(x, y) = a + bx + cy + dx^2 + exy + fy^2 \tag{3.6}$$

Escolhida a função de aproximação e considerando um subdomínio bidimensional triangular, os valores solução em cada nó de *e* podem ser encontrados resolvendo a Equação 3.7, na qual x_j e y_j são as coordenadas do nó *j* no espaço e as incógnitas *a*, *b* e *c* serão de-

terminadas a partir do sistema de equações gerado pelas equações de cada nó do elemento (GOVEIA, 2017).

$$f_j^e = a^e + b^e x_j + c^e y_j \qquad j = 1, 2, 3 \tag{3.7}$$

Realizando substituições é possível gerar a Equação 3.8, na qual é possível obter o valor de f^e em um ponto (x, y) qualquer no interior do elemento (GOVEIA, 2017). Na Figura 3.9 são representadas, para elementos triangulares no plano cartesiano, a expressão matemática de N^e, bem como as equações para encontrar os valores elementares a^e, b^e, c^e e a área A^e do triângulo.

$$f^e = \sum_{j=1}^{3} N_j^e(x, y) f_j^e = N^{eT} f^e \tag{3.8}$$

$$\begin{cases} N_j^e(x,y) = \frac{1}{2A^e}(a_j^e + b_j^e x + c_j^e y) & j = 1, 2, 3 \\ a^e = \{x_2^e y_3^e - x_3^e y_2^e, x_3^e y_1^e - x_1^e y_3^e, x_1^e y_2^e - x_2^e y_1^e\} \\ b^e = \{y_2^e - y_3^e, y_3^e - y_1^e, y_1^e - y_2^e\} \\ c^e = \{x_3^e - x_2^e, x_1^e - x_3^e, x_2^e - x_1^e\} \\ A^e = \frac{1}{2}(b_1^e c_2^e - b_2^e c_1^e) \end{cases} \tag{3.9}$$

3.3.2.3 Propriedade dos elementos

Em decorrência do cálculo em cada subdomínio a partir de uma função de aproximação, surge um resíduo r, como é visto na Equação 3.10

$$r = \Delta\phi - f \neq 0 \tag{3.10}$$

Na qual r é o resíduo gerado, o operador Δ é o laplaciano, f é uma função de excitação conhecida e ϕ é a solução procurada.

Mesmo que os resíduos pontuais sejam diferentes de zero, espera-se que o resíduo total em média seja igual a zero (GALVIS; VERSIEUX, 2011). Para isso, utiliza-se uma estratégia denominada Método dos Resíduos Ponderados, na qual pondera-se o resíduo r por meio de uma função ω adequada a partir de uma perspectiva do contínuo (ao longo do tempo) por meio da integração sobre o domínio Ω (NIKISHKOV, 2004) como visto na Equação 3.11:

$$R = \int_\Omega \omega(\Delta\phi - f) d\Omega = 0 \tag{3.11}$$

Em posse do domínio Ω discretizado, é possível analisar o residuo em cada subdomínio, utilizando funções ω_e mais simples para cada elemento e. Como exemplo, para os subdomínios triangulares no plano cartesiano, gera-se a Equação 3.12 a partir da manipulação algébrica da Equação 3.11 e atribuição da perspectiva de elemento e seus respectivos nós (GOVEIA, 2017).

$$\int_\Omega \omega^e \sum_{j=1}^{3} N_j^e \Delta\phi_j^e d\Omega = \int_\Omega \omega^e f d\Omega \tag{3.12}$$

Existem nomeações diferentes do método dos resíduos ponderados de acordo com a função ω escolhida. Uma convenientemente adotada é a de Galerkin, no qual $\omega = \phi$ (NIKISHKOV, 2004).

A partir da substituição definida por Galerkin e aplicando integração por partes é possível alcançar a Equação 3.13, que tem representação em linguagem matricial na Equação 3.14 (GOVEIA, 2017).

$$\sum_{i=1}^{3} \phi_i^e \int_\Omega \nabla N_j^e \nabla N_i^e d\Omega = \int_\Omega N_j^e f^e d\Omega \quad j = 1, 2, 3 \tag{3.13}$$

$$[K^e]\{\phi^e\} = \{b^e\} \tag{3.14}$$

3.3.2.4 Montagem do sistema global

Com a matriz de massa elementar K^e de cada elemento, é realizado o processo de montagem do sistema global, mapeando identificadores locais para os globais, utilizando uma matriz booleana L de transição que representa as conexões entre os nós (KISS et al., 2012). Ela possui uma coluna para cada vértice do domínio Ω e uma linha para cada vértice de cada elemento. Usando uma matriz E com as matrizes de elemento na diagonal principal, é possível encontrar a matriz de massa global como mostra a Equação 3.15

$$K = L^T E L \tag{3.15}$$

Seguindo o molde da Equação 3.14, utiliza-se os vetores globais K, ϕ e b para gerar o sistema global, que é da ordem do número de nós da malha. Frisa-se que em modelagens de problemas reais a matriz de massa é esparsa e possui a maioria dos elementos na diagonal principal (NIKISHKOV, 2004).

Após esse passo, a geração do sistema global ainda não está completa pois os PVCs estão sujeitos à condições de contorno. Como exemplo o sistema de Equação 3.16 possui especificidades em $\partial\Omega_1$ e $\partial\Omega_2$:

$$\begin{cases} \Delta\phi = f \quad em \quad \Omega \\ \phi = a \quad em \quad \partial\Omega_1 \\ \phi = b \quad em \quad \partial\Omega_2 \end{cases} \tag{3.16}$$

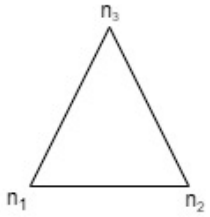
Dessa forma, alguns nós do domínio Ω possuem valores já estabelecidos e que devem ser garantidos, sendo conveniente utilizar condições de contorno de Dirichlet, substituindo incógnitas dos nós de contorno pelos seus valores reais do domínio, de forma a reduzir a ordem do sistema, atender as condições e manter a simetria (GOVEIA, 2017).

Como exemplo, suponha o sistema de Equações mostrado em 3.17, que possui 3 nós no sistema global como implica a quantidade de termos:

$$\begin{bmatrix} k_a & k_b & k_c \\ k_d & k_e & k_f \\ k_g & k_h & k_i \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix} \tag{3.17}$$

Ele pode ser expresso como um triângulo que possui os três nós globais, representado pela Figura 9:

Figura 9 – Representação do sistema com três nós



Fonte: Elaborada pelo autor

Caso esse domínio possua uma condição de contorno do tipo $\phi = a$ em $\partial\Omega_1$ e que o nó n_2 componha esse contorno, têm-se que $\phi = u_1$ em $\partial\Omega_1$, o que modifica o sistema de equações como mostra a Figura 3.18 e seguindo para a redução de ordem demonstrada na Figura 3.19:

$$\begin{bmatrix} k_a & 0 & k_c \\ 0 & 1 & 0 \\ k_g & 0 & k_i \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ u_1 \\ b_3 \end{Bmatrix} \tag{3.18}$$

$$\begin{bmatrix} k_a & k_c \\ k_g & k_i \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_3 \end{Bmatrix} \tag{3.19}$$

3.3.2.5 Resolução do sistema de equações global

Para resolução do sistema de equações global, que é um sistema linear, existem vários algoritmos, podendo ser separados nos tipos diretos e iterativos. Os primeiros com uma sequência de operações para obter a solução exata do problema, modificando diretamente os termos do sistema (BURDEN; FAIRES, 2008) e os iterativos, que como sugere Kiss et al. (2012), aproximam sucessivamente da resposta a partir de uma suposição inicial.

O adotado neste trabalho é o Método dos Gradientes Conjugados, principalmente por sua popularidade e pelo fato de ser iterativo, tornando fácil decompor o problema em partes menores (KISS et al., 2012), como explicado em detalhes na próxima seção.

3.4 Método dos Gradientes Conjugados

Nessa seção será abordado acerca do método iterativo mais popular para resolver grandes sistemas de equações lineares, o Gradiente Conjugado (SHEWCHUK, 1994).

A necessidade de um *solver* de equações lineares existe pois o produto do método dos elementos finitos é uma equação matricial da forma $Ax = b$, dessa forma, a variável alvejada é o vetor x e dispõe-se do conjunto de equações dadas pelas linhas de A e das respostas b.

Dentre os métodos de resolução existem aqueles diretos, que trabalham invertendo a matriz A e por fim encontrando o vetor x buscado, bem como os iterativos, que estabelecem valores iniciais para o vetor x que são ajustados a cada iteração, iterando até um limite de erro aceitável.

Como a matriz gerada pelo método dos elementos finitos é esparsa, ou seja, grande e com muitos valores nulos, a aplicação de métodos iterativos torna-se mais interessante, visto que os solvers iterativos, em detrimento aos diretos, possuem eficiência de memória e computam mais rápido em matrizes esparsas (SHEWCHUK, 1994, p. 1).

3.4.1 Matrizes Positivas Definidas (PD)

Considere x e y vetores. O produto interno desses pode ser expresso por $x^T y$, isto é, como mostrado abaixo na Equação 3.20

$$\begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = x_1 * y_1 + \dots + x_n * y_n \tag{3.20}$$

Ademais, $x^T y = y^T x$. Esses vetores são considerados ortogonais se $x^T y = 0$. Assim, no caso definido $Ax = b$, A pode transformar x para ter a direção vetorial invertida ou A pode transformar x em um vetor na direção ortogonal (caso Ax seja igual ao vetor nulo). Essas são transformações que não devem acontecer no CG (SHEWCHUK, 1994). Para que x não altere sua direção na transformação, a Equação 3.21 expressa a condição que deve ocorrer, que é a mesma para A ser considerada uma matriz positiva definida (PD):

$$x^T Ax > 0, \forall x \in \mathbb{R} \tag{3.21}$$

Além do uso do conceito da PD, o método dos gradientes conjugados parte da ideia do da máxima descida, que será explicado na próxima seção.

3.4.2 Método da Máxima Descida

O método da máxima descida (ou descida mais íngrime) é iterativo assim como o gradiente conjugado e objetiva encontrar o valor do vetor x da equação $Ax = b$. Segundo Bueno (2013), como os solvers iterativos, este método inicia com um vetor x_0 e a cada iteração deseja estar mais próximo do x objetivado. Para isso, seja a Equação 3.22:

$$f(k) = Ak - b \tag{3.22}$$

Deve-se obter a sua integral, que é representada na Equação 3.23:

$$f(k) = \frac{1}{2} k(Ak) - kb + c \tag{3.23}$$

Sabe-se que $b = Ax$ e c é uma constante. Se k é igual ao vetor x acrescido de um vetor q qualquer. Assume-se que A seja simétrica, sendo possível escrever

$$f(k) = f(x) + \frac{1}{2}q(Aq) \tag{3.24}$$

Garantindo-se que A é positiva definida, a segunda parte desta equação será maior que zero sempre para todo q.

Como mostrado anteriormente, a derivada de $f(k)$ é igual a $Ak - b$, então $\dot{f}(x_0)$ será igual a $Ax_0 - b$. Se a derivada indica a direção da maior elevação da função, a direção de maior decaimento será em -$\dot{f}(x_0)$, ou seja, $b - Ax_0$, que é também chamado de residual (r_0), revelando a próxima tentativa de aproximação de x na Equação 3.25:

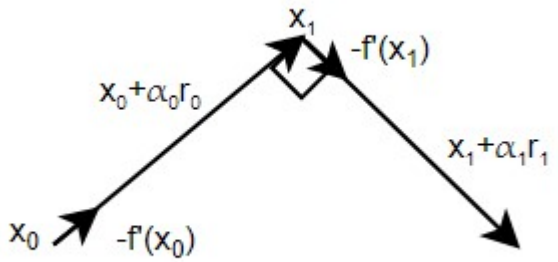
$$x_1 = x_0 + \alpha r_0 \tag{3.25}$$

Essa equação é utilizada para encontrar o valor x_1 mais próximo de x que x_0. O escalar varia sobre a reta de x_0 para x_1 e sabe-se que o objetivo é o decaimento desse erro. Por isso, a função no ponto x_1, ou seja, $f(x_0 + \alpha r_0)$ deve ter o valor mínimo possível, o que ocorre quando a derivada dessa função no ponto é igual a zero, como na Equação 3.26:

$$\dot{f}(x_0 + \alpha_0 r_0) r_0 = 0 \tag{3.26}$$

Como pode-se perceber, a derivada de $f(x_0)$ deve ser ortogonal a r_0, pois a multiplicação entre eles deve ser igual a zero para que $f(x_0)$ seja mínimo. A Figura 10 ilustra essa ortogonalidade. Dado um ponto x_0, o contrário da derivada no ponto indica a direção do próximo chute x_1, que está a $\alpha_0 r_0$ do ponto inicial x_0.

Figura 10 – Demonstração da ortogonalidade



Fonte: Elaborado pelo Autor

Como r_1 é ortogonal a r_0, substitui-se o primeiro por $b - Ax_1$, determinando-se α após destrinchar a Equação 3.27 de ortogonalidade nos passos mostrados na Equação 3.28 como sendo finalmente a Equação 3.29

$$r_1 r_0 = 0 \tag{3.27}$$

$$(b - Ax_1) r_0 = (b - Ax_0 - \alpha_0 A r_0) r_0 = \alpha_0 A(r_0 r_0) - r_0 r_0 = 0 \tag{3.28}$$

$$\alpha_0 = \frac{r_0 r_0}{A r_0 r_0} \tag{3.29}$$

Com a Equação 3.29 é possível calcular o α para a próxima iteração do método, sendo possível também encontrar o r_i que se deseja baseando-se no r_{i-1} mostrado pela Equação 3.30 já simplificada (BUENO, 2013).

$$r_i = r_{i-1} + \alpha_{i-1} A r_{i-1} \tag{3.30}$$

3.4.3 Método do Gradiente Conjugado

O método do gradiente conjugado também procura aproximar do valor x que se deseja, entretanto ele faz isso de maneira diferente. Aqui será utilizado o conceito de ortogonalização e conjugação para explicar o método, assim como abordado por Bueno (2013).

3.4.3.1 Ortogonalização

Para que dois vetores sejam ortogonais, seu produto interno deve ser zero. Partindo dessa ideia, há o método de Gram-Schmidt que ortogonaliza um conjunto de vetores a partir da fórmula de projeção.

A projeção de um vetor x em um y é o componente de x que aponta para y (BUENO, 2013). A Equação de projeção de x em y é definida em 3.31:

$$proj_y x = \frac{xy}{|y|^2} y \tag{3.31}$$

Após a computar a projeção de x em y, será obtido um vetor que tem a direção de y, por isso, ao realizar a operação da Equação 3.32, é obtido um vetor k ortogonal à y.

$$k = x - proj_y x \tag{3.32}$$

De fato, essa formulação, conhecida como processo de Gram-Schmidt pode ser realizada iterativamente eliminando de um vetor x_n do espaço as projeções de x_n sobre os $n-1$ elementos anteriores (BOYCE; DIPRIMA, 2010), tornando possível produzir vetores ortogonais para qualquer número de vetores de entrada não ortogonais, como demonstrado pela Equação 3.33 .

$$v_n = x_n - \sum_{n=1}^{n-1} proj_{vi} x_n = x_n - \sum_{n=1}^{n-1} \frac{v_i x_n}{|v_i|^2} v_i \tag{3.33}$$

Entretanto, para que seja possível gerar o conjunto de n vetores ortogonais é necessário que os vetores de entrada não ortogonais não possam ser representados pela soma dos outros vetores do conjunto, ou seja, precisam ser linearmente independentes, caso contrário será encontrado o valor zero para alguma projeção (BUENO, 2013).

3.4.3.2 Conjugação

Dados dois vetores x e y, eles são considerados conjugados à uma matriz A se $xAy = 0$. Isso ocorre caso x seja ortogonal à multiplicação de Ay. Tomando A como simétrica e positiva-definida, tem-se que xAy é igual a yAx, que é por fim igual a zero caso x e y sejam conjugados a A. Para a solução de matrizes esparsas utilizando o gradiente conjugado, a relação de conjugação de vetores é relevante e podem ser gerados conjuntos de vetores conjugados em um processo semelhante ao de Gram-Schmidt como afirma Bueno (2013).

Na Equação 3.34 é demonstrada essa formulação.

$$v_n = x_n - \sum_{n=1}^{n-1} \frac{v_i A x_n}{v_i A v_i} v_i \tag{3.34}$$

3.4.3.3 Caracterização do Método

O método dos gradientes conjugados é iterativo como o da máxima descida e também utiliza da diferença do valor do vetor x objetivado e o vetor a da iteração atual para estimar os próximos valores de x.

Como demonstrado simplificado na Equação 3.35 e expandido na Equação 3.36, o vetor x objetivado menos o vetor a da iteração atual é igual ao vetor de resíduos r.

$$x - a = r \tag{3.35}$$

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_n \end{bmatrix} \tag{3.36}$$

Enquanto que o método da máxima descida utiliza r_i como sendo a direção de a_i para a_{i+1}, o método do gradiente conjugado gera um vetor gradiente p no qual a direção inicial p_0 é iniciado com r_0 e cada direção que segue será conjugada à direção anterior (BUENO, 2013).

As Equações que geram o vetor de direções são apontadas em 3.37 e 3.38.

$$p_0 = r_0 \tag{3.37}$$

$$p_i = r_i - \sum_{n=1}^{n-1} \frac{p_n A r_n}{p_n A p_n} p_n \tag{3.38}$$

A partir delas, um exemplo de vetor *p* com três direções pode ser representado na Equação 3.39

$$\begin{bmatrix} r_0 \\ r_0 - \frac{p_0 A r_0}{p_0 A p_0} p_0 \\ r_0 - \frac{p_1 A r_1}{p_1 A p_1} p_1 - \frac{p_0 A r_0}{p_0 A p_0} p_0 \end{bmatrix} \tag{3.39}$$

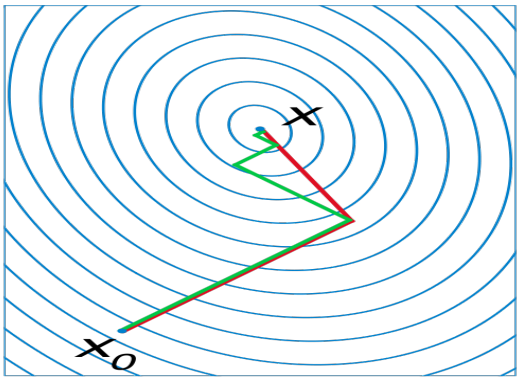
Após a definição do vetor de direções, o processo segue conforme o método da máxima descida, utilizando as Equações 3.40 e 3.41 para realizar a iteração.

$$x_{i+1} = x_i + \alpha_i p_i \tag{3.40}$$

$$\alpha_i = \frac{r_i r_i}{p_i A p_i} \tag{3.41}$$

Na Figura 11, pode-se ver a comparação de convergência entre o Método dos Gradientes Conjugados (vermelho) e Método da Máxima Descida (verde).

Figura 11 – Comparação de Convergência CG versus Máxima Descida



Fonte: Alexandrov, 2007

A partir das expressões desenvolvidas, tem-se um pseudocódigo do método dos gradientes conjugados definido conforme apresentado por Bueno (2013) no Algoritmo 1.

Algoritmo 1 Método dos Gradientes Conjugados (BUENO, 2013)

Fazer um palpite inicial x_0
$r = b - Ax$
$r_0 = p_0 = b$
for $i = 1, 2, \ldots$ **do**
  $\alpha = \frac{r_i r_i}{p_i A p_i}$
  $x_{i+1} = x_i + \alpha_i p_i$
  $r_{i+1} = r_i - \alpha A p_i$
  $p_{i+1} = r_{i+1} + \frac{r_{i+1} r_{i+1}}{r_i r_i} p_i$
  if $r_{i+1} < tol$ **then**
    break
  end
end

3.4.4 Técnica elemento a elemento (EBE)

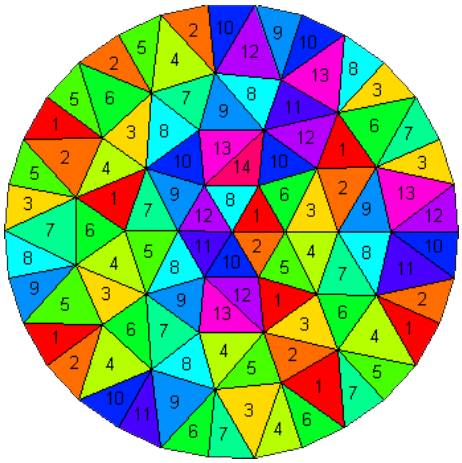
Essa técnica consiste em resolver o sistema de elementos finitos sem gerar a matriz global, dessa forma as operações serão aplicadas em cada matriz de massa dos elementos (KISS et al., 2012).

Inicialmente, essa estratégia era adotada para reduzir a quantidade de dados armazenados (por não precisar guardar a matriz global) e tirar proveito do paralelismo que é possível

ao computar cada cor (CAREY; JIANG, 1986). E a medida que o uso de GPUs se difundiu, ela continua sendo utilizada em trabalhos a fim de explorar o paralelismo possível nesses dispositivos (eg. Kiss et al. (2012) e Goveia (2017)).

Além disso, como cada elemento partilha nós com os seus vizinhos, é adotada também a coloração de malha associada ao método elemento a elemento, o que separa os elementos da malha em vários grupos de execuções diferentes (cores) que podem ser executados de forma paralela entre si, como mostra a Figura 12.

Figura 12 – Estratégia de coloração de malha



Fonte: Site Glaucio H. Paulino.[1]

Ao aplicar a técnica, cada operação do tipo matriz-vetor do Algoritmo 1 será realizada em termos da matriz de elemento (A^e) ao invés da matriz global, ou seja, não é armazenada a matriz global esparsa.

Como o método EBE atrelado à coloração resolvem o sistema atuando a partir de operações menores (por elemento) que são paralelizáveis, essa estratégia pode ser adotada em um contexto de placas gráficas.

4 Procedimentos Metodológicos

"What's behind you doesn't matter"
Enzo Ferrari

A presente pesquisa, segundo Wazlawick, é exploratória ao considerar o que se objetiva, além de ser quantitativa e baseada em experimentação, pois deseja-se comparar o desempenho das aplicações propostas com base em dados numéricos experimentais (WAZLAWICK, 2009).

4.1 Etapas de elaboração

Os procedimentos metodológicos podem ser organizados em três grandes etapas, que se seguem

4.1.1 EbE-FEM com CG em múltiplas GPUs

Em seu trabalho, Amorim et al. (2018) constrói a implementação de uma solução do EbE-FEM com o *solver* CG sendo executado em uma única GPU, para a resolução da equação matricial $Ax = b$.

Neste trabalho foi feita uma adaptação do código de Amorim para que seja possível utilizar mais placas gráficas na computação do gradiente conjugado. Assim, foram realizadas as tarefas:

1. Alterar o código de Amorim para execução dele no ambiente proposto;
2. Analisar o domínio que deu origem a malha adotada por Amorim em seu trabalho, a fim de ser aproveitado neste trabalho;
3. Analisar o código para entender a forma sobre a qual Amorim implementou o CG, com foco em listar variáveis e lógicas correlatas, para serem modificadas;
4. Arquitetar como serão dispostos o conjunto de dados entre os dispositivos na aplicação;
5. Alterar o código para alocar a memória para cada dispositivo, dividindo a computação entre todos a fim de paralelizar a atividade;
6. Modificar a função de execução do gradiente conjugado para funcionar no contexto de cada GPU em cada thread específica.

Seguidos esses passos, foi criado o EbE-FEM com CG em múltiplas GPUs.

4.1.2 Implantando Modelos de Comunicação

Nessa etapa, foi utilizado o trabalho desenvolvido por Kraus e Stephan (2017) em uma conferência de tecnologia da NVIDIA, onde eram definidos modelos de comunicação entre placas gráficas. Na conferência, Kraus e Stephan (2017) explora os tipos de comunicações no *solver* Jacobi, isso tornou possível adaptar o código criado por ele para funcionar no contexto do *solver* gradiente conjugado.

Existe um repositório no *GitHub* em que o autor disponibilizou seu código, tornando fácil a obtenção para posterior adaptação. As atividades realizadas foram:

1. Analisar o código para construção de cada modelo de comunicação, i.e. *multi-threaded copy* e *multi-threaded p2p* no intuito de definir as divergências entre eles, bem como as facilidades ou limitações em relação à pesquisa proposta;
2. Para cada modelo, analisar variáveis importantes e correlatas a este trabalho para realizar a implementação;
3. Modificar o código do EbE-FEM CG com múltiplas GPUs para funcionar com os modelos de comunicação, tornando possível, a partir de uma flag definida, fazer a escolha entre qual modelo deseja-se executar em dado instante.

Após terminada essa parte, está construído o EbE-FEM CG com múltiplas GPUs se comunicando de uma maneira específica definida a priori.

4.1.3 Comparação de Modelos

Essa última parte objetiva-se a fazer a comparação dos resultados obtidos nos modelos de comunicação, levando em conta os critérios de tempo já definidos.

Para tanto, foram realizadas as tarefas:

1. Executar a aplicação para cada modelos de comunicação, variando a malha e o número de placas para duas, três e quatro placas;
2. Levantar os dados de tempo de execução de cada modelo com cada malha, usando duas, três e quatro placas;
3. Desenvolver um gráfico para cada modelo de comunicação com cada malha, usando valores de tempo para duas, três e quatro placas;
4. Analisar os resultados, levando em conta a forma de implementação de cada tipo de comunicação e as demais variáveis que podem impactar o resultado.

5 Desenvolvimento

"Do or do not.
There is no try."
Yoda

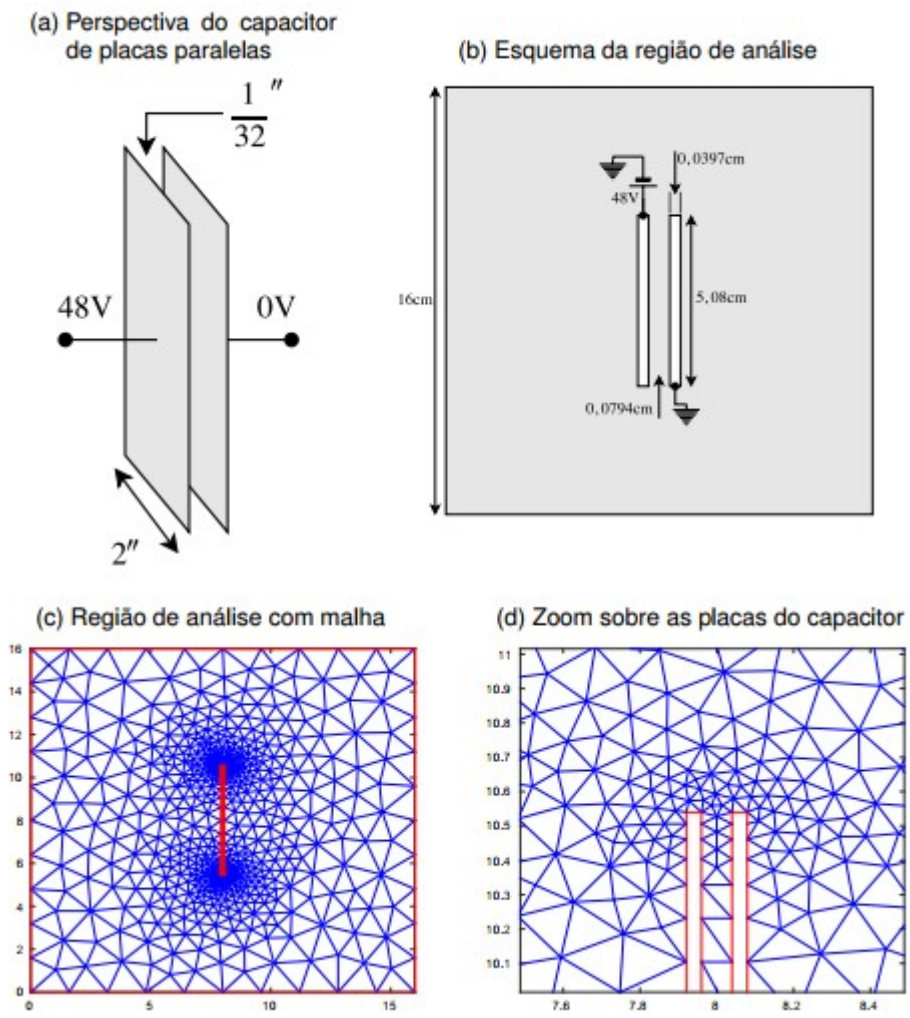
5.1 Caracterização do problema

Particularmente, este trabalho, que se consolidou a partir do trabalho desenvolvido por Goveia (2017), utiliza o mesmo problema adotado por ele e a mesma malha do método dos elementos finitos. Assim, utilizou-se o exemplo *10.4* do livro Análise de Circuitos de Boylestad (2011) que pode ser visto na Figura 13(a).

Como situação problema, existe um capacitor com duas placas quadradas paralelas de 2 polegadas de lado (*5,08cm*) e com distância de $\frac{1}{32}$ polegadas (*0,0794cm*) uma da outra. No exercício do livro é definida uma diferença de potencial (*ddp*) de *48V* entre as placas, dessa forma foi assumido que uma placa possuía *48V* e a outra estava aterrada (BOYLESTAD, 2011). Nessas condições, a distribuição do campo elétrico é calculada no espaço que circunda as placas, sendo considerada uma região quadrada de *16cm* ao redor delas como visto na Figura 13(b). Outra consideração feita é a espessura de cada placa, que é metade da largura da região entre elas, resultando em $\frac{1}{64}$ polegadas (*0,0397cm*).

O trabalho realizado por Goveia (2017) ainda apresenta graficamente como se dá a geometria e as malhas modeladas na ferramenta PDETool, como pode ser visto nas Figuras 13(c) e 13(d).

Figura 13 – Situação Problema - Campo elétrico do Capacitor



Fonte: Goveia, 2017

A partir disso, Goveia (2017) desenvolve a modelagem de duas malhas que serão utilizadas também neste trabalho. A primeira contêm 18080 triângulos e 9277 pontos, já a segunda é mais complexa e possui 40576 triângulos e 20643 pontos.

Conforme também já abordado, a solução EBE FEM adotada têm como base o trabalho de Amorim et al. (2018), no qual é realizada uma antecipação da resolução do sistema de equações sem precisar da matriz global.

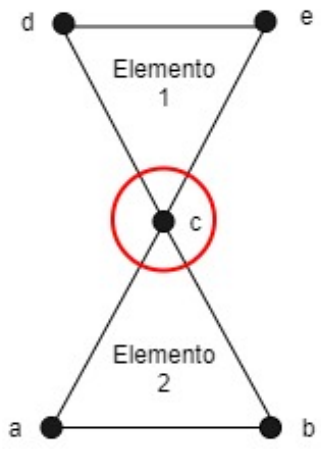
Por fim, é utilizado o CG para solucionar o conjunto de sistemas lineares gerados pela modelagem EBE. Como não é realizada a montagem da matriz global, o método CG foi modificado para solucionar o problema a partir das matrizes dos elementos, tomando como base o trabalho de Amorim et al. (2018). Dessa forma, é usado como referência o algoritmo definido por Barrett et al. (1995) e apresentado no Código 2.

Algoritmo 2 Pseudocódigo do CG (BARRETT et al., 1995)

x_0 = $\{0\}$
r_0 = $b - Ax_0$
for *i = 1,2...* **do**
  z_{i-1} = r_{i-1}
  ρ_{i-1} = ${r_{i-1}}^T z_{i-1}$
  if *i = 1* **then**
    p_1 = z_0
  else
    β_{i-1} = ρ_{i-1} / ρ_{i-2}
    p_i = z_{i-1} + $\beta_{i-1} p_{i-1}$
  end
  q_i = Ap_i
  α_i = ρ_{i-1} / ${p_i}^T q_i$
  x_i = x_{i-1} + $\alpha_i p_i$
  r_i = r_{i-1} - $\alpha_i q_i$
  if *converge(r,tol)* **then**
    break
  end
end

Essa variação do CG torna também necessária o uso da estratégia de coloração. Como já foi explicado neste trabalho, é necessário evitar condições de corrida que podem ocorrer caso dois processos acessem o mesmo endereço de memória ao mesmo tempo, como por exemplo quando dois elementos que dividem um mesmo vértice são computados, (Figura 14). Para isso são definidos grupos de execuções (as cores), nos quais todos elementos tem execuções independentes entre si, sendo computados paralelamente. No exemplo da Figura 14, os elementos 1 e 2 teriam cores diferentes, visto que não podem ser executados paralelamente.

Figura 14 – Elementos que partilham um mesmo nó



Fonte: Elaborado pelo Autor

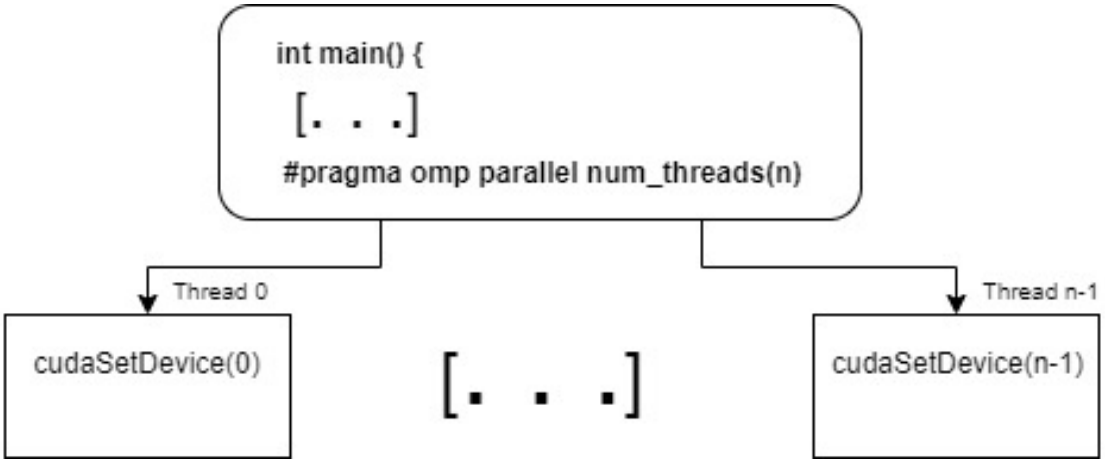
5.2 Escolhas de projeto

Para realização do trabalho, foram tomadas algumas decisões e obedecidas algumas particularidades de implementação que serão descritas nesta seção.

5.2.1 Múltiplas placas gráficas

Para usar mais de uma placa gráfica, como já abordado, Kraus e Stephan (2017) utilizam em seus modelos a biblioteca OpenMP, sendo também uma característica do presente trabalho. Tanto no modelo *"Multi-threaded Copy"* quanto no *"Multi-Threaded P2P"*, é separada uma *thread* na CPU para cada placa gráfica que será utilizada com a biblioteca OpenMP, como pode ser visto na Figura 15.

Figura 15 – Uso do OpenMP para repartição de *threads* no CPU



Fonte: Elaborado pelo Autor

Isso torna possível que, em cada *thread* da CPU seja definido o uso de uma das placas gráficas (com o `cudaSetDevice` conforme apresentado na Figura 15). Assim cada placa executa paralelamente o mesmo código sequencial definido, sendo possível trabalhar de forma particular em cada placa, utilizando o número da *thread* da CPU como referência para atribuir tarefas e conjuntos de dados específicos à placa em questão. A abordagem com OpenMP permite que sejam realizados sincronismos e trocas de informação de forma mais fácil entre as *threads*, uma vez que a biblioteca provê essas funcionalidades.

Um exemplo utilizado no trabalho é o comando `#pragma omp barrier`, a qual ao ser executada dentro de um ambiente definido como paralelo pelo OpenMP (utilizando `#pragma omp parallel`), representa uma barreira que realiza o sincronismo das *threads* da CPU, fazendo com que todas as *threads* devam chegar naquele ponto do código para que continue a execução. Essa função é importante para manter a consistência entre os dados, principalmente aqueles compartilhados por todas as *threads* da CPU (definidos pela palavra chave *shared*, na forma `#pragma omp parallel shared(var1)`).

5.2.2 Etapas do processo

Para facilitar o entendimento do trabalho e a medição de tempo, foram nomeadas cinco etapas dentro da execução do algoritmo, são elas:

1. Estruturas de dados de entrada;
2. Carga de infomações de adjacência;
3. Condição de contorno;
4. Mapeamento;
5. *Solver*.

Cada uma delas será explicada a seguir.

5.2.2.1 Estruturas de dados de entrada

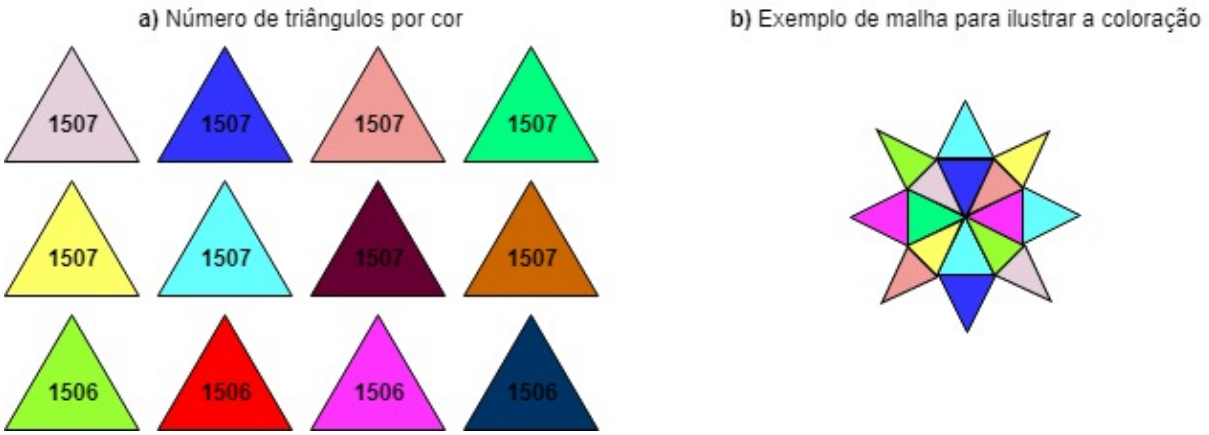
Como já mostrado acima, o algoritmo de CG utiliza um conjunto de variáveis, dentre elas os vetores p, q e r que possuem o tamanho igual ao número de pontos da malha. Como o intuito é paralelizar os cálculos principalmente do CG entre as placas, cada uma fica responsável por um fragmento dos vetores nos quais são realizados os cálculos, assim na etapa de estruturas de dados de entrada é feito um pré-cálculo do tamanho desses vetores para cada placa, utilizando a Equação 5.1 para definir o tamanho (`offset`) a partir do número de pontos (PTS) da malha e a quantidade de gpus ($numgpus$).

$$offset = \lceil PTS/numgpus \rceil \tag{5.1}$$

Caso a divisão não seja exata, é somado 1 ao `offset`, sendo que a última placa fica com o resto dos pontos. Como exemplo, considere uma malha com *9277* pontos e *18080* triângulos para três placas, as duas primeiras têm vetores com *3093* de tamanho e a última com *3091*.

Além disso, cada triângulo da malha possui uma cor específica, relativa ao grupo de execução que ele pertence. No mesmo exemplo da malha acima, seus triângulos (elementos) foram distribuídos de forma balanceada em doze grupos de execuções (cores). Esses valores já são obtidos a priori a partir dos arquivos de malha importados e podem ser observados na Figura 16(a).

Figura 16 – Exemplificação das Cores em uma malha

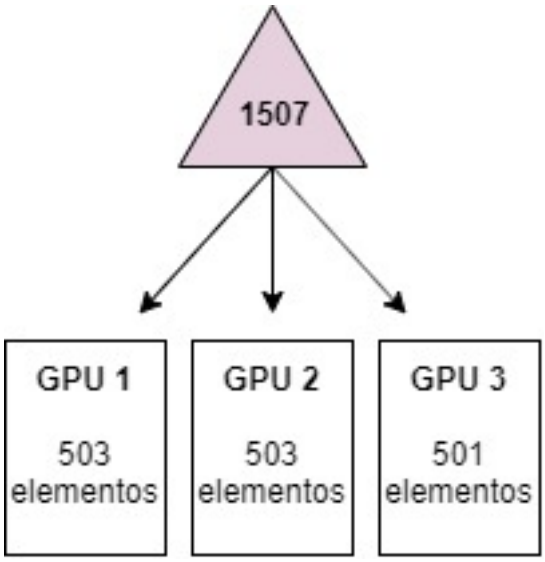


Fonte: Elaborado pelo Autor

Assim sendo, existem oito cores com *1507* elementos e quatro cores com *1506* elementos. Os elementos de cores diferentes não podem ser computados ao mesmo tempo. Na Figura 16(b) é exemplificado o uso de cores na malha, separando elementos que compartilham vértices uns com os outros. Por isso, no ambiente com múltiplas placas gráficas, a estratégia adotada foi a divisão dos elementos de uma mesma cor entre as placas gráficas, de forma que cada GPU fique encarregada de computar uma parcela dos elementos.

Assim, também foi feito um pré-cálculo da quantidade de elementos de cada cor que uma placa executa, seguindo uma lógica parecida com a de divisão de pontos. A Figura 17 demonstra como se dá a divisão de *1507* elementos de uma cor entre três placas, na qual todas as placas, exceto a última, recebem uma quantidade de elementos igual o arredondamento para cima da divisão do total de elementos da cor pela quantidade de GPUs. A última placa recebe o restante dos elementos.

Figura 17 – Separação dos elementos de uma cor entre três GPUs



Fonte: Elaborado pelo autor

Além desses cálculos, nessa etapa é feita a alocação de memória das estruturas de dados utilizadas em todas as placas (paralelamente utilizando OpenMP), bem como a cópia das informações dos triângulos, pontos e cores (que já estão contidos nos arquivos de malha) para todas as GPUs.

5.2.2.2 Carga de informações de adjacência

Na etapa de carga de informações de adjacência, a primeira *thread* é designada para realizar a organização das informações de adjacência, que nada mais é que um vetor (adj) do tamanho da quantidade de pontos que indica que determinado ponto pertence a uma quantidade de triângulos da malha.

As demais *threads*, por meio do comando `barrier` da biblioteca OpenMP, esperam que a primeira termine sua execução. Após esta sincronização, o vetor de adjacencia é alocado e vetores auxiliares são inicializados.

5.2.2.3 Condição de contorno

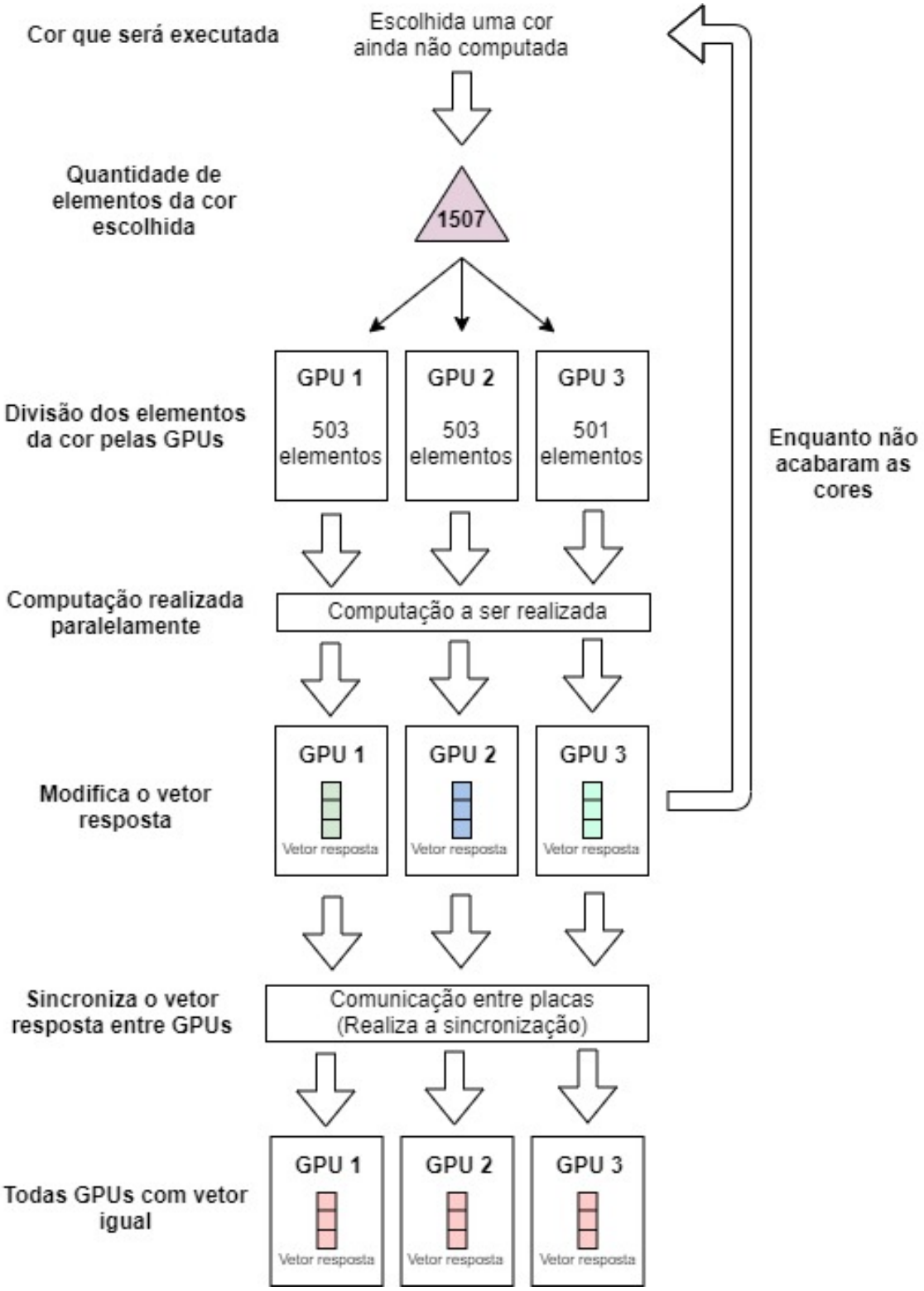
A etapa de condição de contorno é responsável por criar os vetores *b* de cada elemento, que depois serão utilizadas no gradiente conjugado para resolver o sistema de equações lineares por elemento, conforme a Equação 5.2.

$$A_e x = b_e \tag{5.2}$$

É de interesse para o funcionamento paralelo que cada placa gráfica resolva um subconjunto dos grupos de execução (i.e. resolvam uma parcela dos elementos de cada cor), assim ao ser feita uma iteração sobre as cores são computados todos os elementos.

O processo ocorre dessa forma: selecionada uma cor, cada placa obtém um subconjunto de elementos identificados com aquela cor e executa as computações sobre os triângulos sem se preocupar com inconsistências, pois os elementos de uma mesma cor são independentes entre si. Dessa forma, não há condição de corrida entre os elementos. Quando forem computadas todas as cores, cada placa gráfica terá um vetor resposta bem diferente, fruto das computações de seu subconjunto de elementos. Nesse momento, é feita a consolidação das respostas. A Figura 18 ilustra o processo.

Figura 18 – Representação do processo de computação de elementos pelas GPUs



Fonte: Elaborado pelo Autor

Nos momentos de sincronismo dos dados é que ocorre a comunicação entre as GPUs, sendo utilizadas estratégias de implementação para a operação conhecida como *all-reduce*, que é a utilizada no trabalho.

5.2.2.4 Implementação da comunicação

A operação *all-reduce* geralmente é definida por duas ações, primeiro ela obtêm conjuntos de dados (os vetores das placas por exemplo) e faz a redução deles por meio de alguma operação (por exemplo soma ou multiplicação em que a ordem não muda o resultado). Depois de finalizada essa parte, ela replica em todos os conjuntos de dados anteriores o resultado obtido.

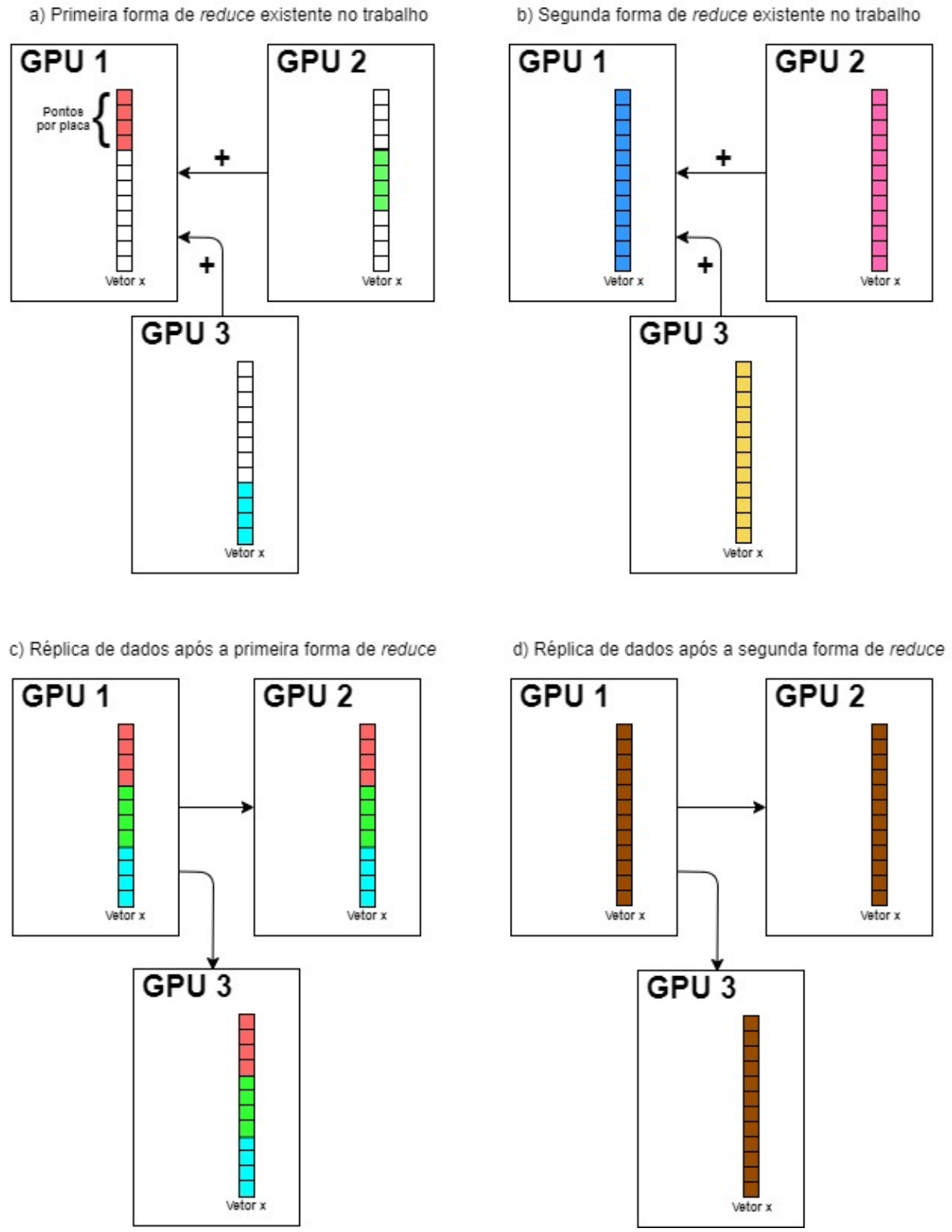
O *all-reduce* de múltiplas placas utilizado neste trabalho foi implementado de forma ingênua e simples: a primeira GPU possui um vetor auxiliar inicialmente zerado que recebe o vetor resposta da outra placa gráfica (a partir da função `cudaMemCpyPeer`). Em seguida a primeira placa realiza a soma do vetor auxiliar (que agora possui o vetor resposta de outra GPU) com o vetor resposta dela e guarda no vetor resposta, utilizando a função `cublasDaxpy` da biblioteca cuBLAS. Esse processo é feito de forma sequencial até que no vetor resposta da primeira placa possua a soma de todos os vetores resposta. Esse processo gera $numgpus - 1$ comunicações.

Como segunda parte do processo, são realizadas mais $numgpus - 1$ comunicações com a função `cudaMemCpyPeer`, copiando o vetor reduzido que está na placa *1* para as outras placas. Gerando um total de $2 * numgpus - 2$ operações de trocas de informação entre placas.

Os *reduces* podem ser de dois tipos: cada placa possui uma parte do vetor resposta que é independente às outras, ou seja, cada vetor é responsável isoladamente por posições de memória da resposta (Figura 19(a)) e no segundo cada placa tem partes do vetor resposta que devem ser somadas à outras (Figura 19(b)), entretanto ambos são resolvidos conforme explicado acima.

As Figuras 19(c) e 19(d) demonstram o processo de replicação da resposta nas outras placas gráficas.

Figura 19 – Operações de Reduce do Trabalho



Fonte: Elaborado pelo Autor

5.2.2.5 Mapeamento

Após calculados e sincronizados os vetores das condições de contorno para cada elemento entre as placas, é iniciado o processo do *solver*, ou seja, são realizadas as iterações do

gradiente conjugado.

No processo entendido como mapeamento, que tem esse nome devido ao mapeamento da contribuição de um elemento para os nós que o compõem, está contido na operação $q = A_e p$. Ou seja, é a computação da múltiplicação das matrizes de elementos (A_e) pelo vetor p utilizado no CG. Isso resulta no vetor q.

Mesmo sendo apenas uma operação, ela possui um elevado tempo de processamento, pois para realizá-la, é necessário:

- zerar o vetor q;
- fazer o sincronismo do vetor p (que já tinha sido anteriormente trabalhado) entre as placas gráficas;
- iterar sobre as cores enviando um subgrupo de elementos para serem executados em cada GPU;
- fazer a multiplicação da matriz desses elementos pelo vetor p;
- incrementar o vetor q de cada placa com os resultados;
- fazer o sincronismo (*all-reduce*) do vetor q entre as placas.

Como percebido pela listagem de tarefas, essa etapa executa duas reduções, assim é a mais complexa em operações de comunicação entre placas.

Vale ressaltar que como esse processo é executado em cada iteração do *solver*, para resultados do tempo desse processo, são somados os tempos de todas execuções.

5.2.2.6 *Solver*

A etapa de *solver* compreende todas as demais operações que são realizadas no CG. É necessário também, ao final de cada iteração, a sincronização do vetor resíduo r entre as placas gráficas, visto que é feito o teste de convergência para avaliar se o erro está abaixo da tolerância definida (neste trabalho, foi utilizada a tolerância de 10^{-6}).

5.2.2.7 Implementação dos modelos de comunicação

Os dois modelos de comunicação definidos por Kraus e Stephan (2017) e adaptados neste trabalho, a saber *"Multi-Threaded Copy"* (MT-Copy) e *"Multi-Threaded P2P"* (MT-P2P) são implementados da mesma forma em todas as etapas. Como já abordado, a solução *peer to peer* do modelo MT-P2P apenas precisa ser habilitada entre as placas que a utilizarão. Como a comunicação das placas foi realizada a partir da ligação da primeira placa com as outras (para o processo de *reduce-all*), foi necessário habilitar a comunicação das demais placas com a primeira a partir da função `cudaDeviceEnablePeerAccess(d1,d2)`, na qual `d1` e `d2` representam respectivamente o dispositivo 1 e 2 que estão sendo ligados.

Assim, antes de iniciar a primeira etapa do algoritmo, é utilizada uma função para habilitar a comunicação entre as placas (ativando o método P2P).

5.3 Resultados

5.3.1 Condições de execução

Os testes foram realizados em um servidor remoto da Universidade Federal de Minas Gerais (UFMG) acessado por SSH que possui oito placas gráficas *NVIDIA Tesla K40m*, cada uma com 745MHz de clock e memória de *12 GB GDDR5*. Além disso, o servidor possui dois CPUs *Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz*.

Uma ressalva é que o ambiente utilizado para testes possui 8 placas gráficas, mas dessas oito, quatro estavam ligadas diretamente a um CPU, pelo *PCI Express* e as outras quatro estavam ligadas à outra CPU por outro *PCI Express*. Como já é sabido, a solução P2P só é habilitável caso as placas que serão ligadas compartilhem o mesmo *PCI Express*. Essa limitação faz com que quantidades maiores que quatro placas não possam ser utilizadas em testes no ambiente proposto e para o modelo MT-P2P. A Figura 20 mostra como as placas se comunicam fisicamente a partir do comando `nvidia-smi topo -m`.

Figura 20 – Padrões de comunicação físicos entre as placas

```
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    CPU Affinity
GPU0     X      PIX     PHB     PHB     SYS     SYS     SYS     SYS     0-5,12-17
GPU1    PIX      X      PHB     PHB     SYS     SYS     SYS     SYS     0-5,12-17
GPU2    PHB     PHB      X      PIX     SYS     SYS     SYS     SYS     0-5,12-17
GPU3    PHB     PHB     PIX      X      SYS     SYS     SYS     SYS     0-5,12-17
GPU4    SYS     SYS     SYS     SYS      X      PIX     PHB     PHB     6-11,18-23
GPU5    SYS     SYS     SYS     SYS     PIX      X      PHB     PHB     6-11,18-23
GPU6    SYS     SYS     SYS     SYS     PHB     PHB      X      PIX     6-11,18-23
GPU7    SYS     SYS     SYS     SYS     PHB     PHB     PIX      X      6-11,18-23

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing a single PCIe switch
  NV#  = Connection traversing a bonded set of # NVLinks
```

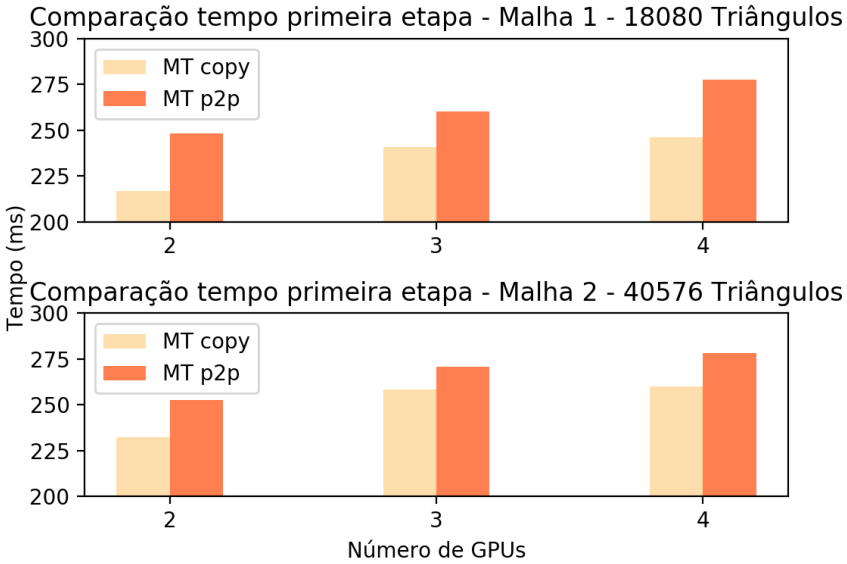
Fonte: Elaborado pelo Autor

Foram utilizados dois tamanhos de malhas nos testes: uma com 18080 triângulos (9277 pontos) e outra com 40576 (20643 pontos). Além disso, para cada uma dessas malhas, foram medidos os tempos das cinco etapas acima mencioadas, sendo: (i)Estruturas de dados de entrada, (ii)carga de informações de adjacência, (iii)condição de contorno, (vi)mapeamento e (v)*solver*. Os valores apresentados em cada gráfico são a média do tempo total gasto em cinco execuções de cada algoritmo.

Algumas considerações podem ser feitas sobre os resultados. As escolhas de implementação são fatores que podem beneficiar ou prejudicar cada método de comunicação utilizado. Também é resaltado que os valores se basaram em 5 execuções, podendo um resultado afetar muito a média geral.

Como notado na Figura 21, o tempo de execução da etapa de estruturas de dados de entrada (etapa 1) é um pouco maior no modelo de comunicação *MT-p2p* para duas, três e quatro placas gráficas em ambas malhas abordadas. Isso pode ser ocasionado devido a possíveis tratativas necessárias no momento da alocação de memória em placas com o *peer to peer* habilitado (maior parte do processamento realizado na primeira etapa).

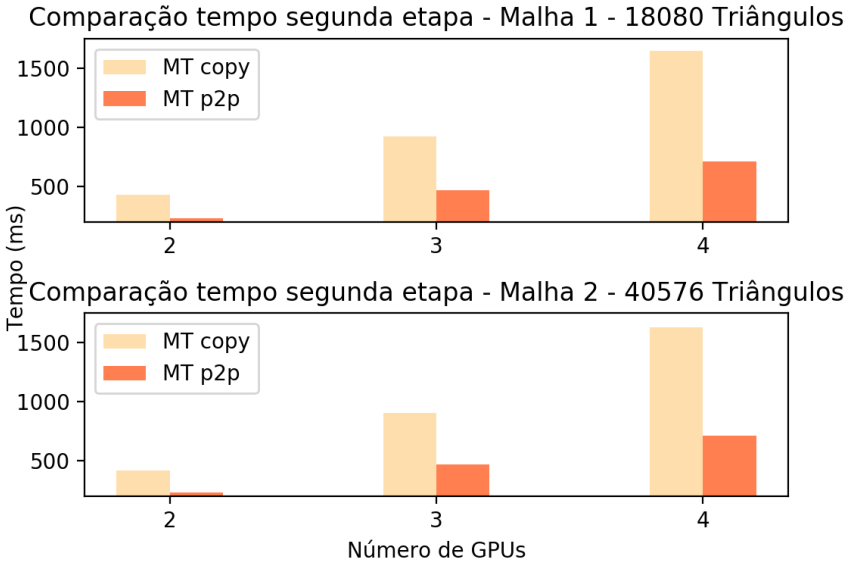
Figura 21 – Gráfico comparativo - Estruturas de dados de entrada (i)



Fonte: Elaborado pelo Autor

A Figura 22 indica a etapa de carga de informações de adjacência (etapa 2). A primeira percepção que se tem são os tempos muito parecidos em ambas as malhas. Outra percepção é o tempo bem menor que o modelo de comunicação *MT-p2p* apresenta. Isso pode ser relacionado com as tratativas no modelo de alocação de memória citadas na etapa anterior, que tornam mais fáceis as cópias, mesmo vindas da CPU.

Figura 22 – Gráfico comparativo - Carga de infomações de adjacência (ii)

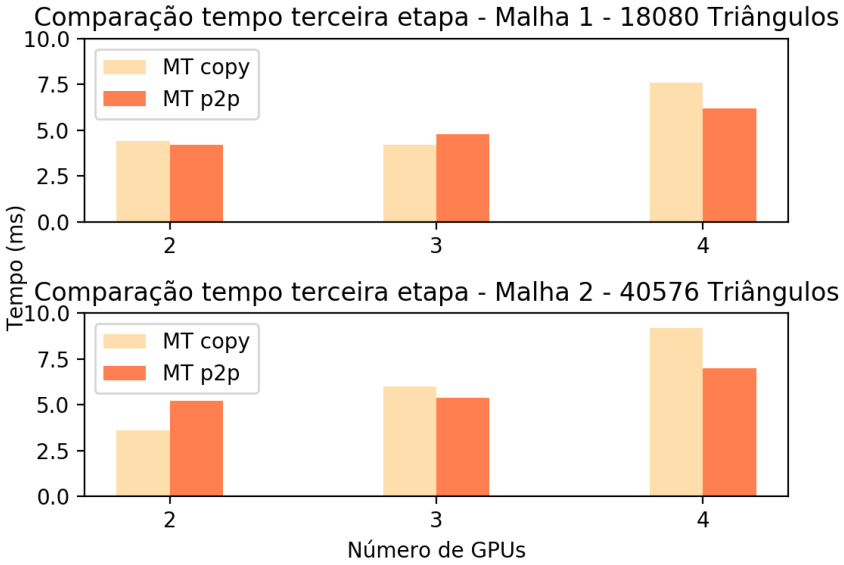


Fonte: Elaborado pelo Autor

A Figura 23 mostra os tempos de condição de contorno (etapa 3). Como esperado, o modelo *MT-p2p* é um pouco mais rápido que o *MT-copy* para quatro placas em ambas as

malhas. Como também pode ser notado, existem resultados em que o modelo MT copy tem um tempo inferior ao MT p2p, sendo consideradas as colocações supracitadas.

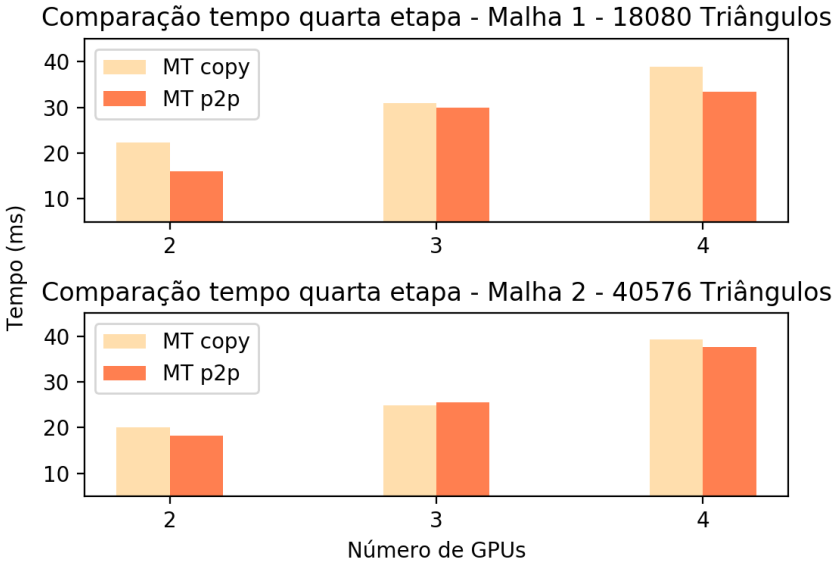
Figura 23 – Gráfico comparativo - Condição de contorno (iii)



Fonte: Elaborado pelo Autor

A Figura 24 demonstra a comparação do mapeamento (etapa 4) entre os modelos. Pode ser percebido que o modelo *MT-p2p* tem desempenho igual ou melhor que o modelo *MT-copy* conforme esperado. Isso se mostrou interessante, pois é nessa etapa que ocorre a operação da multiplicação $q = Ap$, a mais complexa realizada pelo $solver$ e que depende de dois $reduces$.

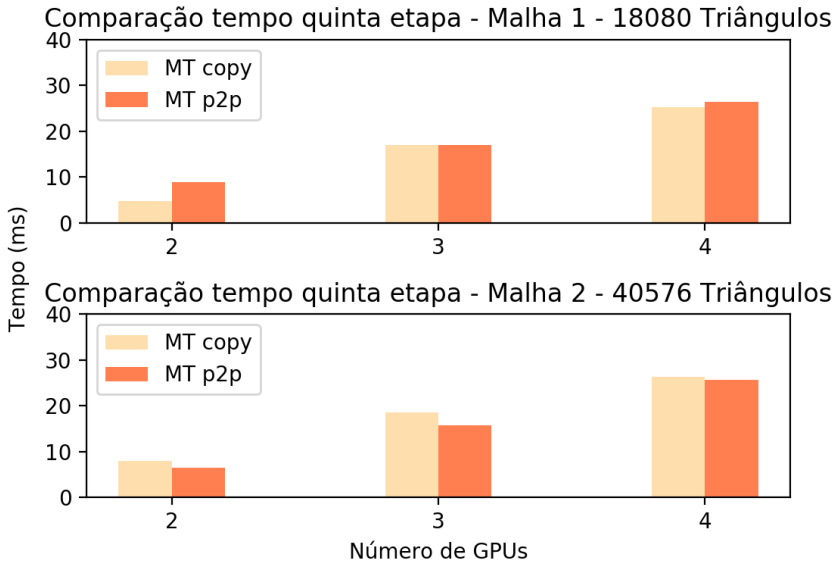
Figura 24 – Gráfico comparativo - Mapeamento (iv)



Fonte: Elaborado pelo Autor

Enfim, a Figura 25 mostra a comparação do $solver$ (etapa 5). É notada na malha 2 um tempo mais rápido do modelo *MT-p2p*, entretanto na malha menor o modelo *MT-copy* tem desempenho melhor, também sendo consideradas as colocações supracitadas.

Figura 25 – Gráfico comparativo - *Solver* (v)



Fonte: Elaborado pelo Autor

Outra percepção verificada em todos os gráficos é o aumento do tempo das etapas conforme aumenta-se o número de placas. As principais causas, além das inicialmente mencionadas, que podem ser relacionadas à isso são:

- A escolha da forma de comunicação abordada neste trabalho, que acaba por tornar as demais placas dependentes da primeira para realizar o *reduce*;
- O tamanho das malhas ser relativamente pequeno, fazendo com que a computação de cada placa gráfica tenha um tempo menor que o *overhead* de comunicação entre elas, o que torna a opção com menos placas ser mais rápida.

Com interesse exploratório, também foram executados testes da resolução do problema com apenas uma GPU. Dado as propriedades das malhas, que possuem poucos elementos para a capacidade do hardware e do problema que é simples, uma placa é suficiente para as computações e é de se esperar que ela obtivesse um desempenho melhor que as demais abordagens supracitadas. Afinal, é mais custosa a comunicação entre placas do que a computação a ser realizada neste caso. Entretanto, o objetivo do trabalho é explorar a comunicação entre as GPUs, assim confirmando as escolhas adotadas.

6 Conclusão

"Ikibu"
Perseguidores de Adapak, em o Espadachin de Carvão

A partir do estudo a respeito de modelos de comunicação envolvendo placas gráficas, o trabalho em questão foi elaborado com a proposta de comparar os modelos de comunicação definidos por Kraus e Stephan (2017), ou seja, *multi-threaded copy* (MT-copy) e *multi-threaded P2P* (MT-P2P). Para tanto foi implementado o método dos elementos finitos elemento por elemento com o *solver* de Gradiente Conjugado em um contexto de múltiplas Unidades de Processamento Gráfico, utilizando como métrica o tempo da execução de cinco subetapas do processo. Tomando como base o trabalho de Kraus e Stephan (2017) e Amorim et al. (2018), foi implementado um *solver* de gradiente conjugado na perspectiva de elementos adaptado aos modelos de comunicação supracitados.

Os modelos foram comparados conforme previsto e notou-se que na etapa de mapeamento, que tem uma complexidade de comunicações maior, o MT-p2p performou igual ou melhor que o MT-copy, entretanto em outras etapas é notada uma variação entre os melhores tempos conforme altera-se o número de placas gráficas e o tamanho da malha.

É perceptível também que para problemas e malhas mais simples, como as utilizadas neste trabalho, o *overhead* de comunicação entre placas é mais custoso que as computações realizadas, sendo alcançado um desempenho melhor na execução em menos placas gráficas. Isto foi verificado a partir da execução deste problema proposto em apenas uma placa gráfica, que obteve um desempenho melhor que os modelos com mais dispositivos. Entretanto, o objetivo do trabalho era explorar a comunicação entre as GPUs, assim confirmando as escolhas adotadas.

Além disso, também é mencionado que esse trabalho utiliza escolhas de projeto específicas, bem como um cálculo de tempo médio baseado em poucas execuções do algoritmo, sendo interessantes novas abordagens para comparar como esses fatores influenciam no trabalho proposto, aumentando a fidelidade do resultado.

6.1 Trabalhos futuros

Este trabalho focou no desenvolvimento dos modelos de comunicação *Multi Threaded Copy* e *Multi Threaded P2P*. Foram utilizadas até quatro placas gráficas na resolução do gradiente conjugado em uma perspectiva de elemento, com dois modelos de malha, colhendo tempo de cinco etapas e tomando como base escolhas de implementação específicas, que podem favorecer ou não o trabalho proposto. Por isso, como sequência deste trabalho, recomenda-se:

- estudar os resultados de outros modelos de comunicação na resolução do mesmo problema;
- realizar testes em malhas mais complexas e possivelmente com mais dimensões;
- estudar o impacto dos modelos de comunicação em outros *solvers*;
- estudar outras escolhas de implementação e como elas impactam nos modelos de comunicação.

Referências

AGOSTINI, E.; ROSSETTI, D.; POTLURI, S. GPUDirect Async: Exploring GPU synchronous communication techniques for InfiniBand clusters. *Journal of Parallel and Distributed Computing*, Elsevier Inc., v. 114, p. 28–45, 2018. ISSN 07437315. Disponível em: <https://doi.org/10.1016/j.jpdc.2017.12.007>. Citado na página 23.

AMORIM, L. P. et al. GPU Finite Element Method Computation Strategy Without Mesh Coloring. *Cefc*, v. 5, p. 1–4, 2018. Citado nas páginas 12, 13, 16, 39, 42 e 56.

BARRETT, R. et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. *Mathematics of Computation*, v. 64, n. 211, p. 1349, 1995. ISSN 00255718. Disponível em: <http://www.jstor.org/stable/2153507?origin=crossref>. Citado nas páginas 42 e 43.

BOYCE, W. E.; DIPRIMA, R. C. *Equacões Diferenciais Elementares e Problemas de Valores de Contorno*. [S.l.: s.n.], 2010. ISBN 978521617563. Citado nas páginas 27 e 35.

BOYLESTAD, R. L. *Análise de Circuitos*. [S.l.]: Pearson, 2011. ISBN 978-85-64574-20-5. Citado na página 41.

BUENO, A. L. C. *Resoluçao de sistemas de equaçoes lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCLTM*. 2013. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, 2013. Citado nas páginas 16, 33, 35, 36 e 37.

BURDEN, R.; FAIRES, J. *Análise numérica*. Cengage Learning, 2008. ISBN 9788522106011. Disponível em: <https://books.google.com.br/books?id=pf8XPwAACAAJ>. Citado na página 32.

Camargos, A. F. P. de; Silva, V. C. Performance analysis of multi-gpu implementations of krylov-subspace methods applied to fea of electromagnetic phenomena. *IEEE Transactions on Magnetics*, v. 51, n. 3, p. 1–4, March 2015. ISSN 0018-9464. Citado na página 16.

CAREY, G. et al. Element-by-element vector and parallel computations. *International Journal for Numerical Methods in Biomedical Engineering*, Wiley-Blackwell, v. 4, n. 3, p. 299–307, 1 1988. ISSN 2040-7939. Citado na página 12.

CAREY, G. F.; JIANG, B.-N. Element-by-element linear and nonlinear solutions. *Communications in Applied Numerical Methods*, v. 2, p. 145–153, 03 1986. Citado na página 38.

CEVAHIR, A.; NUKADA, A.; MATSUOKA, S. Fast conjugate gradients with multiple gpus. In: . [S.l.: s.n.], 2009. v. 2009, p. 893–903. Citado na página 13.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP*. Cambridge: The MIT Press, 2008. ISBN 978-0-262-53302-7. Citado na página 21.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. *Professional CUDA C Programming*. Indianapolis: John Wiley & Sons, Inc., 2014. v. 53. 1689–1699 p. ISSN 1098-6596. ISBN 978-1-118-73932-7. Citado nas páginas 18 e 19.

DZIEKONSKI, A. et al. Generation of large finite-element matrices on multiple graphics processors. n. December 2012, p. 204–220, 2012. Citado na página 13.

GALVIS, J.; VERSIEUX, H. *Introdução à Aproximação Numérica de Equações Diferenciais Parciais Via o Método de Elementos Finitos*. Rio de Janeiro: IMPA, 2011. Citado nas páginas 12, 27 e 30.

GOVEIA, T. D. S. *Investigação do uso do paralelismo de ambientes multiprocessados para a aceleração eficiente da solução de problemas modelados com o método dos elementos finitos*. 2017. Citado nas páginas 16, 22, 28, 29, 30, 31, 38, 41 e 42.

GRISA, M. *GPU Computing : Implementação do método do Gradiente Conjugado utilizando CUDA*. 2010. Tese (Doutorado) — Universidade de Caxias do Sul, 2010. Citado na página 16.

HU, J.; QUIGLEY, S. F.; CHAN, A. An element-by-element preconditioned conjugate gradient solver of 3d tetrahedral finite elements on an fpga coprocessor. In: . [S.l.: s.n.], 2008. p. 575 – 578. ISBN 978-1-4244-1960-9. Citado na página 16.

JIN, J. *The Finite Element Method in Electromagnetics*. 3rd. ed. [S.l.]: Wiley-IEEE Press, 2014. ISBN 1118571363, 9781118571361. Citado na página 29.

KISS, I. et al. Parallel realization of the element-by-element FEM technique by CUDA. *IEEE Transactions on Magnetics*, v. 48, n. 2, p. 507–510, 2012. ISSN 00189464. Citado nas páginas 12, 13, 14, 16, 31, 32, 37 e 38.

KRAUS, J.; STEPHAN, J. MULTI-GPU PROGRAMMING MODELS. In: *GPU Technology Conference*. [S.l.]: NVIDIA Corporation, 2017. Citado nas páginas 6, 7, 8, 13, 14, 16, 21, 22, 23, 24, 25, 26, 40, 44, 51 e 56.

MIOTTO, C. M.; CARGNELUTTI, J.; MACHADO, V. M. *Aplicações Das Equações Diferenciais Na Modelagem Matemática Da Dilatação / Contração Térmica De Cabos Da Rede*. Paraná: UTFPR, 2013. Citado na página 12.

NIKISHKOV, G. P. *Introduction to the Finite Element Method*. Aizu: University of Aizu, 2004. ISSN 0445-2429. Citado nas páginas 27, 29, 30 e 31.

NVIDIA. *CUDA Toolkit 4 . 1 CUFFT Library*. Santa Clara: NVIDIA Corporation, 2012. Citado na página 22.

PEREIRA, O. J. B. a. *Análise De Estruturas Ii Introdução Ao Método Dos Elementos Finitos Na Análise De Problemas Planos De Elasticidade*. Lisboa: Instituto Superior Técnico, 2005. Citado na página 12.

SANDERS, J.; KANDROT, E. *Cuda By Example An Introduction to General-Purpose GPU Programming*. Boston: [s.n.], 2011. ISBN 9780131387683. Citado nas páginas 18, 19, 20 e 21.

SCHWARZ, H. R. The method of conjugate gradients in finite element applications. *Zeitschrift für angewandte Mathematik und Physik ZAMP*, v. 30, n. 2, p. 342–354, 1979. ISSN 00442275. Citado na página 13.

SENA, M. C. R.; COSTA, J. A. d. C. *Tutorial OpenMP C/C++*. 1. ed. Maceió: LCCV-UFAL, 2008. Citado na página 21.

SHEWCHUK, J. R. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. 1994. Citado nas páginas 32 e 33.

SILVA, M. A. *Modelagem Matemática : Equações diferenciais ordinárias em cursos de graduação*. 2014. Tese (Doutorado) — IFSP, 2014. Citado na página 12.

SPAMPINATO, D. *Modeling Communication on Multi-GPU Systems*. 2009. Tese (Thesis) — Norwegian University of Science and Technology, 2009. Citado nas páginas 13, 14 e 16.

WAZLAWICK, R. S. *Metodologia de Pesquisa para Ciência da Computação*. 6ª. ed. [S.l.]: Elsevier, 2009. ISBN 978-85-352-3522-7. Citado na página 39.

XU, J. et al. Capacitance extraction of three-dimensional interconnects using element-by-element finite element method (ebe-fem) and preconditioned conjugate gradient (pcg) technique. *IEICE Transactions on Electronics*, E90C, 01 2007. Citado na página 16.

ZASPEL, P.; GRIEBEL, M. A multi-GPU accelerated solver for the Navier-Stokes equations. *Idea*, p. 1–8, 2010. ISSN 1865-2034. Citado na página 13.

ZIENKIEWICZ, O. C.; TAYLOR, R. L.; ZHU, J. Z. *The Finite Element Method: Its Basis & Fundamentals*. [S.l.]: Elsevier, 2005. ISBN 0750663200. Citado na página 27.