CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Thiago de Sousa Goveia

INVESTIGAÇÃO DO USO DO PARALELISMO DE AMBIENTES MULTIPROCESSADOS PARA A ACELERAÇÃO EFICIENTE DA SOLUÇÃO DE PROBLEMAS MODELADOS COM O MÉTODO DOS ELEMENTOS FINITOS

Timóteo

2017

Thiago de Sousa Goveia

INVESTIGAÇÃO DO USO DO PARALELISMO DE AMBIENTES MULTIPROCESSADOS PARA A ACELERAÇÃO EFICIENTE DA SOLUÇÃO DE PROBLEMAS MODELADOS COM O MÉTODO DOS ELEMENTOS FINITOS

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: Marcio Matias Afonso
Coorientador: Lucas Pantuza Amorim

Timóteo

2017

Thiago de Sousa Goveia

Investigação do uso do paralelismo de ambientes multiprocessados para a aceleração eficiente da solução de problemas modelados com o Método dos Elementos Finitos

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.

Trabalho aprovado. Belo Horizonte, 7 de dezembro de 2017

Prof. Márcio Matias Afonso

Prof. Lucas Pantuza Amorim

Prof. Eduardo Henrique da Rocha Coppoli

Dedico a
todos que contribuíram para a minha educação.

Agradecimentos

Agradeço aos meus familiares e amigos pelo apoio e paciência, dada a minha ausência nestes anos. Em especial, agradeço à minha mãe, Terezinha, por ser um exemplo de sabedoria e dedicação e à minha tia “Nem” pelo incentivo e carinho.

Agradeço aos professores Carlos Eduardo Paulino, Rodrigo de Oliveira Gaiba, Aléssio Miranda Junior e Lucas Pantuza Amorim, os quais foram peças chave para a minha permanência no curso. Ao professor Lucas, agradeço também por ter apontado o caminho que levou à este trabalho.

Agradeço ao professor Marcio Matias Afonso pelo voto de confiança e pela prontidão ao aceitar fazer parte deste trabalho.

Agradeço aos meus companheiros de jornada, em especial aos meus amigos Mike e Tamires, mas também aos que atendem pelo título de “filhotes” da Karini e claro, a ela própria, pela companhia e amizade.

Agradeço ao Moisés, meu ponto de equilíbrio, por sua paciência, apoio e dedicação.

Por fim agradeço a Deus por todas as oportunidades, pelo sustento, aprendizado e por não ter me deixado desanimar, mesmo com todas as dificuldades. Agradeço pelos bons momentos e pelas pessoas memoráveis que conheci ao longo do curso.

“Desesperar Jamais.
Aprendemos muito nesses anos”.
Ivan Lins

Resumo

O Método dos Elementos Finitos é uma técnica numérica destinada à modelagem e solução de equações diferenciais parciais. Devido à sua boa precisão na aproximação do resultado ele é geralmente utilizado em ferramentas computacionais para simulação de fenômenos da ciência e da engenharia. Atividades que demandam simulação em tempo real ou grande precisão, como por exemplo o processamento de imagens médicas ou o projeto de um carro de fórmula 1 exigem que a execução seja realizada com a máxima performance possível de forma a antecipar a obtenção dos resultados. Nas arquiteturas modernas de CPUs este objetivo é geralmente alcançado pelo uso de múltiplas *threads* em sistemas de memória compartilhada ou múltiplos processos em memória distribuída. No entanto, outras formas de paralelismo podem ser utilizadas. Afim de se obter maior eficiência na solução do problema em termos do tempo de execução e capacidade de processamento, é apresentada neste trabalho uma investigação do impacto da adoção dos diferentes níveis de paralelismo existentes em ambientes multiprocessados. Tal investigação é feita por meio da implementação elemento a elemento do método dos Gradientes Conjugados, paralelizado por meio da coloração da malha e da decomposição do domínio pelo método de Jacobi-Schwarz. Os resultados obtidos foram satisfatórios, sobretudo os de granulação fina associados ao paralelismo dos niveis de instrução e de dados. O máximo *speedup* atingido foi igual a $8,6$.

Palavras-chave: Paralelismo de hardware, Método dos Elementos Finitos, Método dos Gradientes Conjugados, Decomposição de domínio.

Abstract

The Finite Element Method is a numerical technique for modeling and solving partial differential equations. Due to its good precision at the approximation of the solution, it is generally used in computational tools to simulate science and engineering phenomena. Tasks that require real-time or high-precision simulation, such as medical image processing or the design of a formula 1 cars, require that execution to be done with maximum performance in order to anticipate the results. In modern CPUs architectures this goal is usually achieved by the use of multiple threads in shared memory systems or multiple processes in a distributed memory system. However, other types of parallelism can be used. In order to maximize the efficiency at the solution of the problem in terms of execution time and processing power, this paper present an investigation of the impact caused by the adoption of different levels of parallelism at multiprocessing environments. Such investigation is done with the implementation of element by element Conjugated Gradients method, parallelized by the mesh coloring and the Jacobi-Schwarz domain decomposition. The results were satisfactory, especially the fine-grained ones, associated with the level of instructions and data parallelism. The maximum speedup reached was 8.6.

Keywords: Hardware Parallelism, Finite Element Method, Conjugated Gradient Method, Domain Decomposition.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Interface de programação de aplicações
FEM	Método dos elementos finitos
FDM	Método das diferenças finitas
IDE	Ambiente de desenvolvimento integrado
PVC	Problema de valor de contorno
PVI	Problema de valor de inicial
CG	Gradientes conjugados
GMRes	Resíduo mínimo generalizado
EBE	Elemento a elemento
SMP	Multiprocessamento simétrico
GPU	Unidade de processamento gráfico
BiCG	Gradiente bi-conjugado
FPGA	Arranjo de Portas Programáveis em Campo
GPGPU	GPU para própósito geral
SIMD	Único fluxo de instruções, múltiplo fluxo de dados
SIMT	Único fluxo de instruções, múltiplas *threads*
SISD	Único fluxo de instruções, único fluxo de dados
MIMD	Múltiplo fluxo de instruções, múltiplo fluxo de dados
MISD	Múltiplo fluxo de instruções, único fluxo de dados
BiCGStab	BiCG estabilizado
CPU	Unidade central de processamento
CMOS	Semicondutor de metal-óxido complementar
ILP	Paralelismo em nível de instrução
TLP	Paralelismo em nível de *threads*
MPI	Interface de passagem de mensagem

DLP	Paralelismo em nível de dados
UMA	Memória de acesso uniforme
NUMA	Memória de acesso não uniforme
SMP	Multiprocessador simétrico
DSM	Memória compartilhada distribuída
GS	Gauss-Seidel
SOR	Sobrerrelaxamento sucessivo
SPD	Simétrico positivo definido
PD	Positivo definido
JSM	Método de Jacobi-Schwarz
RAS	Método aditivo restrito de Schwarz
PDETOOL	Ferramentas do MATLAB®para equações diferenciais parciais
PARTOOL	Ferramentas do MATLAB®para computação paralela
FETI	Partição e interconexão de elementos finitos

Sumário

1 Introdução

"Se você falar com um homem numa linguagem que ele compreende, isso entra na cabeça dele. Se você falar com ele em sua própria linguagem, você atinge seu coração".
Nelson Mandela

Devido ao aumento da demanda por desempenho computacional, dispositivos que possibilitam a execução paralela se estabeleceram no mercado da informática. Por meio das arquiteturas com múltiplos e muitos núcleos, do inglês *multicore* e *manycore* respectivamente, é possível se executar paralela e/ou concorrentemente tanto tarefas corriqueiras como a exibição de vídeos e jogos até cálculos complexos da ciência e da engenharia. Tal popularização propiciou a retomada de problemas tradicionais, tais como o Método dos Elementos Finitos (FEM), a fim de adequa-los aos novos paradigmas e arquiteturas dos sistemas computacionais (IWASHITA et al., 2017; HE et al., 2016; ANZT et al., 2016). De acordo com Kiss et al. (2012), a conformidade entre o problema a ser resolvido e a estrutura do ambiente de execução é capaz de ampliar a performance e reduzir a energia dispendida no processamento. Desta forma, este trabalho se propõe a verificar tal afirmação por meio da aplicação e comparação de estratégias de paralelismo em um sistema com múltiplos núcleos de processamento. Podem ser citados como processadores *multicore* as linhas Core e Xeon da Intel (INTEL, 2017b), Opteron e Ryzen da AMD (AMD, 2017) e a linha Power da IBM (IBM, 2016). Os dispositivos *manycore* por sua vez, têm como principal representante as unidades de processamento gráfico (GPU) da qual fazem parte as placas GeForce, Quadro e Tesla da NVIDIA (NVIDIA, 2017) e Radeon e FirePro da AMD (AMD, 2017).

O FEM é uma técnica numérica utilizada para a solução de problemas de valor de contorno (PVC) modelados por equações diferenciais e consiste na discretização do domínio de análise e em seguida na solução do sistema linear esparso resultante. A fim de se resolver tal sistema em ambientes paralelos, pode ser recorrer aos Métodos do subespaço de Krylov, tais como o dos Gradientes Conjugados (CG) e o do Resíduo Mínimo Generalizado (GMRes), os quais possuem a vantagem da natureza iterativa (menor propagação de erro) bem como a capacidade de fornecer a solução exata ou com a precisão desejada (FAIRES, 2008). Tais características tornam estes métodos competitivos para a solução de sistemas lineares em arquiteturas paralelas (ANZT et al., 2016). Alguns trabalhos correlatos que utilizam a família CG são apresentados por Yao et al. (2015), Ahamed e Magoulès (2016) e Iwashita et al. (2017).

A fim de manter a conformidade com os princípios de vetorização e indexação direta das arquiteturas modernas (HENNESSY; PATTERSON, 2011) é adotado neste trabalho a abordagem elemento a elemento (EBE). Esta técnica baseia-se no fato de que a matriz de coeficientes do sistema de elementos finitos é caracterizada como uma função parcialmente separável (DAYDE; L'EXCELLENT; GOULD, 1995), resultante da soma da contribuição (matriz de massa) de cada elemento. Assim sendo, as operações da etapa de solução podem ser realizadas sem a necessidade de se montar a matriz global de coeficientes. Adicionalmente

tem-se a vantagem de que elementos não adjacentes podem ser processados simultaneamente (WATHEN, 1989).

1.1 Justificativa

A priorização da vazão no processamento em relação ao aumento da taxa de comutação (ou taxa de *clock*) é uma estratégia que se tornou popular e que não se limita ao cenários dos *clusters* e *grids* mas alcança inclusive os dispositivos móveis. Patterson e Hennessy (2008) coloca que no passado, os programadores contavam com melhorias de hardware para aumentar o desempenho, sem a necessidade de adaptação de código, mas que devido à barreira de potência, induzida pela lei de Moore, surge a necessidade de se pensar em termos de paralelismo e vazão a fim de de se aproveitar de forma eficiente os recursos disponíveis. Pacheco (2011) diz que o aumento da performance computacional é o responsável por muitos avanços tecnológicos, como por exemplo a decodificação do genoma humano, imagens médicas mais acuradas e jogos mais realísticos. Segundo ele, à medida em que este aumento ocorre, considera-se resolver uma nova gama de problemas, até então sem solução. Pensando em um futuro próximo, com os avanços da computação ubíqua que introduz temas como internet das coisas, dispositivos “usáveis”, realidade aumentada e realidade virtual, a necessidade de se aproveitar ao máximo e eficientemente todo o poder de processamento disponível se torna ainda mais evidente, uma vez que nessas tecnologias há alta demanda de processamento, pouco espaço físico para comportar um processador adequado e ainda o consumo de potência como um fator limitante (YANG; CHENG, 2015).

Segundo Guo et al. (2014), com aumento de núcleos de processamento ocorre a redução da razão memória por núcleo, o que gera uma demanda para que os algoritmos utilizem eficientemente todos os níveis de paralelismo disponíveis enquanto minimizam a movimentação de dados. Desta forma, para que seja possível trabalhar com a paralelização tendo em vistas aumentar a vazão de resultados, deve-se levar em conta estratégias que reduzam a quantidade de dados armazenados (mesmo se necessário o processamento redundante) (KISS et al., 2012) e que tirem proveito do princípio da localidade espacial (armazenamento contíguo) a fim de amenizar a percepção da latência de memória.

A escolha de um problema modelado por elementos finitos ocorreu devido a uma demanda real de otimização do tempo de execução desse tipo de modelagem, a qual é comumente empregada em simulações nos campos da ciência e engenharia (KAPUSTA et al., 2016; WU et al., 2015; IWASHITA et al., 2017). A técnica EBE prioriza a localidade e contiguidade dos dados e possibilita o acesso linear, sem a necessidade da indexação indireta, como ocorre no armazenamento de estruturas esparsas.

1.2 Problema

Este trabalho investiga o processo de paralelização do FEM nas arquiteturas com multiprocessamento simétrico (SMP) de forma a transportar para o contexto dos processadores com múltiplos núcleos a afirmação feita por Kiss et al. (2012), a qual diz que a eficiência da

execução é relacionada com a conformidade entre a estrutura do problema e a arquitetura do ambiente. Para realizar esta tarefa, são apresentados contra exemplos e uma nova solução adequada aos sistemas multiprocessados. Adicionalmente é feita uma análise quantitativa do desempenho do algoritmo do EBE-FEM utilizando-se diferentes níveis de paralelismo.

O problema *benchmark* a ser resolvido refere-se à equação de Laplace originada do cálculo da distribuição de potencial e do campo elétrico de um capacitor de placas paralelas. A escolha deste problema se deve à simplicidade de sua modelagem e ao seu comportamento já explorado em livros de eletromagnetismo e circuitos elétricos, o que respalda a corretude dos resultados encontrados. A complexidade computacional exigida é obtida pelo refinamento sucessivo da malha.

A equação de Laplace surge na eletrostática por meio das equações de Maxwell correspondentes às leis de Faraday e de Gauss mostradas na forma diferencial pela equação 1.1 (VOLAKIS; CHATTERJEE; KEMPEL, 1998), nas quais $\mathbf{E}$ é a intensidade do campo elétrico em volts por metro (V/m), ϵ é a permissividade dielétrica do meio em farads por metro (F/m), ρ_v é a densidade volumétrica de cargas em coulombs por metro cúbico (C/m^3). Os operadores ∇ assumem respectivamente os papeis de rotacional do campo elétrico $\mathbf{E}$ e de divergente do fluxo elétrico $D = \epsilon E$.

$$\begin{cases} \nabla \times \mathbf{E} = 0 \\ \nabla \cdot (\epsilon \mathbf{E}) = \rho_v \end{cases} \tag{1.1}$$

O campo $\mathbf{E}$ pode ser representado em função do potencial elétrico V, como pode ser visto na equação 1.2. Devido à conservatividade de $\mathbf{E}$ ($\nabla \times \mathbf{E} = 0$), é possível se determinar a sua distribuição (grandeza vetorial) à partir de um escalar, que é o potencial elétrico V (SADIKU, 2004). Desta forma, sendo V estabelecido sobre uma parte $\partial\Omega$ de uma região Ω pode se definir o comportamento do campo elétrico sobre toda esta região e consequentemente a distribuição V sobre toda ela.

$$\mathbf{E} = -\nabla V \tag{1.2}$$

Ao substituir a equação 1.2 na equação do divergente do campo em 1.1 obtém-se em 1.3 uma equação diferencial de segunda ordem, conhecida como equação de Poisson (VOLAKIS; CHATTERJEE; KEMPEL, 1998). Se a permissividade ϵ é constante e a densidade de cargas é nula obtém-se a equação de Laplace apresentada em 1.4. O exemplo 10.4 de Boylestad (2011) que utiliza a equação de Laplace para a determinação do campo elétrico foi utilizado para a validação do algoritmo implementado neste trabalho.

$$\nabla \cdot (\epsilon \nabla V) = -\rho_v \tag{1.3}$$

$$\begin{aligned} \nabla \cdot \nabla V &= -\frac{\rho_v}{\epsilon} \\ \Delta V &= 0 \end{aligned} \tag{1.4}$$

1.3 Objetivos

De forma a se estabelecer um roteiro para este trabalho, foram levantados os objetivos a seguir, cujo atendimento é relatado ao longo do texto.

1.3.1 Objetivos Gerais

1. Demonstrar o processo de mudança do algoritmo sequencial para uma versão paralela do método do elementos finitos, por meio da decomposição de dados (coloração) e decomposição de domínio;
2. Investigar a viabilidade da técnica EBE aplicada ao método CG como alternativa do solucionador MATLAB® para PVCs em termos do tempo de execução, acurácia e economia de potência;
3. Apresentar uma relação de resultados experimentais obtidos a partir do uso de diferentes níveis de paralelismo.

1.3.2 Objetivos Específicos

1. Reproduzir a estratégia de Kiss et al. (2012) em um computador *multicore* utilizando o problema *benchmark* deste trabalho e o método CG;
2. Comparar o impacto em performance do CG com e sem armazenamento das matrizes dos elementos;
3. Avaliar quantitativamente, segundo as métricas propostas (tempo de execução, uso da CPU, *speedup* e escalabilidade) o desempenho do processo de solução do EBE-FEM paralelizado;
4. Desenvolver um material auto-contido e de fácil compreensão que sirva de introdução para o estudo do FEM e dos níveis de paralelismo do hardware para alunos da graduação;
5. Relacionar as métricas e formulações apropriadas para a avaliação de performance.

1.4 Resultados Esperados

Espera-se com este trabalho iniciar no CEFET-MG uma nova linha de pesquisa a ser continuada nos trabalhos futuros, voltada para a análise numérica de problemas de valor de contorno da engenharia. Deseja-se criar a conscientização da importância de se gerar um pensamento voltado para o paralelismo e simultaneidade nos primeiros anos do curso, visto que esta é a realidade da computação moderna e que tal prática geralmente é desafiadora, quando se está habituado ao pensamento sequencial. De forma similar, espera-se o incentivo e a adoção, por parte da universidade, do uso de paradigmas e linguagens paralelas/concorrentes, a fim de diversificar o currículo dos graduandos e despertar o pensamento crítico sobre como utilizar eficientemente os recursos disponíveis na solução de um problema.

1.5 Estrutura da monografia

O restante deste texto está estruturado em quatro capítulos e apêndices, ordenados pelo momento em que foram concluídos dentro do ciclo de vida desta pesquisa, a saber:

- O capítulo 2 apresenta os trabalhos correlatos em ordem cronológica, os quais abordam sobretudo o processo de paralelização do EBE-FEM ao longo dos anos, desde sua utilização em computadores vetoriais na década de 10 às GPUs nos anos 2000.
- A fim de se produzir um texto auto contido, o capítulo 3 relaciona os conceitos necessários para a compreensão dos materiais e métodos utilizados bem como para a interpretação resultados obtidos. O capítulo é organizado em 4 seções que abordam as principais áreas de conhecimento envolvidas no trabalho, a saber: Programação paralela, Método dos Elementos Finitos, método do Gradiente Conjugado e estratégias de aceleração da convergência.
- O capítulo 4 apresenta sequência de passos tomada para a obtenção dos resultados. Inicialmente é feita a modelagem e validação do problema no ambiente do MATLAB® com o uso de coloração. A partir dos resultados obtidos é aplicada a estratégia decomposição de domínio e o uso de *threads* em C/C++. Por fim, são realizadas otimizações no código implementado por meio do paralelismo em nível de dados e instrução e na estrutura do problema, por meio do precondicionamento de Jacobi.
- Os resultados obtidos, as considerações sobre o trabalho e os trabalhos futuros são apresentados no capítulo 5.
- O apêndice A apresenta a solução numérica passo a passo de um exemplo do FEM em $2D$.

2 Estado da arte

“Se vi mais longe
foi por estar de pé sobre ombros de gigantes”.
Isaac Newton

Segundo Kiss et al. (2012) o processamento paralelo de um problema modelado pelo FEM pode ser feito a partir da decomposição do domínio do problema, tanto por meio do particionamento da malha quanto por meio da estruturação apropriada da matriz de coeficientes. Os trabalhos de Boehmer et al. (2011) e Ahamed e Magoulès (2016) apresentam casos com a solução paralela do FEM por meio do particionamento da malha. No primeiro, o processamento é executado comparativamente nas arquiteturas de memória compartilhada e distribuída com o uso da API OpenMP e da biblioteca MPI respectivamente. No segundo, a comparação é feita entre as linguagens CUDA e OpenCL executadas em GPU.

A técnica EBE é uma forma de decomposição da matriz de coeficientes e é utilizada neste trabalho. Por meio desta, as operações são realizadas na matriz de cada elemento, sem que seja necessária a montagem do sistema global. Devido ao seu desempenho, precisão, economia de memória e à possibilidade de processamento paralelo (LEVIT, 1987; HU; QUIGLEY; CHAN, 2008; KISS et al., 2012) a adoção da abordagem EBE tem sido recorrente à medida em que surgem novas tecnologias que demandam alta performance.

Carey et al. (1988) apresenta uma implementação do Gradiente Biconjugado (BiCG) elemento a elemento (EBE-BiCG) para solucionar um sistema de elementos finitos. Segundo ele, o surgimento de novas arquiteturas, tais como os processadores vetoriais, paralelos e estações de trabalho microprocessadas foram responsáveis por se repensar o algoritmo original do FEM. A execução vetorial foi realizada no computador CRAY-XMP e foi 8 vezes mais rápida em relação ao processamento sequencial e apresentou *speedup* variando de 4.25 à 6.5. A execução paralela foi feita no ALLIANT-FX/8, um mini supercomputador com 8 CPUs. Foi adotado um esquema de ordenação de nós a fim de se evitar condições de corrida no acesso às variáveis globais. Com a utilização de todas as unidade de processamento, o *speedup* foi em torno de 7 em relação ao uso de uma única CPU. Foi observado que a multiplicação de matriz por vetor com a técnica EBE em malhas não estruturadas pode ser de difícil paralelização.

O trabalho de Dayde, L’Excellent e Gould (1995) faz uma análise comparativa entre 5 precondicionadores baseados na matriz de massa de cada elemento para o algoritmo CG, a saber: *Element matrix factorization* (EMF), *finite element preconditioner* (FEP), *one-pass element-by-element preconditioner* (EBE), *two-pass element-by-element preconditioner* (EBE2) e Gauss-Seidel *element-by-element preconditioner* (GS-EBE). Como os autores enfatizam, uma vez que se têm problemas de larga escala mas parcialmente separáveis, torna-se necessário explorar diferentes alternativas a fim de se aproveitar as vantagens oferecidas por sua estrutura. A abordagem EBE se mostrou a melhor opção entre suas concorrentes. EMF e FEP requerem a montagem parcial do sistema, o que agrega maior custo de processa-

mento. EBE2 e GS-EBE não apresentaram boas aproximações para elementos com pouca vizinhança. A fim de se realizar a paralelização, é sugerida a coloração da malha de forma que elementos sem nós em comum seja executados paralelamente. O termo EBE pode ter duas interpretações: Estratégia de solução do sistema sem montagem da matriz global (FEM desmontado) (SAAD, 2003) ou técnica de precondicionamento proposta por Hughes, Levit e Winget (1983) para a aceleração da solução do FEM desmontado. Na maior parte da literatura encontrada, a origem do EBE tanto como uso do sistema desmontado quanto como precondicionador é atribuída indistintivamente à Hughes, Levit e Winget (1983).

Uma implementação do CG para solução de um modelo EBE-FEM em FPGA (*Field Programmable Gate Array*) é proposta por Hu, Quigley e Chan (2008). Neste trabalho as operações sobre as matrizes dos elementos foram realizadas por meio da configuração de um circuito lógico no chip $4VLX160$ da Xilinx. O processamento sequencial foi feito em um PC 2.01 GHz Athlon 64 com as devidas otimizações na compilação. Graças à implementação diretamente em hardware foi alcançado um *speedup* máximo igual a 40. Estes autores apresentam uma abordagem de *gather-scatter* (juntar e espalhar) que possibilita a multiplicação de matriz por vetor em malhas não estruturadas.

O surgimento da linguagem CUDA em 2006 e a popularização das GPGPU possibilitaram que a técnica de EBE pudesse ser revisitada e aplicada nas arquiteturas SIMD (*Single Instructions Multiple Data*) modernas. Kiss et al. (2012) apresenta uma nova abordagem da técnica EBE na qual o processamento redundante é priorizado em detrimento do acesso à memória. Com essa estratégia ele soluciona um modelo de elementos finitos por meio do BiCG com precondicionador de Jacobi tendo armazenadas apenas as informações de topologia da malha. A coloração é utilizada para tornar o processamento altamente localizado, sem a necessidade de comunicação entre as *threads*. De acordo com este trabalho, tal abordagem é mais adequada para o processamento em GPU, cuja arquitetura embora seja massivamente paralela, apresenta um gargalo em latência e capacidade de memória. A minimização do trânsito de dados reduz o consumo de energia e maximiza o potencial do dispositivo. Para a realização dos testes foram utilizados o processador quad-core Xeon $X3440$ da Intel e a placa $GTX\ 590$ da NVIDIA, contendo 2 GPUs. A execução com aceleração em GPU consumiu 20 vezes menos memória e foi 10 vezes mais rápida que a execução unicamente em CPU.

O trabalho de Wu et al. (2015) também apresenta o método BiCG elemento a elemento com precondicionador de Jacobi implementado em GPU. Foi utilizado o processador Xeon $E5-2696v2$ da Intel e a placa Tesla $K20c$ da NVIDIA. Nos dois refinamentos de malha testados foi alcançado um *speedup* de 4.63 e como ressalta o autor, o resultado obtido se torna ainda mais efetivo a medida em que o número de elementos aumenta. No trabalho mais recente dos mesmos autores (YAN et al., 2017) é apresentado o uso do precondicionador de Gauss-Seidel para a abordagem EBE, o qual apresentou melhores resultados para o número de iterações e tempo de execução em relação ao precondicionamento de Jacobi, mantendo a mesma precisão.

Outros trabalhos importantes para esta monografia são os de Xu, Yin e Mao (2005), Yao et al. (2015) e Chou e Chen (2016). O primeiro apresenta a implementação do EBE-FEM

solucionado com CG e uma técnica de atribuição das condições de contorno diferente da utilizada por Wu et al. (2015). O segundo apresenta a comparação da performance da implementação do BiCG estabilizado (BiCGStab) em CPU e GPU, utilizando-se diferentes formas de armazenamento de matriz esparsa e diferentes ferramentas de resolução de sistema linear em GPU, a saber: CUSPARSE, CUSP e CULA Sparse. O último trabalho apresenta a comparação entre tecnologias de processamento paralelo (Cuda C, Cuda Fortran, MPI e OpenMP) na resolução de dois problemas *benchmark*.

3 Fundamentação Teórica

“Ler fornece ao espírito materiais para o conhecimento,
mas só o pensar faz nosso o que lemos”.
John Locke

3.1 Programação paralela

A *lei de Moore* é uma estimativa do crescimento do número de transistores por chip apresentada pelo co-fundador da Intel, Gordon Earl Moore, no ano de 1965. Segundo esta, a quantidade de transistores em um chip tende a dobrar a cada 18 meses (HENNESSY; PATTERSON, 2011). De acordo com Tanenbaum e Austin (2012) esta proposição deu início à popularização dos computadores por meio de um círculo virtuoso no qual progressos na tecnologia levam a melhores produtos e a preços mais baixos, os quais levam a novas aplicações, que por sua vez implicam em novos mercados os quais geram concorrência, a qual cria uma demanda por melhores tecnologias.

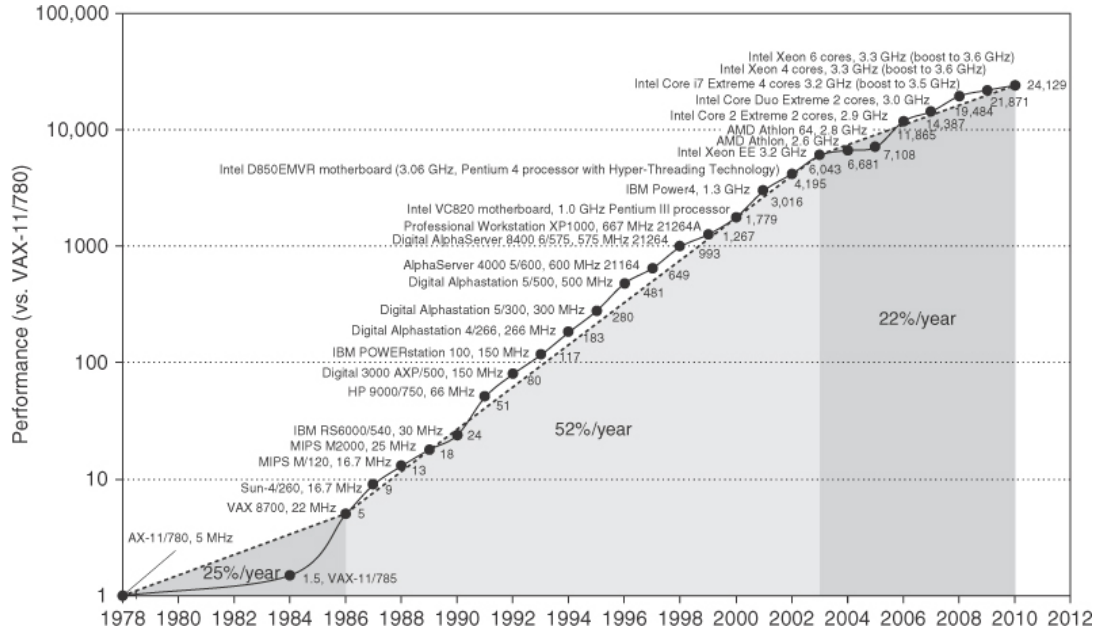
De acordo com Patterson e Hennessy (2008), o desempenho de um computador é determinado pela contagem de instruções, tempo de ciclo de *clock* e ciclos de *clock* por instrução. Desta forma, o aumento na frequência de comutação (ou taxa de *clock*) implica em uma máquina mais rápida (TANENBAUM; AUSTIN, 2012), uma vez que ocorre o aumento da velocidade de execução de cada instrução e assim a diminuição do tempo de resposta (ou latência) de um programa. As figuras 1 e 2 mostram respectivamente o crescimento do numero de transistores e a correspondente taxa de *clock* nos processadores.

Embora estas melhorias tenham contribuído para o aumento da performance ao longo dos anos, elas também levaram à então conhecida como *barreira da potência*, a qual representa o limite da viabilidade na dissipação de calor (PATTERSON; HENNESSY, 2008) no chip. A equação 3.1 (HENNESSY; PATTERSON, 2011) apresenta a relação das grandezas que impactam na potência dinâmica dissipada em um transistor CMOS (*Complementary Metal Oxide Semiconductor*). A carga capacitiva está associada à quantidade de transistores e portanto à lei de Moore. A frequência de chaveamento é dada em função da taxa de *clock* e a voltagem é a tensão elétrica necessária para abrir e fechar os contatos dos transistores. De acordo com (PATTERSON; HENNESSY, 2008) nos últimos 20 anos a voltagem foi diminuída de $5V$ para menos de $1V$, no entanto esta diminuição chegou à um limite que compromete o funcionamento adequado dos transistores, causando vazamento de tensão quando abertos.

$$Potência = \frac{Carga\ capacitiva}{2} \times Voltagem^2 \times Frequência\ de\ chaveamento \tag{3.1}$$

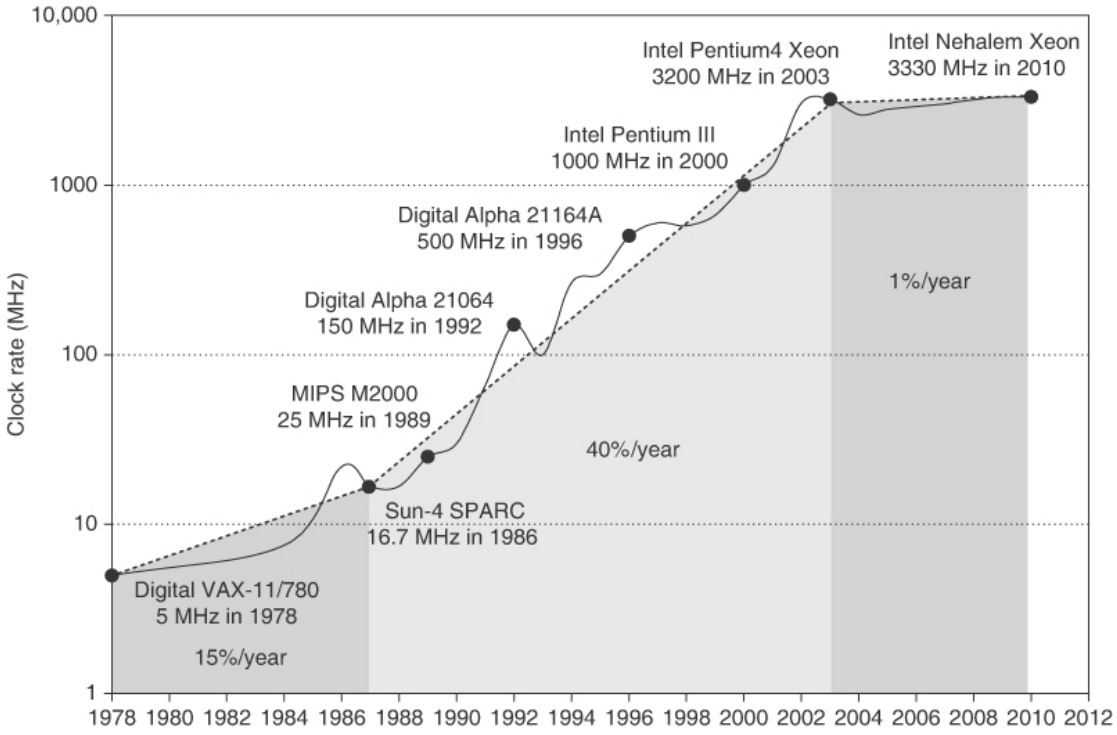
Como a barreira da potência impôs uma relativa estagnação na taxa de *clock*, uma nova estratégia de projeto tem sido adotada a fim de dar continuidade ao aumento da perfor-

Figura 1 – Crescimento do número de transistores por chip ao longo dos anos.



Fonte: (PATTERSON; HENNESSY, 2008)

Figura 2 – Crescimento da taxa de *clock* ao longo dos anos



Fonte: (HENNESSY; PATTERSON, 2011)

mance. Tal estratégia tira proveito da densidade de transistores orientada pela lei de Moore, priorizando o aumento da vazão total (mais resultados em menos tempo) ao invés diminuir o tempo de resposta de cada instrução (PACHECO, 2011). Enquanto nas gerações passadas

de processadores os níveis de paralelismo do hardware eram abstraídos para o programador, existe hoje a necessidade de se programar de forma explicitamente paralela a fim de utilizar o máximo dos recursos disponíveis e assim amplificar a performance (PATTERSON; HENNESSY, 2008). As subseções seguintes apresentam os diferentes níveis de paralelismo explorados neste trabalho.

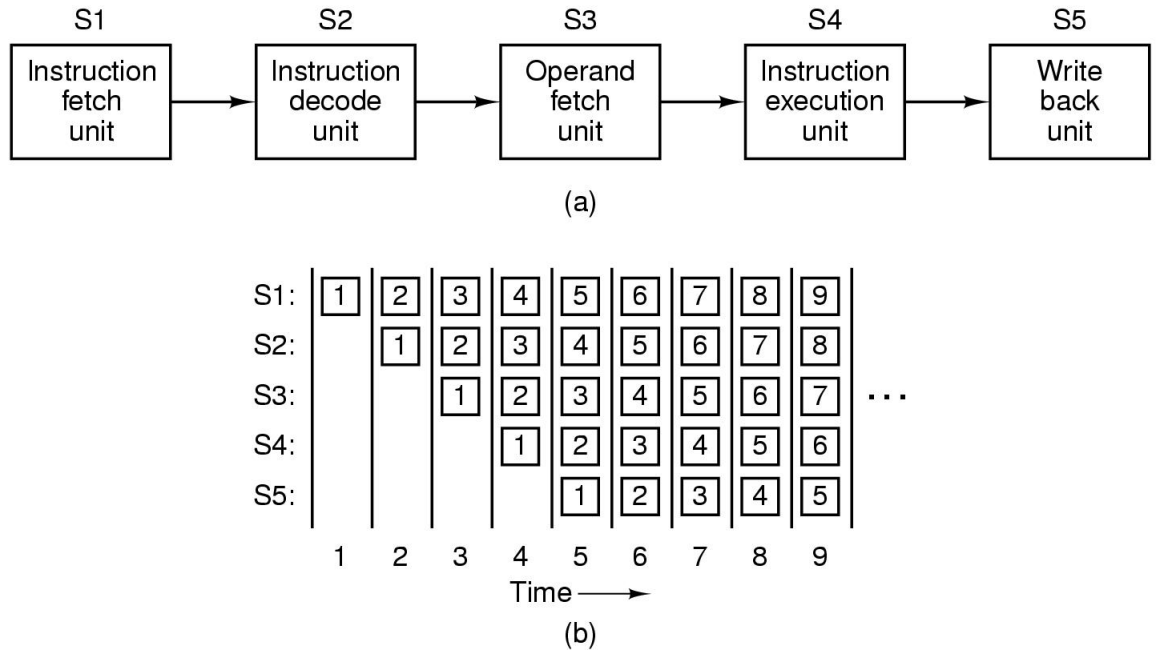
3.1.1 Paralelismo em nível de instrução

O paralelismo em nível de instrução (ILP) é a estratégia de mais baixo nível para se melhorar a vazão de um sistema (TANENBAUM; AUSTIN, 2012). Esta técnica consiste na utilização simultânea de diferentes unidades funcionais do caminho de dados a fim de se obter múltiplas instruções executadas a cada ciclo de *clock*. Duas abordagens se destacam em ILP e são tratadas nos dois tópicos a seguir: *Pipeline* de dados e múltiplo despacho (PACHECO, 2011).

3.1.1.1 *Pipeline* de dados

Utilizado na maioria dos processadores desde 1985, o *pipeline* ou canalização, possibilita que múltiplas instruções coexistam em diferentes estágios, os quais podem ser: busca de instrução, decodificação da instrução, busca de operandos, execução e gravação (HENNESSY; PATTERSON, 2011; TANENBAUM; AUSTIN, 2012). A figura 3 apresenta a ilustração do funcionamento de um *pipeline* de 5 estágios, o qual se assemelha a uma linha de produção fordista, cujas funções e tempo de sincronização (*timing*) são bem definidos (PACHECO, 2011).

Figura 3 – Percurso de instruções em um *pipeline* de 5 estágios. A partir do 5^o ciclo de *clock* o *pipeline* está cheio, executanto estágios de 5 diferentes instruções paralelamente



Fonte: (TANENBAUM; AUSTIN, 2012)

Para que o paralelismo em nível de instruções possa ser explorado é necessário determinar quais instruções são independentes entre si, a fim de que possam ser processadas no mesmo ciclo de *clock*. As dependências entre as instruções podem ser dos tipo: dependência de dados, de nomes ou de controle (HENNESSY; PATTERSON, 2011). Quando uma dependência é verificada ao longo do *pipeline* são criados seus respectivos *hazards* (ou acasos), os quais são estratégias que mantém a corretude do algoritmo e a sequencialidade do código em detrimento da vazão (PATTERSON; HENNESSY, 2008; TANENBAUM; AUSTIN, 2012).

Em se tratando da arquitetura do processador, uma forma de aumentar o paralelismo no *pipeline* é subdividir os estágios existentes, fazendo com que mais instruções possam se sobrepor (PATTERSON; HENNESSY, 2008).

3.1.1.2 Múltiplo despacho

A estratégia de múltiplo despacho consiste na replicação das unidades funcionais de um caminho de dados, de forma a executar múltiplas instruções simultaneamente em um mesmo estágio (PACHECO, 2011). Este tipo de abordagem pode ser classificada como estática ou dinâmica, dependendo da divisão de trabalho entre o compilador e o hardware (PATTERSON; HENNESSY, 2008).

No múltiplo despacho estático a alocação das unidades funcionais é feita em tempo de compilação. Neste tipo de operação, as intruções a serem despachadas em um ciclo de *clock* são agrupadas pacotes com o auxílio do compilador, o que facilita o processamento pelo hardware (TANENBAUM; AUSTIN, 2012). A esta técnica dá-se o nome de VLIW (*Very Long Instruction Word*), uma vez que o pacote a ser processado pode ser visto como uma única instrução com mais de um campo de *opcode* (PATTERSON; HENNESSY, 2008).

Processadores com múltiplo despacho dinâmico recebem o nome de *superescalares*, e permite a seleção em tempo de execução das instruções a serem processadas em paralelo. Este tipo de implementação pode executar as funções fora de ordem (escalonamento dinâmico), embora seu carregamento e o *commit* dos resultados sejam feitos em ordem (PACHECO, 2011). Como afirma (PATTERSON; HENNESSY, 2008), mesmo o compilador sendo importante para a identificação e organização das dependências, apenas o escalonamento dinâmico pode ocultar, por exemplo, alguns tipos de falha de cache.

3.1.1.3 Programação voltada para ILP

Embora o paralelismo em nível de instrução possa ser considerado uma extensão da arquitetura de Von Neumann (PACHECO, 2011) e seja em grande parte realizado em tempo de compilação ou de execução sem a necessidade de adaptações de código (transparente ao programador)(PATTERSON; HENNESSY, 2008), algumas práticas podem ser adotadas para facilitar as especulações do compilador e do hardware.

O desdobramento de *loop* consiste na replicação do corpo da iteração de forma a garantir a independência entre os blocos bem como reduzir a sobrecarga incluída pelo *loop*, tais como o incremento do contador e a verificação do limite de iteração (HENNESSY; PATTERSON, 2011). Um exemplo é apresentado no algoritmo 1. Iteradores apresentam considerável

tempo de execução e se tornam mais performáticos ao serem desdobrados, embora haja o aumento no tamanho do código (ARMKEIL, 2015). O compilador g++ realiza automaticamente o desdobramento por meio da opção `-funroll-loops` ao se utilizar as *flags* `-O3` e `-Otime`. Embora possa ser feito o desdobramento manual, este pode comprometer outras otimizações realizadas pelo compilador (ARMKEIL, 2015). Como o objetivo do trabalho é investigar estratégias de paralelização independentes da linguagem e do compilador, nenhuma *flag* de otimização é utilizada e portanto o desdobramento é feito diretamente no código.

Algoritmo 1 Soma de vetores sem e com desdobramento de *loop*

```
1: for i = 1,2,3,...,100 do
2:     r[i] = x[i] + y[i]
3: end for
```

```
1: for i = 1,5,9,...,100 do
2:     r[i] = x[i] + y[i]
3:     r[i+1] = x[i+1] + y[i+1]
4:     r[i+2] = x[i+2] + y[i+2]
5:     r[i+3] = x[i+3] + y[i+3]
6: end for
```

Um outro fator que afeta o ILP e pode ser levado em consideração durante a codificação/compilação é a reutilização de variáveis, a qual pode levar a uma dependência de nomes. Este tipo de dependência é uma pseudo ou anti-dependência e gera uma ordenação forçada, sem que haja uma relação real entre os dados, mas apenas o reaproveitamento de variáveis (PATTERSON; HENNESSY, 2008).

3.1.2 Paralelismo em nível de *threads*

Uma vez que existem poucas possibilidades de se explorar o paralelismo do hardware por meio de ILP tende-se a se recorrer à um nível mais alto de paralelismo, o qual é realizado por *threads* e é denominado TLP (PACHECO, 2011). Além disso, as ILP conseguem ocultar apenas alguns tipos de espera, não sendo eficazes para bloqueios mais demorados tais como as falhas de cache que levam ao acesso à memórias mais altas, como por exemplo $L3$ ou memória principal (HENNESSY; PATTERSON, 2011).

A técnica de *multithreading* possibilita que várias *threads* (ou linhas de execução) sejam executadas simultaneamente em multiprocessadores e concorrentemente em um único processador, de forma a utilizar o máximo dos recursos disponíveis e a evitar a ociosidade em cada um. Para tanto são duplicados alguns recursos mais utilizados, tais como o contador de programa e o mapa de registradores, enquanto outros são mantidos compartilhados, como por exemplo as linhas de cache. Existem ainda os recursos que são divididos em partes entre as *threads*, como por exemplos as filas de um *pipeline* (TANENBAUM; AUSTIN, 2012). Esta modificação ocorre de forma que a área necessária para os recursos adicionais e o tempo de comutação entre as linhas de execução sejam mínimos(PACHECO, 2011).

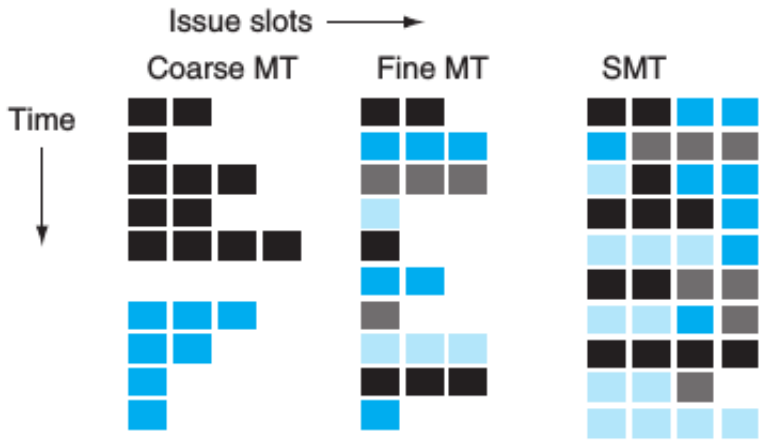
3.1.2.1 Ganulação de *multithreading*

Existem basicamente dois tipos de *multithreading*, o de granulação fina e o de granulação grossa. O conceito de granulação (ou granularidade) está relacionado à proporção de um problema que pode ser executada em paralelo em um dado período. Dessa forma, o ILP pos-

sui granulação mais fina que o TLP, uma vez que no primeiro, o paralelismo ocorre instrução a instrução e no segundo ele ocorre entre trechos maiores de um programa (PACHECO, 2011).

Um processador com *multithreading* de granulação fina comuta entre as linhas de execução a cada instrução, de forma a ocultar as esperas. Segundo Tanenbaum e Austin (2012), se houverem no mínimo k *threads* disponíveis em um *pipeline* de granulação fina com k estágios, o processador nunca ficará ocioso. Uma desvantagem dessa abordagem é que nem sempre existem tantas *threads* disponíveis quanto os estágios do *pipeline* e além disso, se mesmo que uma *thread* esteja apta a executar uma sequência de instruções ela deve se sujeitar à política de alternância (PACHECO, 2011). O processador com granulação grossa surgiu como uma alternativa para o de granulação fina e realiza a comutação apenas em operações mais caras, como o acesso à cache nível 3 ou memória principal. Assim sendo, esta abordagem prioriza a conclusão de cada *thread* individualmente e demanda um número menor de *threads* para manter o processador ocupado. Contudo, como a comutação ocorre apenas após a comprovação de uma espera onerosa, perde-se sempre um ciclo de *clock* na troca de contexto (TANENBAUM; AUSTIN, 2012). Uma terceira alternativa e mais comumente utilizada (HENNESSY; PATTERSON, 2011) é o *multithreading* simultâneo (SMT), o qual decorre do uso de um processador superescalar, ou seja, a combinação de ILP e TLP (PATTERSON; HENNESSY, 2008). Por meio dessa estratégia, uma linha de execução pode processar múltiplas instruções de múltiplas *threads*.

Figura 4 – Comparação entre os tipo de *multithreading*: Granulação grossa (*coarse*), fina (*fine*) e SMT



Fonte: (PATTERSON; HENNESSY, 2008)

3.1.2.2 Programação *multithread*

De acordo com Patterson e Hennessy (2008), a dificuldade em relação ao paralelismo ocorre principalmente porque a maioria dos programas são pensados sequencialmente, ainda que maior parte das máquinas de uso geral da atualidade (linha Intel Core) ofereçam suporte a *multithreading*. Embora tenham sido feitos esforços para a a paralelização automática de programas seriais, os ganhos ainda são limitados(PACHECO, 2011).

Com o objetivo de manter uma linha de raciocínio sequencial e ainda assim criar programas paralelos, foram desenvolvidas APIs que abstraem em diferentes níveis o paralelismo do hardware. As mais comuns na linguagem C/C++ são a *OpenMP* e as *PThreads* (BARNEY, 2017b; BARNEY, 2017a). A *OpenMP*, criada em 1997, é uma API portável e escalável e tem como principais objetivos ser fácil de se utilizar e fornecer recursos de paralelização padronizados para os processadores de diferentes fabricantes. Ela possibilita uma forma de programação *multithreading* de alto nível, por meio de instruções de preprocessamento denominadas *diretivas* ou *pragmas*. Estas instruções têm o papel de informar ao compilador quais partes de um programa são paralelizáveis (BARNEY, 2017a). Por outro lado as *PThreads*, foram incluídas no padrão POSIX dos sistemas Unix no ano de 1995 e possibilitam a manutenção de linhas de execução em um nível mais baixo. Desconsiderando o comportamento de não determinismo intrínseco do hardware (PACHECO, 2011), é possível se obter o controle do comportamento de cada *thread*, ficando a cargo do programador a gestão do ciclo de vida de cada uma, como por exemplo sua criação, comunicação, sincronização e término(BARNEY, 2017b).

À medida em que os sistemas *multithreading* e multiprocessadores se popularizaram, muitas linguagens até então sequenciais passaram a implementar nativamente o suporte à programação concorrente. Outras linguagens mais novas já nasceram com este suporte. A lista a seguir contém a descrição de algumas linguagens com suporte a *multithreading*.

- **C++**: Originada no início da década de 80 com o objetivo de implementar a orientação a objetos na linguagem C passou a ter suporte nativo a *multithreading* em 2011, à partir da especificação *C++11*. É compilada, fortemente tipada e multiparadigma. Possibilita a criação *threads* a partir de uma classe `std::thread` e apresenta recursos nativos para a manutenção do ciclo de vida, como por exemplo `join`, `yield`, `sleep` e `swap` bem como recursos de sincronização como `mutex` e variáveis de condição (CPLUSPLUS, 2017; BANCILA, 2013) .

- **Java**: Linguagem de propósito geral, orientada a objetos, compilada para *bytecodes* que rodam na JVM (*Java Virtual Machine*) e a partir de sua versão 5 passou a fornecer a API de concorrência de alto nível `java.util.concurrent`. Além de disponibilizar os recursos básicos de manutenção do ciclo de vida de uma *thread*, a linguagem Java disponibiliza uma interface `Runnable` que pode ser implementada por classes cujos objetos executam em *threads*. Um outro recurso da linguagem são os *thread pools* o qual minimiza a sobrecarga gerada pela alocação e desalocação de *threads* (ORACLE, 2017b; ORACLE, 2017a).

- **Rust**: Linguagem lançada em 2012 nos laboratórios da Mozilla em a parceria da Samsung para a criação de uma nova *engine* para o Firefox, chamada *Servo*. Tem como objetivo solucionar alguns problemas enfrentados pelo C++, como por exemplo a reprodução de erros causados por *multithreading*, dada a característica de não determinismo do hardware (PACHECO, 2011). Ela é uma linguagem de sistemas, multiparadigma, que previne falhas de segmentação e garante a segurança entre *threads*. As linhas de execução em Rust são livres de condição de corrida e a comunicação entre elas pode ser

feita pelo uso de `channels` os quais possibilitam a comunicação por troca de mensagem, como ocorre na linguagem Go e em sistemas de memória distribuída por meio do MPI (Message Passing Interface). A sincronização por meio de `mutex` é feita sobre os dados e não sobre trechos de código e ao invés de acesso à memória compartilhada, ocorre a transferência de posse (*ownership*) das variáveis. (DEVELOPERS, 2011; BLANDY, 2015).

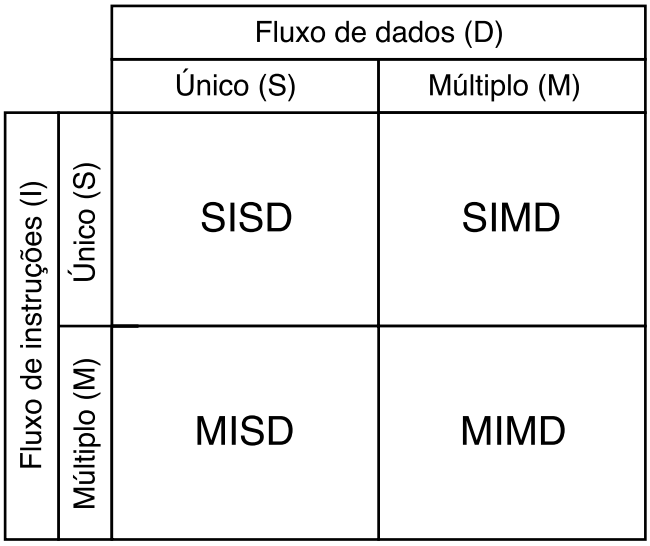
- **Elixir**: Esta linguagem surgida em 2012 e criada por um brasileiro, José Valim, é compilada para *bytecode* da máquiva virtual da *Erlang*, uma das mais tradicionais linguagens paralelas. Elixir é uma linguagem escalável, dinâmica e funcional voltada para a criação de aplicações distribuídas. O seu modelo de concorrência é baseado em troca de mensagens e utiliza o conceito de processos que rodam dentro da máquina virtual, os quais são mais leves que as *threads* e por isso podem existir em maior quantidade (PLATAFORMATEC, 2017; AL., 2017).

Como apresentado na lista anterior, os modelos de concorrência podem ser classificados em dois tipos: compartilhamento de estado, no qual um endereço de memória pode ser alterado por mais de uma *thread* e passagem de mensagem no qual cada *thread* detém seu estado, podendo enviá-lo para as demais ao invés de compartilha-lo.

3.1.3 Paralelismo em nível de dados

Embora nas subseções anteriores o paralelismo tenha sido explorado em termos do fluxo de instruções (instruções em si ou por blocos), é possível se explorar o paralelismo em nível de dados (DLP). A Taxonomia de Flynn proposta na década de 60 relaciona os fluxos de instruções com os fluxos de dados e pode ser vista na figura 5 (PATTERSON; HENNESSY, 2008). Uma máquina SISD apresenta um único fluxo de instruções e um único fluxo de dados e pode ser representada por um computador de Von Neumann. Não há exemplos práticos de uma máquina MISD, mas ela pode ser entendida como uma espécie de *pipeline*, com muitas instruções alterando um único dado (TANENBAUM; AUSTIN, 2012). Os computadores MIMD são os mais utilizados na atualidade e englobam os multiprocessadores e os multicomputadores, uma vez que nestes casos existem várias instruções operando em diferentes dados ao mesmo tempo. Programas que particionam um problema e executam cada parte em diferentes instâncias (*threads* ou processos) de um computador MIMD são denominados SPMD (*Single program, multiple data*) (PACHECO, 2011). Por fim, as arquiteturas SIMD podem ser representadas pelas GPUs e são capazes de aproveitar o DLP aplicando uma única instrução a um vetor de dados. De acordo com (HENNESSY; PATTERSON, 2011), o modelo SIMD apresenta algumas vantagens sobre o MIMD. A primeira delas é a eficiência energética, uma vez que uma única instrução é buscada e aplicada sobre um conjunto de dados. A segunda é que há uma grande demanda para este tipo de processamento, como por exemplo em som e imagem. Por último tem-se que a lógica da programação SIMD permanece sequencial (similar às diretivas do OpenMP) ao contrário dos SPMD que rodam em máquinas MIMD, os quais geralmente demandam a reestruturação do problema.

Figura 5 – Quadro da taxonomia de Flynn



Fonte: Elaborada pelo autor

Embora as GPUs sejam os representantes SIMD mais comuns hoje, o processamento vetorial teve início na década de 70 com os computadores vetoriais Cray(PATTERSON; HENNESSY, 2008). No ano de 1996 instruções SIMD começaram a ser incorporadas à ISA (*Instruction Set Architecture*) das arquiteturas $\times 86$ para o processamento multimídia. Em 2006 a NVIDIA lançou a CUDA (*Compute Unified Device Architecture*) sua API para programação de propósito geral em GPU. Nos tópicos a seguir serão apresentadas as extensões da ISA e a arquitetura das GPUs(HENNESSY; PATTERSON, 2011).

3.1.3.1 Extensões da ISA para multimídia

A inclusão de novas instruções à ISA ocorreu ao se perceber que as aplicações de mídia tendem a operar sobre dados curtos, como por exemplo de 8 ou 16 bits. Dessa forma, uma ULA (*Unidade Lógica e Aritmética*) com capacidade de 256 bits poderia ser dividida em 32 menores que operem simultaneamente sobre cadeias de 8 bits, ou 16 que operem sobre cadeias de 16 bits e assim sucessivamente (HENNESSY; PATTERSON, 2011). As primeiras extensões multimídia foram adicionadas em 1996 e pertencem ao conjunto MMX(Multimedia Extensions). Elas possibilitavam a execução de 8 operações de 8 bits ou 4 de 16 bits simultaneamente. Em 1999 foram incluídas as intruções da tecnologia SSE (*Streaming SIMD Extensions*) com o objetivo de substituir as MMX. Esta tecnologia passou por 3 modificações, SSE2, SSE3 e SSE4, sendo adicionadas novas instruções e otimizações em cada uma. O conjunto SEE podem atualmente realizar 4 operações de ponto flutuante com precisão simples (*float*) ou 2 de precisão dupla (*double*). No ano de 2010 foram adicionadas as instruções da extensão AVX (*Advanced Vector Extensions*) as quais, na versão atual (AVX-512), realizam 8 operações do tipo *double*, 16 do tipo *float* ou 32 do tipo *int* simultaneamente (INTEL, 2017a; REINDERS, 2017).

3.1.3.2 A arquitetura das GPUs

O desenvolvimento de um hardware dedicado para a renderização de gráficos em *2D* e *3D* se deu sobretudo pela crescente demanda das indústrias de jogos e do cinema. Embora o processamento gráfico consista basicamente em repetidas operações aritméticas sobre vértices $\{x, y, z, w\}$ e pixels $\{r, g, b, a\}$, a quantidade massiva de dados, a qual pode chegar a centenas de megabytes por frame, exige um melhoramento do hardware de forma a ocultar a latência, ampliar a largura de banda no acesso à memória e reduzir o tempo de processamento de cada frame (PATTERSON; HENNESSY, 2008). A estratégia adotada nas arquiteturas das GPUs consiste em explorar ao máximo os tipos de paralelismo disponíveis (*multithreading*, MIMD, SIMD e ILP) de forma aninhada, diretamente no nível do hardware. Essa nova abordagem é capaz de executar milhares de (*threads*) simultâneamente.

Em 2006 NVIDIA lançou sua plataforma de computação paralela, a CUDA (*Compute Unified Device Architecture*), com o objetivo de abstrair as informações relativas à hierarquia das *threads*, compartilhamento de memória e sincronização (PATTERSON; HENNESSY, 2008). Esta tecnologia possibilitou que as GPUs pudessem ser programadas para tarefas de outras áreas, além do processamento gráfico, o que ficou conhecido como programação de Propósito Geral para GPU (GPGPU) (PATTERSON; HENNESSY, 2008). O padrão de execução massiva encontrado nas *threads* CUDA se diferencia do processamento *SIMD* dos computadores vetoriais e por isso é chamado nos sistemas da NVIDIA de SIMT (*Single Instruction, Multiple Threads*) (HENNESSY; PATTERSON, 2011).

As GPUs são dispositivos (*device*) coprocessadores utilizados para acelerar o processamento de uma CPU, chamada de hospedeiro ou *host*. Dessa forma, os cálculos repetitivos sobre grandes quantidades de dados que seriam realizados iterativamente no processador são enviados para o dispositivo coprocessador, os quais os realizam simultaneamente em uma espécie de “lote”. Sistemas que operam com o auxílio de coprocessadores são chamados de heterogêneos (PATTERSON; HENNESSY, 2008).

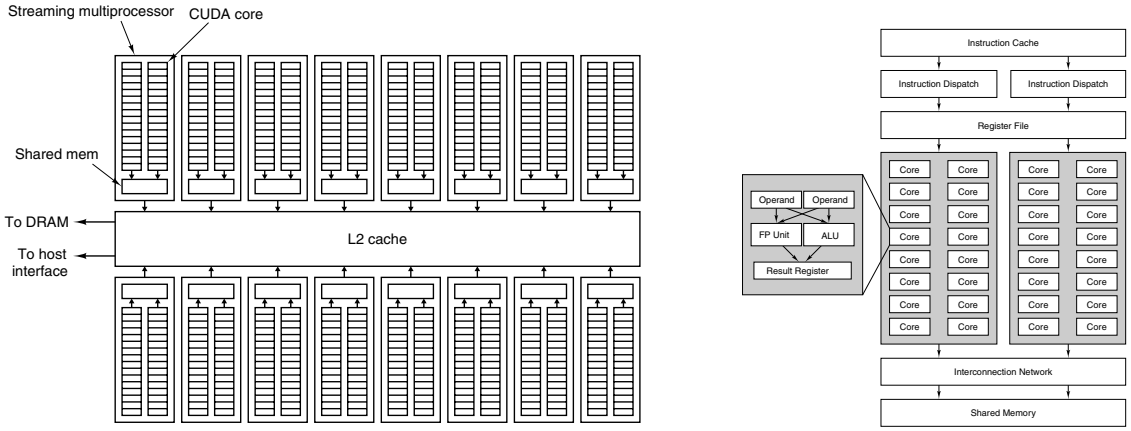
Como pode ser visto na figura figura 6, uma GPU da NVIDIA é dividida em SMs (*Streaming Multiprocessors*), que por sua vez são divididos em núcleos, os CUDA *cores*, nos quais são executadas operações aritméticas e de ponto flutuante (TANENBAUM; AUSTIN, 2012).

3.1.3.3 Programação voltada para DLP

A principal estratégia utilizada para se explorar o DLP consiste na organização adequada dos dados na memória. As técnicas de *strip mine*, redução de *stride*, *gather-scatter* e identificação de dependências em *loops* serão vistas a seguir.

A técnica de *strip mine* (ou *tiling*) consiste em dividir os dados em conjuntos ou blocos menores a fim de tirar proveito dos princípios de localidade espacial e temporal. O algoritmo 2 apresenta um exemplo de multiplicação de matriz por vetor convencional e com *strip mining*. Esta estratégia também é util para separar conjuntos de dados que possam ser calculados simultaneamente, por meio de *threads* ou instruções multimídia (HENNESSY; PATTERSON, 2011).

Figura 6 – Esquema com a relação dos SMs de uma gpu Fermi (esquerda) e detalhe de um único SM (direita)



Fonte: (TANENBAUM; AUSTIN, 2012)

Algoritmo 2 Multiplicação de matriz por vetor (ordem 100) convencional e por meio de *strip mining* (blocos de ordem 4)

```
1: for i = 1,2,...,100 do
2:     r[i] =0
3:     for j = 1,2,...,N do
4:         r[i] += a[i][j] * b[j]
5:     end for
6: end for
```

```
1: for i = 1,5,9,...,100 do
2:     r[i:i+4] =0
3:     for j = 1,5,9,...,N do
4:         for ii = i,i+1,...,i+4 do
5:             for jj = j, j+1,...,j+4 do
6:                 r[ii] += a[ii][jj] * b[jj]
7:             end for
8:         end for
9:     end for
10: end for
```

Como vetores e matrizes são armazenados de forma linearizada na memória é possível por exemplo, endereçar uma matriz `A` $2D$ tanto por meio de `A[i][j]` quanto por `A[j*t+i]`. Ao se iterar o valor `j` no segundo caso caminha-se em posições não contíguas da memória, separadas a uma distâcia de `t` elementos, chamada de passo ou *stride*. Se `t` for grande, perde-se a localidade espacial, o que leva a falhas de cache (LAWLOR, 2006). Assim sendo, deve-se organizar os dados de forma a diminuir o tamanho do passo. O algoritmo 3 mostra duas formas de armazenamento dos dados dos nós de uma malha triangular de 100 elementos. Na primeira, o arquivo é estruturado com 3 linhas e 100 colunas, sendo o *stride* da obtenção de pontos do mesmo triângulo é igual a 100. Na segunda o arquivo possui 100 linhas e 3 colunas, sendo o *stride* igual a 3.

Operações de *gather-scatter* consistem na manipulação de vetores indexados por outros vetores, como por exemplo `b[c[i]]`. Este tipo de acesso é muito mais lento que os armazenamentos não indexados, uma vez que é necessário buscar da memória a posição a ser acessada e só então buscar o valor da variável referente a essa posição. Como cada elemento possui o seu endereço individual não é possível realizar agrupamentos para se aplicar instruções SIMD (HENNESSY; PATTERSON, 2011). Para contornar este problema, utiliza-se uma operação de *gather* que reúne os dados em um vetor temporário e realiza as operações sobre

Algoritmo 3 Redução de *stride* de 100 para 3

```
1: pts[300] = le("tri.txt",3,100)
2: for i = 1,2,...,100 do
3:     p[1] = pts[0*100+i]
4:     p[2] = pts[1*100+i]
5:     p[3] = pts[2*100+i]
6:     area = calcArea(p,...)
7:     ...
8: end for
```

```
1: pts[300] = le("tri.txt",3,100)
2: for j = 1,2,...,100 do
3:     p[1] = pts[(j-1)*3+1]
4:     p[2] = pts[(j-1)*3+2]
5:     p[3] = pts[(j-1)*3+3]
6:     area = calcArea(p,...)
7:     ...
8: end for
```

eles. Após isso, aplica-se uma operação de *scatter* que "espalha" novamente os dados para suas respectivas posições.

As operações no corpo de um *loop* podem ser dependentes dos resultados de iterações anteriores. Alguns casos, como em recorrências do tipo `y[i] = y[i-1] + y[i]`, não há o que se fazer, mas em outros é possível resolver o problema de dependências transportadas pelo *loop* a fim de executá-lo paralelamente(HENNESSY; PATTERSON, 2011). Um caso comum é a operação de redução (*MapReduce*), apresentada no algoritmo 4. Embora seja gerado um *loop* adicional que acumula as somas em um outro vetor, este pode ser processado paralelamente.

Algoritmo 4 Algoritmo de *MapReduce* com e sem dependência transportada por *loop*

```
1: soma = 0
2: for i = 1,2,...,100 do
3:     soma += x[i] + y[i]
4: end for
```

```
1: soma[300] = 0
2: for i = 1,2,...,100 do
3:     soma[i] = x[i]+y[i]
4: end for
5: reduce = 0
6: for i = 1,2,...,100 do
7:     reduce += soma[i]
8: end for
```

3.1.4 Paralelismo em nível de processadores

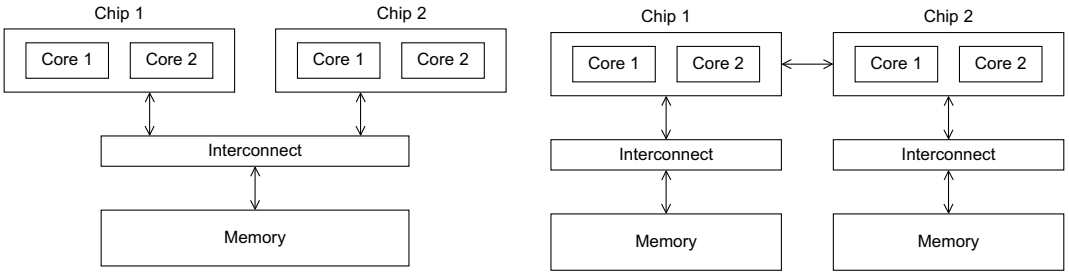
O último nível de paralelismo a ser explorado consiste na replicação de processadores, a qual pode ocorrer basicamente de três formas: em um único chip, pela associação múltiplos chips ou ainda pela interligação de computadores em rede (TANENBAUM; AUSTIN, 2012). Neste trabalho serão explorados apenas os dois primeiros tipos.

3.1.4.1 Sistemas de memória compartilhada

Os processadores interligados por rede são chamados de sistemas de memória distribuída ou multicomputadores. Nesse tipo de sistema cada CPU possui a sua própria memória e comunicação geralmente é feita por troca de mensagem (PACHECO, 2011). Em contrapartida, os sistemas de memória compartilhada, também chamados de multiprocessadores, possuem forte acoplamento entre seus nós, os quais podem ser os núcleos de um chip *multicore* ou CPUs de outros chips. Como o nome sugere, estes sistemas se comunicam principalmente pelo compartilhamento de memória (HENNESSY; PATTERSON, 2011).

Dependendo do tipo de acesso à memória compartilhada os multiprocessadores podem ser classificados em UMA (*Uniform Memory Access*) ou NUMA (*Nonuniform Memory Access*). No primeiro caso, também chamado de SMP (*Symmetric Multiprocessor*) a memória é centralizada fazendo com que os processadores levem aproximadamente o mesmo tempo para acessá-la, independente de qual o endereço requisitado. No segundo caso, também chamado de DSM(*Distributed Shared Memory*) o acesso à memória é mais rápido em alguns casos, dependendo do processador que faz o acesso e do endereço buscado (PATTERSON; HENNESSY, 2008). Embora máquinas UMA sejam mais fáceis de programar, os sistemas NUMA tem maior potencial para expansão (PACHECO, 2011). A figura 7 mostra os esquemas de comunicação de arquiteturas UMA e NUMA.

Figura 7 – Esquema de arquitetura UMA e NUMA



Fonte: (PACHECO, 2011)

A paralelização em nível de processadores pode ser explorada por meio de multiplos processos ou multiplas *threads*. Estas últimas por posuírem menor sobrecarga durante a troca de contexto (demandam menos ciclos de *clock*) (TANENBAUM; AUSTIN, 2012) são adotas neste trabalho.

3.1.5 Procedimentos e métricas para a paralelização

Uma vez definidos os intrumentos e as estratégias para a paralelização de um algoritmo deve-se definir as etapas que serão percorridas e as métricas de desempenho que serão adotadas. A metodologia de Foster é utilizada para direcionar o processo de paralelização e as métricas de *speedup*, eficiência e escalabilidade são aplicadas a fim de se realizar uma avaliação quantitativa dos resultados.

3.1.5.1 Metodologia de Foster

Este método proposto por Ian Foster em 1996 tem como objetivo abstrair inicialmente os aspectos relacionados à arquitetura e focar nas características de paralelismo do problema. Esta estratégia consiste em 4 passos listados a seguir (PACHECO, 2011).

1. *Particionamento*: Dividir o problema em tarefas menores de forma a identificar quais delas podem ser executadas em paralelo;
2. *Comunicação*: Determinar qual a comunicação necessária entre cada tarefa;

3. *Aglomeração*: Agrupar as tarefas e as comunicações em conjuntos maiores, de forma que tarefas dependentes fiquem em um mesmo conjuntos e tarefas paralelizáveis em conjuntos distintos;

4. *Mapeamento*: Atribuir os conjuntos da etapa anterior aos processos ou *threads* de forma que a comunicação seja mínima e carga de cada um balanceada.

3.1.5.2 Métricas de desempenho

O *speedup* pode ser calculado pela equação 3.2 e é uma métrica que mede o ganho de velocidade por uma aplicação paralela em relação à sua versão sequencial (ou à melhor versão sequencial existente (PATTERSON; HENNESSY, 2008)). A equação 3.3 apresenta o cálculo da eficência ou da contribuição média de cada *thread* p para o ganho de velocidade (PACHECO, 2011). A escalabilidade possibilia verificar o ganho ou perda de eficência na solução de um problema à medida em que a quantidade *threads* cresce. Se a eficiência é mantida ao se aumentar o número de *threads* sem aumentar o tamanho do problema, este é chamado de formetemente escalável. Se a eficiência é mantida ao se aumentar o tamanho do problema proporcionalmente ao aumento de *threads* este é dito fracamente escalável (PATTERSON; HENNESSY, 2008).

$$S = \frac{T_{serial}}{T_{paralelo}} \tag{3.2}$$

$$E = \frac{S}{N_{threads}} \tag{3.3}$$

3.2 Método dos elementos finitos

O FEM é uma técnica numérica para a solução de problemas de valor de contorno. Neste método, o domínio de análise é visto como uma coleção de subdomínios, chamados de elementos finitos, sobre os quais, a equação que modela o problema é aproximada por um método variacional ou de resíduos ponderados (REDDY, 2006). Essas diferentes vertentes do método surgiram devido aos esforços independentes de matemáticos, cientistas e engenheiros (ZIENKIEWICZ, 2005). Originalmente o método foi proposto para a análise de deslocamentos e elasticidade de estruturas mecânicas, mas em seguida foi estendido para solucionar problemas de outros campos da física e da engenharia (JIN, 2002; DESAI, 1972; ZIENKIEWICZ, 2005).

3.2.1 Problema de Valor de Contorno

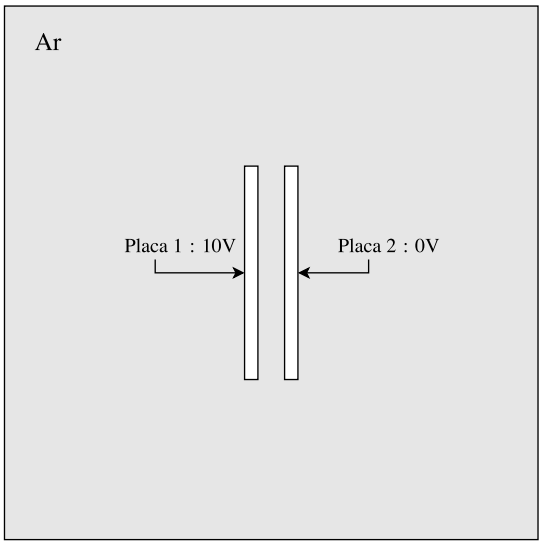
Um problema de valor inicial (PVI), pode ser definido como uma equação diferencial (ou sistema de equações) na qual são dadas as condições iniciais do fenômeno, as quais são impostas sobre a variável dependente e suas derivadas em um único instante de tempo (inicial) t_0. Um problema de valor de contorno (PVC) por sua vez, apresenta tais condições aplicadas em pontos distintos, como por exemplo em x_i e x_f. Geralmente os PVI são dados em função do

Condição	**Tipo**
$u(x_i) = \alpha$	Dirichlet
$u(x_i) = 0$	Dirichlet Homogênea
$u'(x_i) = \beta$	Neumann
$u'(x_i) = 0$	Neumann Homogênea

Tabela 1 – Exemplos de condições de contorno

tempo enquanto os PVC são estabelecidos em função do espaço (BOYCE; DIPRIMA, 2010). Na figura 8 é apresentada a geometria do problema *benchmark* com as condições de contorno (tensão) pré-definidas sobre cada uma das placas. A partir dessa informação sobre uma parte do domínio, deseja-se obter obter os valores nas demais partes.

Figura 8 – Condições de contorno do capacitor de placas paralelas.



Fonte: Elaborada pelo autor

Conforme mostra a tabela 1, as restrições aplicadas sobre a variável dependente são chamadas de *condições de Dirichlet* ou *essenciais*, enquanto as que são estabelecidas sobre as derivadas da variável dependente são conhecidas como *condições de Neumann* ou *naturais*. Além destas duas classes, existem as restrições específicas do fenômeno modelado, como por exemplo, condições de radiação ou de impedância para problemas do eletromagnetismo (JIN, 2002).

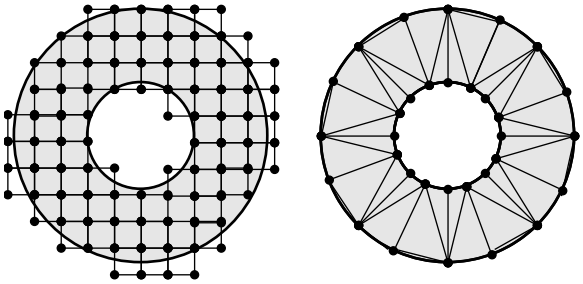
A solução analítica de um PVC pode ser obtida por meio da integração direta ou a partir da aplicação de técnicas como a separação de variáveis, expansão em séries ou pela transformada de Laplace. No entanto, a maioria dos problemas da engenharia e da ciência não são lineares e apresentam geometria ou condições de contorno complexas. Estas características fazem com que a resolução analítica de tais problemas seja impraticável, sendo necessário recorrer a métodos numéricos a fim de se obter uma solução aproximada (BOYCE; DIPRIMA, 2010; POWERS, 2006).

3.2.2 Métodos numéricos para PVCs

Anteriormente ao FEM, o Método das Diferenças Finitas (FDM) destacava-se como uma estratégia para a solução de PVC, a qual realiza a discretização do domínio por meio de uma grade regular de pontos e a aproximação de cada derivada da equação por um quociente-diferença adequado (FAIRES, 2008). Embora este método seja útil em muitos casos, se torna difícil aplicá-lo em problemas com geometria irregular ou com condições de contorno não usuais. Um exemplo pode ser visto na figura 9.

Diferentemente do FDM, como destaca Huebner et al. (2001), o FEM divide o domínio em subdomínios ao invés de pontos e representa mais fielmente a geometria do problema. Assim sendo, ele se apresenta como uma técnica mais poderosa e versátil para a modelagem de fenômenos com geometria complexa e meios não homogêneos (SADIKU, 2001).

Figura 9 – Método das Diferenças Finitas e método dos Elementos Finitos



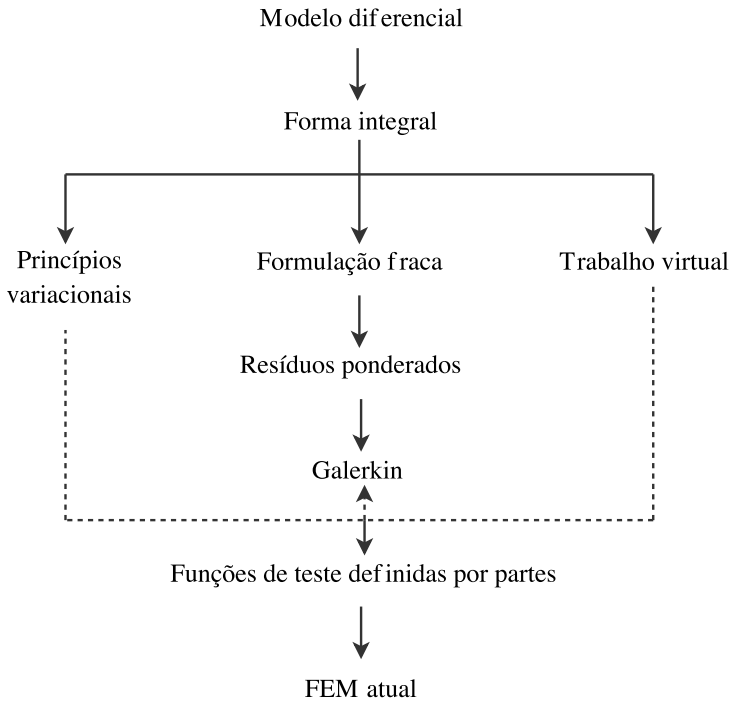
Fonte: Elaborada pelo autor

3.2.3 Formulações do FEM

As primeiras formulações do FEM são conhecidas como *abordagem direta* ou *formulação física*, que embora forneça a interpretação intuitiva do método, é util apenas para a resolução de problemas relativamente simples (HUEBNER et al., 2001; DESAI, 1972; ZIENKIEWICZ, 2005). O uso do princípio do trabalho virtual para a determinação de forças na abordagem direta levou à generalização do FEM por meio da estratégia de minimização do funcional de energia. Esta técnica mais genérica ficou conhecida como *formulação variacional* (DESAI, 1972; ZIENKIEWICZ, 2005; JIN, 2002) e tem como principal representante o método de Rayleigh-Ritz. Uma terceira abordagem, conhecida como *Método dos Resíduos Ponderados* ou *formulação generalizada* (ZIENKIEWICZ, 2005; HUEBNER et al., 2001), é tradicionalmente utilizada e é ainda mais genérica que o princípio variacional, pois resolve diretamente as equações diferenciais do modelo sem necessitar da existência de um funcional de energia (DESAI, 1972) ou em outras palavras, sem exigir que o sistema seja positivo definido (todos os autovalores positivos).

A figura 10 baseia-se nos diagramas apresentados por Zienkiewicz (2005) e contém a relação entre as diferentes formulações para a solução de um PVC relacionadas FEM. No presente trabalho será adotado o método dos resíduos ponderados, utilizando a abordagem de Galerkin para a escolha das funções de peso.

Figura 10 – Associação entre as estratégias de solução de PVC. Linha cheias representam relação direta, linhas tracejadas relação indireta. Embora o FEM seja apresentado na literatura por meio de diferentes abordagens, todas conduzem a resultados equivalentes.



Fonte: Elaborada pelo autor

3.2.4 Método dos Resíduos Ponderados

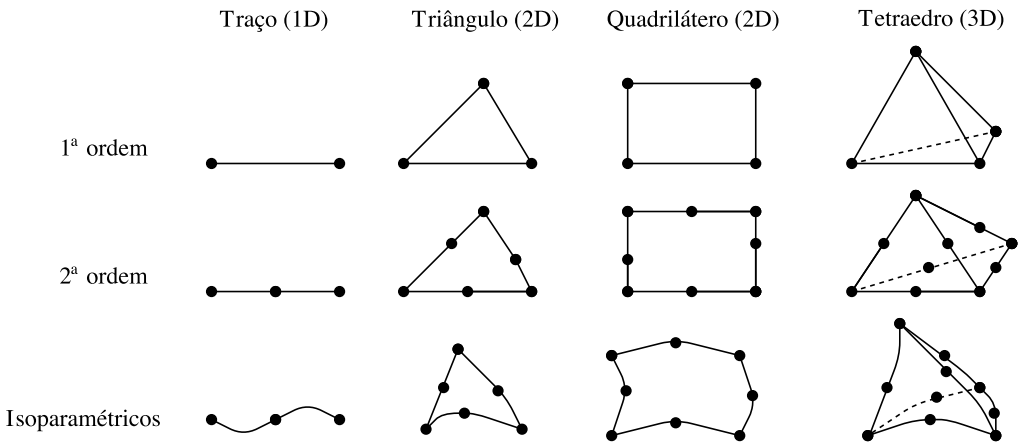
De acordo com Jin (2002), a aplicação do FEM pode ser feita a partir de quatro passos básicos: discretização do domínio, seleção das funções de interpolação, formulação do sistema de equações e solução do sistema de equações. Zienkiewicz (2005) faz uma separação em três etapas voltada para programas de computadores. A etapa de pré-processamento engloba a especificação da geometria e geração da malha. O processamento envolve a formação das matrizes dos elementos (matrizes de massa), montagem do sistema global, atribuição das condições de contorno e solução do sistema reduzido. Por fim, o pós-processamento é encarregado de exibir os resultados graficamente e realizar cálculos secundários, caso necessários. Nos tópicos a seguir são exploradas as etapas propostas por Jin (2002).

3.2.4.1 Discretização do domínio

A discretização do domínio consiste na transformação do contínuo Ω em uma malha de elementos finitos (discretos). Cada elemento Ω_e dessa malha representa um subdomínio de Ω. Nesta etapa são definidas a forma, a quantidade e o tamanho dos elementos, de maneira que a representação em malha seja a mais próxima possível do objeto em análise (DESAI, 1972). A escolha da geometria dos elementos está associada tanto à própria geometria do problema quanto à precisão que se deseja obter. Na figura 11 são mostrados exemplos de elementos que podem ser utilizados no processo de discretização. Os elementos que apresentam curva-

turas nas arestas ou faces são denominados isoparamétricos e fornecem maior acurácia na modelagem de superfícies curvadas (JIN, 2002). É possível também melhorar a precisão de elementos regulares (sem curvatura) por meio da inserção de nós nas arestas ou faces. Estes nós “adicionais” formam os chamados elementos de ordem superior.

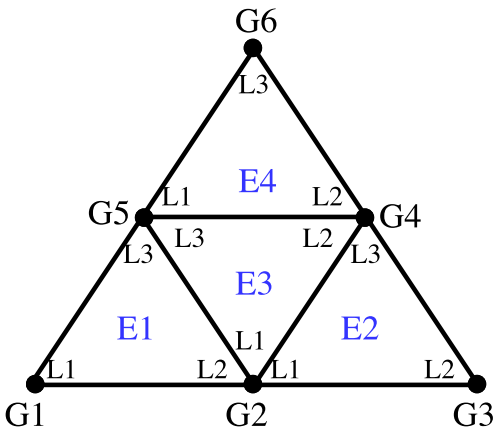
Figura 11 – Exemplos de possíveis elementos de uma malha



Fonte: Elaborada pelo autor

Como pode ser visto na figura 12, cada elemento é identificado na malha a partir de um identificador único que lhe é atribuído. Os vértices de cada elemento também são numerados. Cada nó possui dois valores vinculados a ele, um atuando como identificador global (numeração do nó na malha) e o outro como identificador local (numeração dentro de um dado elemento). A numeração local é geralmente feita no sentido anti-horário, a fim de se obter um valor positivo no cálculo da área ou volume por meio do determinante (SADIKU, 2001; JIN, 2002). Além da localização geral feita em termos das coordenadas xyz pode ser conveniente mapear ou transformar a localização de um elemento em termos das coordenadas naturais/adimensionais $\xi\eta\zeta$ (JIN, 2002). Também são comumente utilizadas coordenadas normalizadas de área e volume para elementos triangulares e tetraédricos (ZIENKIEWICZ, 2005).

Figura 12 – Identificadores de elementos e nós



Fonte: Elaborada pelo autor

3.2.4.2 Seleção das funções de interpolação

A etapa seguinte é a geração das funções de interpolação (funções de base ou forma) (HUEBNER et al., 2001), as quais são responsáveis por aproximar a solução ϕ no interior de cada elemento (JIN, 2002). Para atingir este objetivo são necessários três passos:

1. Tendo em mente a ordem e características dos elementos da malha, escolher uma função $\tilde{\phi}$ que aproxime a solução ϕ;
2. Escolhida a função $\tilde{\phi}$, determinar a aproximação $\tilde{\phi}_i^e$ para os nós i de cada elemento e, com base nas sua localização espacial;
3. Determinar as funções de interpolação N^e com suporte compacto no domínio de e, ou seja, uma função que cujo valor seja diferente de zero apenas dentro de Ω^e.

No primeiro passo geralmente é escolhido um polinômio para a função $\tilde{\phi}$, e isso ocorre por dois motivos a priori (DESAI, 1972):

- Facilidade de manipulação matemática, principalmente derivação e integração;
- Aproximação satisfatória quando truncado em uma ordem qualquer.

Na prática são escolhidos polinômios de primeira ou segunda ordem, mas ordens superiores podem ser adotadas para reduzir o erro de aproximação, sobretudo em bordas com curvaturas acentuadas, no entanto ocorre também o aumento da carga computacional (JIN, 2002). A escolha da ordem do polinômio $\tilde{\phi}$ está relacionada à ordem do elemento escolhido, por exemplo, em um triângulo de primeira ordem (linear) a aproximação pode ser feita por uma função linear (3 incógnitas), já para um triângulo de segunda ordem, a aproximação é feita por uma função quadrática (6 incógnitas). As equações 3.4 e 3.5 mostram respectivamente os polinômios completos de primeira e segunda ordem.

$$\tilde{\phi}(x, y) = a + bx + cy \tag{3.4}$$

$$\tilde{\phi}(x, y) = a + bx + cy + dx^2 + exy + fy^2 \tag{3.5}$$

Para o segundo passo, considerando um subdomínio bidimensional triangular, a aproximação para o valor de ϕ no domínio de um elemento e pode ser feita pelo polinômio apresentado em 3.6. A obetenção dos valores da solução em cada nó de e pode pode então ser feita resolvendo-se o sistema 3.7 no qual x_i e y_i são as coordenadas do nó i e a, b e c são as incógnitas a se determinar. O sistema com as incógnitas isoladas é dado na equação 3.8.

$$\tilde{\phi^e} = a^e + b^e x + c^e y = \{1 \; x \; y\} \begin{Bmatrix} a^e \\ b^e \\ c^e \end{Bmatrix} \tag{3.6}$$

$$\tilde{\phi_i^e} = a^e + b^e x_i + c^e y_i \qquad i = 1, 2, 3 \tag{3.7}$$

$$\begin{Bmatrix} a^e \\ b^e \\ c^e \end{Bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} \begin{Bmatrix} \tilde{\phi_1^e} \\ \tilde{\phi_2^e} \\ \tilde{\phi_3^e} \end{Bmatrix} \tag{3.8}$$

O terceiro passo é realizado por meio da substituição o resultado de 3.8 na equação 3.6, pelo qual se obtém as funções de interpolação N_i^e para os pontos ϕ_i^e. A partir de então, é possível se obter o valor de $\tilde{\phi^e}$ em um ponto (x, y) qualquer no interior de e, conforme mostra a equação 3.9. Para elementos triangulares no plano cartesiano as funções N_i^e são geradas pelas relações apresentadas na equação 3.10 (JIN, 2002). Especificamente para o caso de triângulos de primeira ordem, as próprias funções N_i^e representam as coordenadas de área, normalizadas pela área A do triângulo, no entanto, para elementos de ordem superior são necessárias algumas manipulações adicionais e o uso de integração numérica (VOLAKIS; CHATTERJEE; KEMPEL, 1998; JIN, 2002; ZIENKIEWICZ, 2005).

$$\tilde{\phi^e} = \sum_{i=1}^{3} N_i^e(x, y)\phi_i^e = \{N^e\}^T\{\phi^e\} \tag{3.9}$$

$$\begin{cases} N_i^e(x, y) = \frac{1}{2A^e}\left(a_i^e + b_i^e x + c_i^e y\right) \qquad i = 1, 2, 3 \\ a^e = \{x_2^e y_3^e - x_3^e y_2^e,\ x_3^e y_1^e - x_1^e y_3^e,\ x_1^e y_2^e - x_2^e y_1^e\ \} \\ b^e = \{y_2^e - y_3^e,\ y_3^e - y_1^e,\ y_1^e - y_2^e\ \} \\ c^e = \{x_3^e - x_2^e,\ x_1^e - x_3^e,\ x_2^e - x_1^e\ \} \\ A^e = \frac{1}{2}\left(b_1^e c_2^e - b_2^e c_1^e\right) \end{cases} \tag{3.10}$$

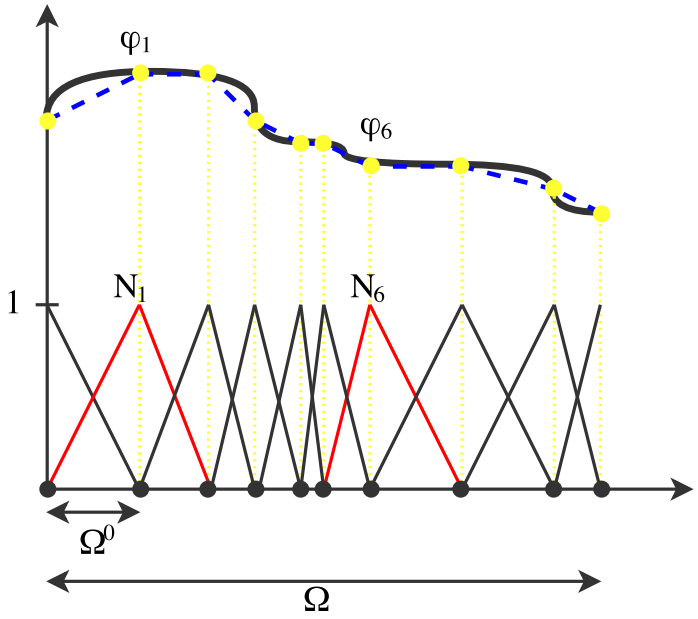
A figura 13 mostra um exemplo das funções de interpolação em um domínio Ω unidimensional e a figura 14 mostra uma função N_i em duas dimensões, composta pelas funções N_i^e de seis elementos. Strang (1973) chama as funções N_i de "chapéu" para problemas unidimensionais e "telhado" para problemas bidimensionais.

Além do suporte compacto, as funções N_i devem se escolhidas de forma que se comportem como o delta de Kronecker (JIN, 2002), como mostra a equação 3.11 a seguir. Dessa forma elas preservam os valores nodais ϕ_i.

$$N_i(x_j, y_j) = \delta_{ij} = \begin{cases} 1,\ i = j \\ 0,\ i \neq j \end{cases} \tag{3.11}$$

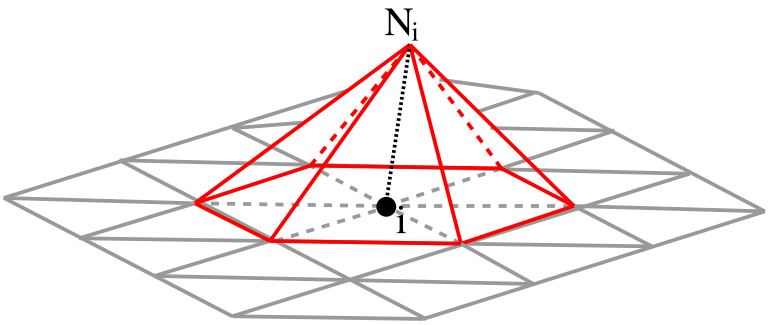
Além da família triangular/tetraédrica, existem as famílias *serendipity* e lagrangeana para quadriláteros e blocos. A diferença básica entre as duas é que os elementos da família de Lagrange podem apresentar nós internos (nas faces ou espaço), o que não ocorre com os elementos da *serendipity* (ZIENKIEWICZ, 2005; VOLAKIS; CHATTERJEE; KEMPEL, 1998).

Figura 13 – Funções de forma em 1D. Destaque para as funções N_1 e N_6



Fonte: Elaborada pelo autor

Figura 14 – Funções de forma em 2D para o nó i



Fonte: Elaborada pelo autor

3.2.4.3 Formulação do sistema de equações

Considere o PVC apresentado em 3.12. A equação de Poisson que modela a distribuição do potencial ϕ está sujeita às condições de contorno em $\partial\Omega_1$ e $\partial\Omega_2$. O operador Δ é o laplaciano, f é uma função de excitação conhecida e ϕ é a grandeza física procurada.

$$\begin{cases} \Delta\phi = f & em\ \Omega \\ \phi = a & em\ \partial\Omega_1 \\ \phi = b & em\ \partial\Omega_2 \end{cases} \tag{3.12}$$

Se for feita a substituição da solução exata ϕ por um valor aproximado, um resíduo r

surge em decorrência dos erros de aproximação, como pode ser visto na equação 3.13.

$$r = \Delta\tilde{\phi} - f \neq 0 \tag{3.13}$$

Embora o resíduo nos elementos seja diferente de zero em sua maioria (verificar os erros de aproximação na figura 13), deseja-se que o resíduo total, na média, seja igual a zero (VOLAKIS; CHATTERJEE; KEMPEL, 1998). Para este fim, conforme mostra a equação 3.14, deve-se realizar a ponderação do resíduo r por meio de uma função w adequada, e em seguida fazer a análise da perspectiva do contínuo, por meio da integração sobre o domínio Ω (ao invés da análise pontual fornecida pela equação diferencial). Esta estratégia é conhecida como Método dos Resíduos Ponderados e é originada da formulação fraca da equação diferencial (ZIENKIEWICZ, 2005). A derivada fraca é uma generalização da derivada clássica e por meio dela a função ϕ, chamada solução fraca, é buscada em um espaço de funções mais amplo, o espaço de Sobolev (EVANS, 1998; FURTADO, 2012).

$$R = \int_{\Omega} w \left(\Delta\tilde{\phi} - f\right) d\Omega = 0 \tag{3.14}$$

Uma vez que se tem o domínio Ω discretizado, é conveniente fazer a análise por partes, sobre cada domínio Ω_e. Ao invés de se escolher uma função w que atenda a todo o domínio, faz-se o uso de funções w^e mais simples, para cada elemento e. A equação 3.15 apresenta como é feita a análise para os três nós de um elemento triangular de primeira ordem.

$$R^e = \int_{\Omega} w^e \left(\Delta\tilde{\phi}^e - f^e\right)\ d\Omega = 0 \tag{3.15}$$

Com a expansão do sistema em 3.15 tem-se a equação 3.16. A aproximação $\tilde{\phi}^e$ foi obtida na etapa anterior e é apresentada na equação 3.9. Ao substituir $\tilde{\phi}^e$ na equação 3.16, tem-se o sistema 3.17. Os parâmetros x e y foram omitidos para melhor legibilidade da fórmula.

$$\int_{\Omega} w^e \Delta\tilde{\phi}^e\ d\Omega = \int_{\Omega} w^e f^e\ d\Omega \tag{3.16}$$

$$\int_{\Omega} w^e \sum_{i=1}^{3} N_i^e \Delta\phi_i^e\ d\Omega = \int_{\Omega} w^e f^e\ d\Omega \tag{3.17}$$

Dependendo da escolha da função de ponderação w o método dos resíduos ponderados recebe nomes específicos. Alguns casos especiais são apresentados em 3.18.

Método de Petrov-Galerkin	$w = \psi_i \neq \phi$
Método de Galerkin	$w = \phi$
Método dos Mínimos quadrados	$w = \frac{d}{dx}\left(a(x)\frac{d\phi}{dx}\right)$
Método da colocação	$\delta(x - x_i)$

(3.18)

O método de Galerkin para um operador auto-adjunto e positivo definido equivale ao método de Ritz (ZIENKIEWICZ, 2005) (verificar relação indireta com os métodos variacionais

na figura 10). A equação 3.19 mostra a representação de Galerkin para para o elemento e. Após empregar a integração por partes (ou o teorema de Green) em 3.19 fazendo $u = N_j^e$ e $dv = \Delta N_i^e$ dada a relação $\int udv = uv - \int vdu$, obtém-se a forma fraca do laplaciano, a qual possui menor restrição de diferenciabilidade/suavidade, passando de C^1 para C^0. Além disso, é importante notar que durante essa manipulação surgem termos integrais sobre o contorno $\partial\Omega_1$ e $\partial\Omega_2$, no entanto, devido à propriedade apresentada na equação 3.11 essas integrais de borda desaparecem (valem zero no contorno) (REDDY, 2006) restando apenas as parcelas apresentadas na equação 3.20.

$$\sum_{i=1}^{3} \int_\Omega N_j^e \Delta N_i^e \phi_i^e \, d\Omega = \int_\Omega N_j^e f^e \, d\Omega \qquad j = 1, 2, 3 \tag{3.19}$$

$$\sum_{i=1}^{3} \phi_i^e \int_\Omega \nabla N_j^e \nabla N_i^e \, d\Omega = \int_\Omega N_j^e f^e \, d\Omega \qquad j = 1, 2, 3 \tag{3.20}$$

A equação 3.20 fornece o sistema linear para um elemento e o qual pode ser visto em linguagem matricial na equação 3.21. De posse das matrizes K^e (matriz de massa) é feito o mapeamento dos identificadores locais para os identificadores globais. Este processo é conhecido como montagem do sistema global (matriz de rigidez). Do ponto de vista algébrico, o mapeamento das matrizes dos elementos para o sistema global pode ser ser feito por meio de uma matriz L chamada matriz de conexão de nós (NCM) (WU et al., 2015) ou de transição (KISS et al., 2012). L é uma matriz booleana com uma linha para cada vértice de cada elemento e uma coluna para cada vértice do sistema. A matriz global K pode ser obtida pela equação 3.22, na qual a matriz E é a matriz de bloco diagonal contendo as matrizes de cada elemento na diagonal principal.

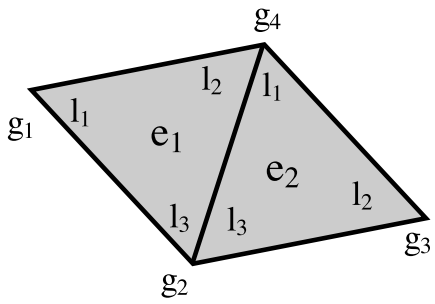
$$[K^e]\{\phi^e\} = \{b^e\} \tag{3.21}$$

$$K = L^T E L \tag{3.22}$$

Considere a ilustração da figura 15. Para este exemplo as matrizes L e E são mostradas na equação 3.23. É conveniente também pensar em termos de uma função Map que associa os identificadores locais com os identificadores globais. As equações 3.24, 3.25 e 3.26 mostram respectivamente as funções de mapeamento para este exemplo, o mapeamento do primeiro elemento e o mapeamento total. A equação 3.27 mostra o sistema global sobre o qual serão aplicadas as condições de contorno.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad E = \begin{bmatrix} K_{11}^1 & K_{12}^1 & K_{13}^1 & 0 & 0 & 0 \\ K_{21}^1 & K_{22}^1 & K_{23}^1 & 0 & 0 & 0 \\ K_{31}^1 & K_{32}^1 & K_{33}^1 & 0 & 0 & 0 \\ 0 & 0 & 0 & K_{11}^2 & K_{12}^2 & K_{13}^2 \\ 0 & 0 & 0 & K_{21}^2 & K_{22}^2 & K_{23}^2 \\ 0 & 0 & 0 & K_{31}^2 & K_{32}^2 & K_{33}^2 \end{bmatrix} \tag{3.23}$$

Figura 15 – Exemplo para a montagem do sistema global



Fonte: Elaborada pelo autor

$$\begin{aligned} Map(e_1) &: l_1 \rightarrow g_1,\ l_2 \rightarrow g_4,\ l_3 \rightarrow g_2 \\ Map(e_2) &: l_1 \rightarrow g_4,\ l_2 \rightarrow g_3,\ l_3 \rightarrow g_2 \end{aligned} \tag{3.24}$$

$$Map(e_1) \Rightarrow \begin{bmatrix} K^1_{11} & K^1_{13} & 0 & K^1_{12} \\ K^1_{31} & K^1_{33} & 0 & K^1_{32} \\ 0 & 1 & 0 & 0 \\ K^1_{21} & K^1_{23} & 0 & K^1_{22} \end{bmatrix} \tag{3.25}$$

$$Map(e_1) + Map(e_2) \Rightarrow \begin{bmatrix} K^1_{11} & K^1_{13} & 0 & K^1_{12} \\ K^1_{31} & K^1_{33} + K^2_{33} & K^2_{32} & K^1_{32} + K^2_{31} \\ 0 & K^2_{23} & K^2_{22} & K^2_{21} \\ K^1_{21} & K^1_{23} + K^2_{13} & K^2_{12} & K^1_{22} + K^2_{11} \end{bmatrix} \tag{3.26}$$

$$\begin{bmatrix} K^1_{11} & K^1_{13} & 0 & K^1_{12} \\ K^1_{31} & K^1_{33} + K^2_{33} & K^2_{32} & K^1_{32} + K^2_{31} \\ 0 & K^2_{23} & K^2_{22} & K^2_{21} \\ K^1_{21} & K^1_{23} + K^2_{13} & K^2_{12} & K^1_{22} + K^2_{11} \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{Bmatrix} \tag{3.27}$$

O sistema global é da ordem do número de nós da malha. Para problemas reais ele é esparso e dependendo da ordem da numeração dos nós pode ter maior parte dos elementos confinados em torno da diagonal principal, formando assim uma matriz de banda (ZIENKIEWICZ, 2005). Uma vez montado o sistema, as condições de contorno são aplicadas.

Neste trabalho são abordadas apenas as condições de contorno de Dirichlet, que consistem na substituição das incógnitas dos nós de contorno por seus respectivos valores. Assim, se um sistema de ordem 1000 possui 200 valores pré-fixados ele pode ser reduzido à ordem 800. Considerando ainda o exemplo da figura 15, suponha que os nós g_1 e g_3 componham respectivamente os contornos $\partial\Omega_1$ e $\partial\Omega_2$. Tem-se que $\phi = u_1$ em $\partial\Omega_1$ e $\phi = u_2$ em $\partial\Omega_2$. Assim sendo, modifica-se o sistema 3.27 de forma a atender às condições de contorno sem perder

a simetria (JIN, 2002). O resultado das atribuições são apresentados na equação 3.28 e o sistema reduzido a resolvido na próxima etapa na equação 3.29.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & K_{33}^1 + K_{33}^2 & 0 & K_{32}^1 + K_{31}^2 \\ 0 & 0 & 1 & 0 \\ 0 & K_{23}^1 + K_{13}^2 & 0 & K_{22}^1 + K_{11}^2 \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix} = \begin{Bmatrix} u_1 \\ b_2 - K_{31}^1 - K_{32}^2 \\ u_2 \\ b_4 - K_{21}^1 - K_{12}^2 \end{Bmatrix} \tag{3.28}$$

$$\begin{bmatrix} K_{33}^1 + K_{33}^2 & K_{32}^1 + K_{31}^2 \\ K_{23}^1 + K_{13}^2 & K_{22}^1 + K_{11}^2 \end{bmatrix} \begin{Bmatrix} \phi_2 \\ \phi_4 \end{Bmatrix} = \begin{Bmatrix} b_2 - K_{31}^1 - K_{32}^2 \\ b_4 - K_{21}^1 - K_{12}^2 \end{Bmatrix} \tag{3.29}$$

3.2.4.4 Solução do sistema de equações

Para resolver o sistema linear originado da etapa anterior podem ser utilizados diferentes algoritmos numéricos, os quais são geralmente agrupados em duas classes: métodos diretos e métodos iterativos (VOLAKIS; CHATTERJEE; KEMPEL, 1998). Segundo Faires (2008), as técnicas da primeira classe consistem em uma sequência finita de operações a fim obter a solução exata do problema, a qual está sujeita apenas aos erros de arredontamento. Os métodos iterativos por sua vez são estratégias que usam aproximações sucessivas a partir de um "chute" ou suposição inicial a fim de obter a solução dentro de uma tolerância pré-estabelecida (STRANG, 2013). A taxa de convergência, ou seja, o quão rápido o método se aproxima da solução exata depende do espectro da matriz de coeficientes e portanto, do seu número de condição (BARRETT et al., 1995).

Conforme apresenta Faires (2008) a solução de um sistema linear por métodos diretos tem como principal instrumento a eliminação gaussiana, a qual consiste em transformar um sistema linear qualquer em um sistema triangular por meio de repetidas combinações lineares entre as linhas (FRANCO, 2006). No entanto, dada a complexidade algorítimica desse método ($O(n^3/3)$), foram desenvolvidas estratégias conhecidas como fatoração ou decomposição, as quais tiram proveito da estrutura do sistema para realizar a solução por partes. Alguns tipos de fatoração são nomeados na tabela 2 .

Contudo, como ressalta (VOLAKIS; CHATTERJEE; KEMPEL, 1998), o uso de métodos diretos para sistemas reais, grandes e esparsos, pode demandar alto consumo de processamento e memória. Para tais sistemas, a recomendação recai sobre os métodos iterativos Franco (2006), Faires (2008). Segundo Barrett et al. (1995) esta classe pode ser separada em métodos estacionários e não estacionários. A equação 3.30 mostra o esquema geral de métodos estacionários, nos quais os parâmetros B e c não variam durante as iterações. A tabela 3 contém a descrição de três algoritmos deste tipo.

$$x^{(k)} = Bx^{(k-1)} + c \qquad k = 1, 2, ... \tag{3.30}$$

Ainda segundo Barrett et al. (1995), métodos não estacionários se diferenciam dos anteriores devido à tomada de decisão baseada em iterações anteriores. Os métodos do subespaço de Krylov fazem parte desta classe e são comumente utilizados na literatura devido

Fatoração	**Descrição**
$A = LU$	L é uma matriz triangular inferior e U uma matriz triangular superior. Se U apresentar diagonal unitária tem-se a fatoração de Crout
$A = LDU$	Similar à LU sendo que D uma matriz diagonal e L e U apresentam diagonal unitária
$PA = LU$	Fatoração LL com permutação de linhas por meio da matriz P
$A = LL^T$	Denominada fatoração de Cholesky, aplicável a sistemas positivos definidos
$A = LDL^T$	Fatoração de Crout. Similar à de Cholesky mas com L apresentando diagonal unitária.
$A = QR$	Aplicável a uma matriz $m \times n$. Q é uma matriz unitária e R é triangular superior

Tabela 2 – Exemplos de técnicas de fatoração de matrizes (FRANCO, 2006)

Método	**Descrição**
Jacobi	Resolve uma variável x em relação às demais. A cada iteração resolve todas as variáveis uma única vez. Possui a seguinte forma matricial: $x^{(k)} = D^{-1}(L + U)x^{(k-1)} + D^{-1}b$. É fácil de implementar e entender, mas converge lentamente.
Gauss-Seidel (GS)	Utiliza as atualizações de x mais recentes (realizadas anteriormente na mesma iteração). Em geral, se o método de Jacobi converge, o GS converge mais rapidamente. Sua representação matricial é dada por: $x^{(k)} = (D - L)^{-1}Ux^{(k-1)} + (D - L)^{-1}b$.
Sobrerrelaxamento Sucessivo (SOR)	Derivado do método de Gauss-Seidel por meio do uso de um parâmetro de extrapolação ω. Dependendo da escolha de ω, o método converge mais rapidamente que o GS. Sua forma matricial é: $x^{(k)} = (D - \omega L)^{-1}[(1 - \omega)D + \omega U]x^{(k-1)} + \omega(D - \omega L)^{-1}b$

Tabela 3 – Métodos estacionários (BARRETT et al., 1995; FAIRES, 2008)

à sua robustez. Alguns trabalhos recentes são os de Anzt et al. (2016), Ahamed e Magoulès (2016), Yang, Liu e Chen (2016). De acordo com Strang (2013), estes métodos apresentam baixo custo computacional uma vez que são realizadas apenas poucas multiplicações de matriz por vetor. Basicamente, em cada iteração dos métodos de Krylov é calculada uma nova direção p e um novo tamanho de passo (ou distância) α nesta direção, como apresenta a equação 3.31. A tabela 4 apresenta três métodos do subespaço de Krylov.

$$x^{(k)} = x^{(k-1)} + \alpha_{k-1}p^{(k-1)} \qquad k = 1, 2, ... \tag{3.31}$$

Além de úteis para o processamento de grandes sistemas esparsos, os métodos iterativos têm a capacidade de se autocorrigirem em caso de erro (STRANG, 2013) e podem ser utilizados para a redução dos erros de arredondamento e sob algumas condições, serem aplicados para sistemas não lineares (FRANCO, 2006). Essa classe de solucionadores também é facilmente adaptada para uso em computadores paralelos (FAIRES, 2008). Na seção

Método	**Descrição**
Gradientes Conjugados (CG)	O CG funciona de forma similar ao método da descida mais íngreme, no entanto gera uma sequência e direções ortogonais entre si (não repete uma mesma direção). Esta técnica é aplicável a sistemas positivos definidos e tem como objetivo minimizar o resíduo (ou o gradiente) a cada iteração. Além disso apresenta a vantagem de armazenar apenas alguns poucos vetores.
Resíduos Mínimos Generalizados (GMRES)	Este método calcula uma sequência de vetores ortogonais e os combina por meio de mínimos quadrados. Ele é útil para matrizes não simétricas mas é custoso em termos de armazenamento, pois armazena os dados calculados em cada iteração. para limitar o armazenamento podem ser feitas renicializações periódicas.
Gradiente Biconjugado (BiCG)	O BiCG faz a busca de direções utilizando 2 resíduos mutuamente ortogonais, os quais são obtidos partir da matriz de coeficientes A e de sua transposta A^T. Com esta adaptação o método pode ser utilizado em sistemas assimétricos. De forma a melhorar a convergência pode-se utilizar o método com estabilização (BiCGStab).

Tabela 4 – Métodos do subespaço de Krylov (BARRETT et al., 1995)

seguinte, o método CG utilizado neste trabalho é explicado em detalhes.

3.3 Método dos Gradientes Conjugados

O método do gradiente conjugado (CG) foi proposto por Hestenes e Stiefel em 1952 como um método direto para a solução de sistemas lineares positivos definidos (com todos os autovalores positivos) (FAIRES, 2008). Embora sua concepção seja de um método direto que converge após n iterações, sendo n a ordem do sistema, ele diferencia-se da eliminação gaussiana por poder ser encerrado parcialmente (STRANG, 2013). Assim sendo, sua aplicação tem sido feita como um método iterativo para se aproximar a solução de sistemas esparsos, os quais com o uso de um precondicionador adequado convergem em torno de $\sqrt{n}$ iterações (FAIRES, 2008).

3.3.1 Matrizes Positivas Definidas (PD)

Considere uma função $y = f(x)$ na qual a derivada primeira num ponto x_a é igual a zero, ou seja, $f'(x_a) = 0$. Neste caso fala-se que x_a é um ponto estacionário (ou crítico) de f e pode ser classificado como ponto de máximo, mínimo ou de inflexão (ponto de sela) (STRANG, 2013). A determinação deste tipo está associada à derivada segunda, sendo que se $f''(x_a) > 0$, x_a é um ponto de mínimo, se $f''(x_a) < 0$, x_a é um ponto de máximo e se $f''(x_a) = 0$, x_a é um ponto de sela (FRANCO, 2006).

Seja $f(x)$ uma função quadrática no plano cartesiano, a qual pode ser escrita como $f(x) = ax^2 + bx + c$. Extrapolando tal função para o espaço multidimensional, pode-se escrevê-la convenientemente, como será visto a seguir, na forma vetorial, dada pela equação 3.32 (SHEWCHUK, 1994). Para um domínio bidimensional em que $\mathbf{A}$ é uma matriz de ordem 2, $\mathbf{b}$ é um vetor 2×1 e γ é um escalar, é obtida uma função tal qual a apresentada na equação 3.5. Para se obter exatamente esta função genérica, deve se aplicar na função de 3.32 os valores relacionados na equação 3.33. A partir da forma quadrática $f(x, y)$ é possível se obter diferentes superfícies, como mostra a figura 16, dependendo dos valores que compõem $\mathbf{A}$ e $\mathbf{b}$. A fim de facilitar a notação, vetores e matrizes estão indicados em negrito.

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + \gamma \tag{3.32}$$

$$\mathbf{x} = \begin{Bmatrix} x \\ y \end{Bmatrix} \qquad \mathbf{A} = \begin{bmatrix} 2d & e \\ e & 2f \end{bmatrix} \qquad \mathbf{b} = \begin{Bmatrix} b \\ c \end{Bmatrix} \qquad \gamma = a \tag{3.33}$$

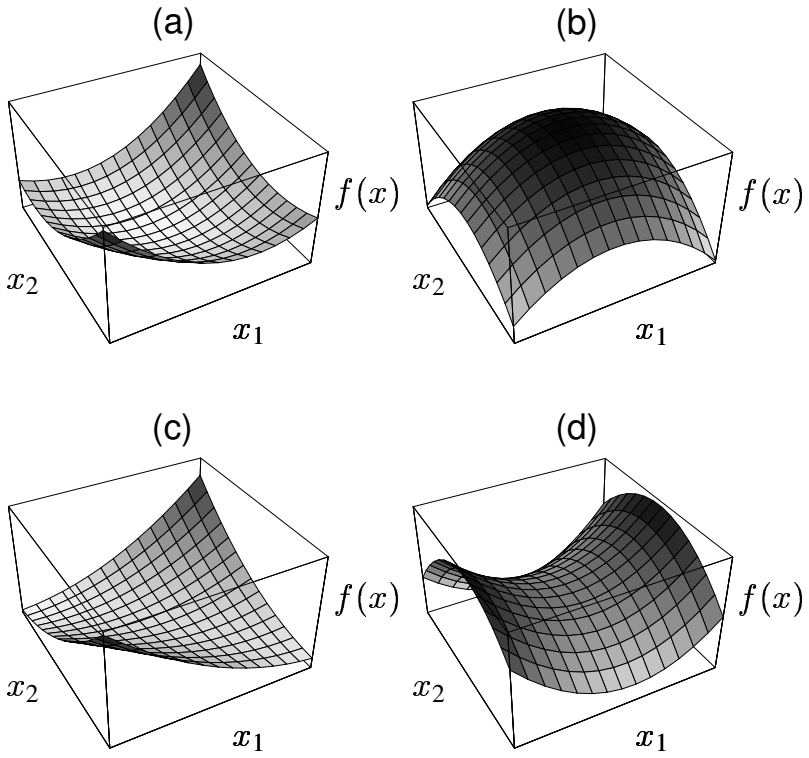
O comportamento de um sistema PD em três dimensões é dado pelo paraboloide na figura 16(a), o qual apresenta um único ponto de mínimo, fazendo com que todos os demais sejam maiores que este (SHEWCHUK, 1994). Segundo Strang (2013), embora não seja possível visualizar, o mesmo comportamento pode ser obtido em formas quadráticas do espaço $\mathbf{R}^n$, variando-se a ordem da matriz $\mathbf{A}$ de 2 para n e utilizando-se um vetor $\mathbf{b}$ de comprimento $n \times 1$. Se os autovalores de $\mathbf{A}$ são positivos, o problema na n-ésima dimensão é PD. Este tipo de sistema é originado de diferentes campos da física e da engenharia (SADIKU, 2001) e é conhecido como funcional de energia, cuja análise está associada à área de análise funcional e do cálculo de variações (ZIENKIEWICZ, 2005; SADIKU, 2001). Em seu livro, Jin (2002) apresenta vários funcionais, como por exemplo para análise de dispersão, antenas e para os campos elétrico e magnético.

Conforme apresentado por Franco (2006), a solução de um funcional de energia consiste em encontrar o seu ponto de mínimo $\mathbf{x_s}$. Dada a sua característica de positividade, tal ponto é o único ponto crítico do sistema e por isso sua derivada primeira é nula. Para problemas de muitas variáveis, utiliza-se o conceito de gradiente apresentado na equação 3.34, o qual é vetor contendo as derivadas parciais do problema e aponta para o sentido de maior crescimento da função(SHEWCHUK, 1994).

$$f'(\mathbf{x}) = \nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \frac{\partial}{\partial x_2} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{bmatrix} \tag{3.34}$$

A equação 3.35 apresenta o gradiente da forma quadrática 3.32. Se a matriz $\mathbf{A}$ é simétrica ($\mathbf{A} = \mathbf{A}^T$), este gradiente equivale ao problema $\mathbf{A}\mathbf{x} = \mathbf{b}$. Assim sendo, dado um sistema linear PD pode-se considerá-lo como o gradiente de um funcional de energia e encontrar por

Figura 16 – Superfícies quádricas. (a) Paraboloide elíptico originado de um sistema PD. (b) Paraboloide elíptico gerado por um sistema negativo definido. (c) Cilindro parabólico originado de um sistema positivo indefinido. (d) Paraboloide hiperbólico gerado por um sistema indefinido.



Fonte: (SHEWCHUK, 1994)

meio deste segundo problema o hiperplano que o estacionariza (ponto de mínimo), o qual equivale à solução do sistema linear original Franco (2006), Shewchuk (1994).

$$\begin{aligned} f'(\mathbf{x}) &= \nabla f(\mathbf{x}) = \frac{1}{2}\mathbf{A}^T\mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - b \\ f'(\mathbf{x_s}) &= 0 = \mathbf{A}\mathbf{x_s} - \mathbf{b} \end{aligned} \tag{3.35}$$

3.3.2 Método da descida mais íngreme

O método dos gradientes, também conhecido como método da descida mais íngreme, obtém o ponto de mínimo de um funcional de energia caminhando no sentido contrário do gradiente ($-\nabla f$) a cada iteração k (FRANCO, 2006). Uma vez que são feitas aproximações $\mathbf{x^{(k)}}$ para o valor da solução $\mathbf{x}$ partindo de uma suposição inicial $\mathbf{x_0}$, o gradiente pode ser interpretado como um resíduo $\mathbf{r^{(k)}}$, como mostra a equação 3.36, sendo que no ponto $\mathbf{x_s}$ o gradiente (ou resíduo) vale zero.

$$\mathbf{r}^{(k)} = -\nabla f^{(k)}(\mathbf{x}^{(k)}) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} \tag{3.36}$$

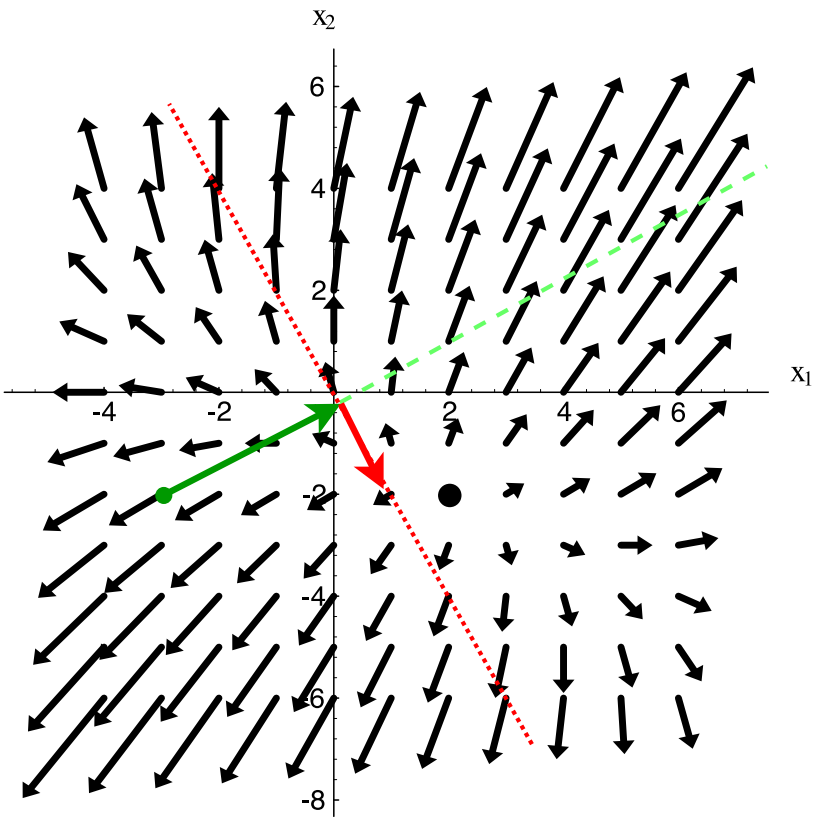
Segundo a equação 3.31, para a obtenção das aproximações de $\mathbf{x}$, os parâmetros

de direção (**p**) e passo (α) devem ser estabelecidos. Intuitivamente, partindo-se de um ponto qualquer do funcional representado na figura 16(a), a direção mais rápida para se chegar no ponto de mínimo é a direção da descida mais íngreme, ou seja, a direção do gradiente, mas em sentido oposto (SHEWCHUK, 1994). Como dito no parágrafo anterior, a variável que aponta para este sentido é o resíduo $\mathbf{r^{(k)}}$. A equação 3.37 apresenta uma modificação de 3.31 incluindo o resíduo como direção de busca.

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \alpha_{k-1}\mathbf{r}^{(k-1)} \qquad k = 1, 2, ... \tag{3.37}$$

Uma vez que se conhece a direção $\mathbf{r^{(k)}}$ em que se deve caminhar, define-se o quanto caminhar. A resposta para esta questão é apresentada em detalhes por Saad (2003) e baseia-se no princípio de projeções. Como pode ser visto na figura 17, quando se caminha no sentido contrário ao do gradiente partindo-se de um determinado ponto, o limite de descida é dado quando se encontra um novo gradiente (ou resíduo) ortogonal ao atual. Esta ortogonalidade, mostrada na equação 3.38, garante que o resíduo da próxima direção será zero em relação ao da direção atual (STRANG, 2013). O cálculo da distância α é demonstrado pela equação 3.39 (SHEWCHUK, 1994).

Figura 17 – Tomando-se um ponto qualquer ($(-2, -3)$ por exemplo) deve-se caminhar no sentido do resíduo (vetor verde) até que se chegue ao gradiente na direção ortogonal (vetor vermelho) ao atual. Este processo é repetido até se chegar a uma distância satisfatória do valor mínimo x_s



Fonte: Elaborada pelo autor

$$\mathbf{r}^{T(k)}\mathbf{r}^{(k-1)} = 0 \qquad k = 1, 2, ... \tag{3.38}$$

$$\begin{aligned}
&\mathbf{r}^{T(k)}\mathbf{r}^{(k-1)} = 0 \\
&(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)})^T\mathbf{r}^{(k-1)} = 0 \\
&(\mathbf{b} - \mathbf{A}(\mathbf{x}^{(k-1)} + \alpha_{k-1}\mathbf{r}^{(k-1)}))^T\mathbf{r}^{(k-1)} = 0 \\
&(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k-1)})\mathbf{r}^{(k-1)} - \alpha_{k-1}(\mathbf{A}\mathbf{r}^{(k-1)})^T\mathbf{r}^{(k-1)} = 0 \\
&(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k-1)})\mathbf{r}^{(k-1)} = \alpha_{k-1}(\mathbf{A}\mathbf{r}^{(k-1)})^T\mathbf{r}^{(k-1)} \\
&\mathbf{r}^{T(k-1)}\mathbf{r}^{(k-1)} = \alpha_{k-1}\mathbf{r}^{\mathbf{T}(k-1)}(\mathbf{A}\mathbf{r}^{(k-1)}) \\
&\alpha_{k-1} = \frac{\mathbf{r}^{T(k-1)}\mathbf{r}^{(k-1)}}{\mathbf{r}^{T(k-1)}\mathbf{A}\mathbf{r}^{(k-1)}}
\end{aligned} \tag{3.39}$$

3.3.3 Direções conjugadas e gradientes conjugados

Embora o método da descida mais íngreme seja um algorítimo efetivo para se encontrar a solução de problemas SPD (simétricos e positivos definidos), algumas melhorias podem ser realizadas para acelerar a convergência (FRANCO, 2006; SHEWCHUK, 1994).

Como pode ser visto na figura 17, caso fosse realizado um terceiro passo, ele seria feito na mesma direção do passo inicial. Este custo adicional devido à repetição de direções pode ser evitado caminhando-se o máximo necessário em uma determinada direção (SHEWCHUK, 1994). Dessa forma, em um problema SPD bidimensional seriam necessárias apenas duas iterações (ou passos) para se alcançar a solução, como por exemplo uma iteração caminhando na direção do eixo x e outra na direção do eixo y. Uma outra alternativa além das bases canônicas poderia ser caminhar o máximo na primeira direção e só então buscar uma nova direção.

Para que isso seja possível, é necessário que ao invés de se caminhar no sentido de resíduos consecutivos ortogonais, deve-se caminhar nas direções de uma base ortogonal de $\mathbf{A}$. Esta condição de $\mathbf{A}$-ortogonalidade (FAIRES, 2008), também conhecida como direções conjugadas (FRANCO, 2006) é obtida para um conjunto de vetores $\mathbf{d}$ por meio do produto interno apresentado na equação 3.40. Partindo-se deste resultado chega-se ao novo valor de α mostrado em 3.41 de forma similar ao obtido em 3.39. Este processo de conjugação pode ser feito por meio da ortogonalização de Gram-Schmidt (SAAD, 2003; STRANG, 2013).

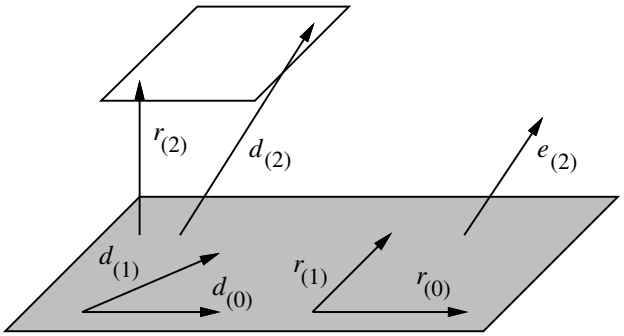
$$\mathbf{d}^{T(k)}\mathbf{A}\mathbf{d}^{(k-1)} = 0 \qquad k = 1, 2, ... \tag{3.40}$$

$$\alpha_{k-1} = \frac{\mathbf{d}^{T(k-1)}\mathbf{r}^{(k-1)}}{\mathbf{d}^{T(k-1)}\mathbf{A}\mathbf{d}^{(k-1)}} \tag{3.41}$$

Tendo em vista que conjunto de direções de busca $\{\mathbf{d^{(0)}}, \mathbf{d^{(1)}}, \ldots, \mathbf{d^{(k)}}\}$ é obtido a partir de uma base ortogonal de $\mathbf{A}$ e que os resíduos tomados para a minimização do funcional são ortogonais, pode-se utilizar uma base composta por resíduos para gerar $\mathbf{A}$. Esta é a

estratégia do método dos gradientes conjugados, por meio do qual uma nova direção de busca passa a ser tomada em função das direções e dos resíduos anteriores (SHEWCHUK, 1994). Considere a figura 18 a qual apresenta a direção atual $\mathbf{d^{(1)}}$ e a anterior $\mathbf{d^{(0)}}$ que geram o subespaço (cinza) de $\mathbf{A}$. Tomando convenientemente os resíduos $\mathbf{r^{(0)}}$ e $\mathbf{r^{(1)}}$ ortogonais como base para este subespaço, calcula-se o resíduo seguinte $\mathbf{r^{(2)}}$, ortogonal aos anteriores, e por meio da combinação linear deste com a direção atual $\mathbf{d^{(1)}}$ obtém-se um nova direção $\mathbf{d^{(2)}}$ e consequentemente bases para um novo subespaço.

Figura 18 – Obtenção de uma nova direção de busca em uma nova dimensão, de 2D para 3D.



Fonte: (SHEWCHUK, 1994)

A garantia de $\mathbf{A}$-ortogonalização de uma direção $\mathbf{d}$ em relação à todas as anteriores demanda o armazenamento do vetor $\mathbf{d}$ e sua comparação com os demais a cada iteração, o que leva a uma complexidade $\mathcal{O}(n^3)$ (SHEWCHUK, 1994). Contudo, no CG é necessário o armazenamento apenas entre iterações consecutivas e a garantia de $\mathbf{A}$-ortogonalidade ocorre por definição, sendo necessário apenas o cálculo do parâmetro de correção β , como mostra a equação 3.42 (BARRETT et al., 1995).

$$\beta_k = \frac{\mathbf{r}^{T(k)}\mathbf{r}^{(k)}}{\mathbf{r}^{T(k-1)}\mathbf{r}^{(k-1)}} \tag{3.42}$$

Em cada iteração do CG são obtidos vetores que geram um subespaço $\mathbf{K}^k$ de Krylov, conforme apresentado na equação 3.43. Esta construção possibilita que a cada iteração sejam realizadas apenas operações de produto interno e multiplicação de matriz por vetor as quais são menos custosas que as multiplicações de matriz por matriz (STRANG, 2013). O comportamento expresso em 3.43 é o que caracteriza um método como “do subespaço de Krylov”. O algoritmo 5 contém um pseudocódigo do CG proposto por (BARRETT et al., 1995).

$$\mathbf{K}^k = \left[\mathbf{b}\ \ \mathbf{Ab}\ \ \mathbf{A^2b}\ \ \ldots\ \ \mathbf{A^{k-1}b}\right] \tag{3.43}$$

3.3.4 Análise de convergência

Como dito na seção 3.2.4.4, métodos iterativos fornecem uma série de aproximações para a solução de um problema, sendo necessária a definição de alguns parâmetros adicionais a fim de se verificar se os valores aproximados estão perto o suficiente da solução.

Algoritmo 5 Algoritmo dos Gradientes Conjugados (SHEWCHUK, 1994)

$d_0 = r_0 = b - Ax_0$
for j = 1,2... **do**
  $\alpha = \frac{{r^i}^T r^i}{{d^i}^T A d^i}$
  $x^{i+1} = x^i + \alpha d^i$
  $r^{i+1} = r^i - \alpha A d^i$
  $\beta = \frac{{r^{i+1}}^T r^{i+1}}{{r^i}^T r^i}$
  $d^{i+1} = r^{i+1} + \beta d^i$
  if $\frac{\|r^{i+1}\|}{\|b\|}$ **then**
    break
  end if
end for

A taxa de convergência, ou seja, o quão rápido o método obtém uma solução adequada (ou ainda, o número de iterações necessárias para se obter tal solução) é dada em função do número de condição da matriz de coeficientes, o qual está associado à sensibilidade de um sistema à pequenas alterações no modelo $\mathbf{A}$ ou na entrada $\mathbf{A}$ (STRANG, 2013). A equação 3.44 mostra um exemplo de como a inclusão de um pequeno erro afeta o resultado do sistema.

$$\begin{bmatrix} 2 & 6 \\ 2 & 6.00001 \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} 8 \\ 8.00001 \end{Bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 2 & 6 \\ 2 & 5.99999 \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} 8 \\ 8.00002 \end{Bmatrix} \tag{3.44}$$

Enquanto o primeiro sistema apresenta o resultado $\{1, 1\}^T$ a inclusão de um erro da ordem de $1e^{-5}$ retorna como resultado o vetor $\{10, -2\}^T$, o qual é consideravelmente diferente do primeiro. A distância euclidiana entre estes vetores é $\sqrt{90} \approx 9.48$, o que leva a uma amplificação do erro de entrada para o erro na saída num fator de $1e^6$.

O número de condição $\kappa(\mathbf{A})$ de uma matriz estabelece a relação entre uma alteração na entrada (ou modelo) e seu efeito na saída do sistema (STRANG, 2013). Se a introdução de um erro produz uma variação equivalente na saída, tem-se que a matriz é *bem condicionada* e que o valor de $\kappa(\mathbf{A})$ é proximo de 1. No entanto, se uma pequena variação na entrada é amplificada muitas vezes na saída, tem-se um sistema *mal condicionado*, e o valor de $\kappa(\mathbf{A})$ neste caso é significantemente maior que 1 (FAIRES, 2008). O número $\kappa(\mathbf{A})$ é definido conforme a equação 3.45 para o caso geral e equivalentemente em 3.46 para matrizes simétricas (STRANG, 2013).

$$\kappa(\mathbf{A}) = \| \mathbf{A} \| \cdot \| \mathbf{A}^{-1} \| \tag{3.45}$$

$$\kappa(\mathbf{A}) = \frac{\lambda_{max}}{\lambda_{min}} \tag{3.46}$$

Dada a grande amplificação do erro no exemplo 3.44, imagina-se que este seja mal condicionado. Por meio da função $cond$ do MATLAB® é possível calcular o número de condição da primeira matriz do exemplo, o qual é igual a $4e^6$. A fim de melhorar a convergência

de um método iterativo, pode ser utilizada uma matriz similar a $\mathbf{A}$, mas que apresente melhor valor de $\kappa(\mathbf{A})$ (STRANG, 2013). A obtenção de tais matrizes pode ser feita por meio de técnicas de precondicionamento, decomposição de valores singulares (SVD) ou por meio de regularização (SAAD, 2003). Na subseção seguinte serão apresentadas algumas estratégias de precondicionamento utilizadas neste trabalho.

3.4 Precondicionamento e aceleração da convergência

Como foi visto na seção anterior, a taxa de convergência do sistema está vinculada ao número de condição da matriz de coeficientes. A fim de se reduzir a quantidade de iterações de um método iterativo ou até mesmo possibilitar a convergência de sistemas quase singulares ($\kappa(\mathbf{A})$ tendendo ao infinito) são utilizadas matrizes semelhantes a $\kappa(\mathbf{A})$ mas que tenham melhores propriedades espectrais, ou seja, autovalores que forneçam um número de condição mais próximo de 1 (BARRETT et al., 1995; SAAD, 2003).

A forma geral de um sistema precondicionado é dada na equação 3.47. A matriz $\mathbf{M}$ é denominada *precondicionador* e deve ser escolhida de tal forma que o número de condição de $\mathbf{M}^{-1}\mathbf{A}$ seja menor que o de $\mathbf{A}$ (BARRETT et al., 1995). Segundo Strang (2013) os precondicionadores podem ser obtidos a partir de uma separação $\mathbf{A} = \mathbf{M} - \mathbf{N}$. Se $\mathbf{M} = \mathbf{A}$ logo $\mathbf{N} = \mathbf{0}$ o sistema é resolvido diretamente por meio de eliminação (única iteração). Contudo, se $\mathbf{M}$ é semelhante a $\mathbf{A}$ e mais facilmente invertível, tem-se que $\mathbf{M}$ é um bom precondicionador.

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \tag{3.47}$$

Como sugere Strang (2013), uma escolha inicial satisfatória para precondicionadores pode ser a adoção da separação realizada nos métodos estacionários apresentados na tabela 3. Sendo a matriz de coeficientes decomposta na forma $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$, tem-se no método de Jacobi que $\mathbf{M} = \mathbf{D}$ e $\mathbf{N} = -\mathbf{L} - \mathbf{U}$. No método de Gauss-Seidel a separação fica: $\mathbf{M} = \mathbf{D} + \mathbf{L}$ e $\mathbf{N} = -\mathbf{U}$. Em ambos os casos $\mathbf{M}$ fornece uma aproximação de $\mathbf{A}$ e menor número de condição (SAAD, 2003). As fórmulas apresentadas na tabela 3 são originadas a partir destes valores de $\mathbf{M}$ e $\mathbf{N}$ como é mostrado na equação 3.48, a qual culmina na fórmula geral 3.30 dos métodos estacionários.

$$\begin{aligned}
&\mathbf{A}\mathbf{x} = \mathbf{b} \\
&(\mathbf{M} - \mathbf{N})\mathbf{x} = \mathbf{b} \\
&\mathbf{M}\mathbf{x}^{(k)} = \mathbf{N}\mathbf{x}^{(k-1)} + \mathbf{b} \\
&\mathbf{x}^{(k)} = \mathbf{M}^{-1}\mathbf{N}\mathbf{x}^{(k-1)} + \mathbf{M}^{-1}\mathbf{b} \\
&\mathbf{x}^{(k)} = \mathbf{B}\mathbf{x}^{(k-1)} + \mathbf{c}
\end{aligned} \tag{3.48}$$

À partir da escolha de $\mathbf{M}$, pode-se submeter o sistema obtido em 3.47 para ser solucionado pelo CG (algoritmo 5) ou ainda submeter o sistema original $\mathbf{A}\mathbf{x} = \mathbf{b}$ ao método CG precondicionado apresentado no algoritmo 6.

Algoritmo 6 Algoritmo dos Gradientes Conjugados Precondicionado (SHEWCHUK, 1994)

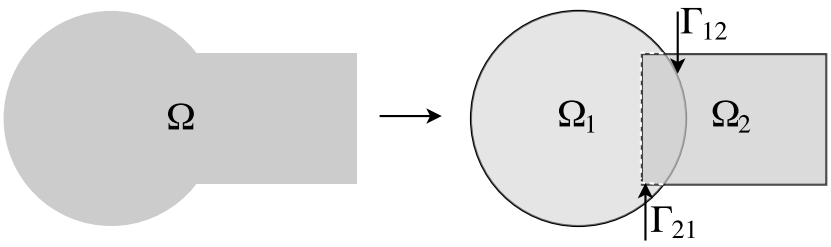
1: $r_0 = b - Ax_0$
2: $d_0 = M^{-1}r_0$
3: **for** j = 1,2... **do**
4: $\alpha = \frac{{r^i}^T M^{-1} r^i}{{d^i}^T A d^i}$
5: $x^{i+1} = x^i + \alpha d^i$
6: $r^{i+1} = r^i - \alpha A d^i$
7: $\beta = \frac{{r^{i+1}}^T M^{-1} r^{i+1}}{{r^i}^T M^{-1} r^i}$
8: $d^{i+1} = M^{-1} r^{i+1} + \beta d^i$
9: **if** $\frac{\|r^{i+1}\|}{\|b\|}$ **then**
10: break
11: **end if**
12: **end for**

Além do uso de métodos iterativos como precondicionadores existem também as fatorações incompletas, comumente utilizadas em sistemas esparsos (VOLAKIS; CHATTERJEE; KEMPEL, 1998; SAAD, 2003). As representantes mais comuns dessa categoria são as decomposições $ILU(0)$ e $IChol(0)$ as quais preservam o padrão de esparsidade da matriz original. Saad (2003) e Barrett et al. (1995) explicam que além das técnicas tradicionais de precondicionamento, diferentes métodos podem ser adotados de forma a se beneficiar do conhecimento que se tem sobre o problema modelado. Além disso, conforme Kiss et al. (2012) a convergência para a solução pode ser otimizada quando há a correspondência entre a estratégia de precondicionamento e o ambiente de execução. Com base nestas afirmações são apresentados nas subseções a seguir duas estratégias utilizadas para a aceleração da convergência em computadores paralelos, as quais são adotadas neste trabalho.

3.4.1 Decomposição de domínio de Schwarz

A técnica de decomposição de domínio de Schwarz foi descrita pela primeira vez em 1870 por Hermann Schwarz como uma estratégia para a solução PDEs baseada na política de dividir para conquistar Saad (2003). Sendo Ω o domínio original deseja-se resolver o problema sobre um conjunto de n subdomínios Ω_i menores e sobrespostos, como mostra a figura 19.

Figura 19 – Exemplo de decomposição de Schwarz



Fonte: Elaborada pelo autor

Originalmente o método foi concebido em sua forma sequencial, o denominado método alternado (ou multiplicativo) de Schwarz, o qual consiste na alternância de solução entre os

subdomínios e na atualização das fronteitas Γ após cada iteração. Como pode ser visto na figura 19, a fronteira Γ_{21} é interior ao domínio Ω_1 e é uma borda "artificial" de Ω_2. Ao se iterar sobre o subdomínio Ω_1 os valores dessa fronteira são atualizados e utilizados como condições de contorno de Dirichlet em Ω_2 (DOLEAN; JOLIVET, 2016; SAAD, 2003). O algoritmo 7 ilustra o processo de solução de um PVC pelo método multiplicativo de Schwarz.

Algoritmo 7 Algoritmo do método alternado de Schwarz (DOLEAN; JOLIVET, 2016)

1: $u^1 = 0$
2: **for** j = 1,2... **do**
3: $\quad -\Delta u^{j+1/2} = f \quad em\ \Omega_1$
4: $\quad u^{j+1/2} = v_1 \quad em\ \partial\Omega_1 \backslash \Gamma_{12}$
5: $\quad u^{j+1/2} = u^j \quad em\ \Gamma_{12}$
6: $\quad -\Delta u^{j+1} = f \quad em\ \Omega_2$
7: $\quad u^{j+1} = v_2 \quad em\ \partial\Omega_2 \backslash \Gamma_{21}$
8: $\quad u^{j+1} = u^{j+1/2} \quad em\ \Gamma_{21}$
9: **end for**

A versão paralela desta técnica é apresentada no algoritmo 8 e é denominada método de Jacobi-Schwarz (JSM) ou método Aditivo Restrito de Schwarz (RAS). Ela consiste na utilização de soluções locais u_i ao invés de uma solução global u, de forma que as iterações possam ocorrer simultaneamente sem que haja sobreposição de dados (DOLEAN; JOLIVET, 2016; SAAD, 2003).

Algoritmo 8 Algoritmo do método aditivo restrito de Schwarz (DOLEAN; JOLIVET, 2016)

1: $u^1 = 0$
2: **for** j = 1,2... **do**
3: $\quad -\Delta u_1^{j+1} = f \quad em\ \Omega_1$
4: $\quad u_1^{j+1} = v_1 \quad em\ \partial\Omega_1 \backslash \Gamma_{12}$
5: $\quad u_1^{j+1} = u^j \quad em\ \Gamma_{12}$
6: $\quad -\Delta u_2^{j+1} = f \quad em\ \Omega_2$
7: $\quad u_2^{j+1} = v_2 \quad em\ \partial\Omega_2 \backslash \Gamma_{21}$
8: $\quad u_2^{j+1} = u^j \quad em\ \Gamma_{21}$
9: **end for**

Em ambos os casos, se em uma dada iteração os valores das regiões sobrespostas são iguais, ou a distância entre eles é menor que uma certa tolerância tem-se a convergência do método.

3.4.2 Solução elemento a elemento

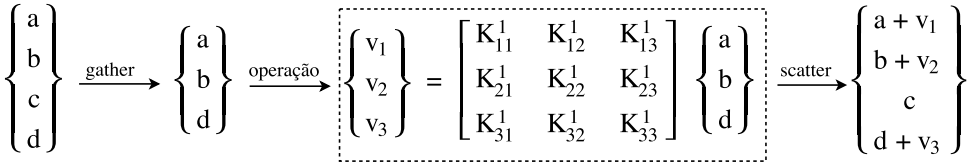
Esta forma de solução consiste em resolver um sistema de elementos finitos sem montar a matriz de coeficientes global, o que Saad (2003) chama de "FEM desmontado" (*Unassembled FEM*). Assim sendo, as operações do solucionador são decompostas e aplicadas sobre as matrizes de massa (matrizes dos elementos) ao invés da matriz global (KISS et al., 2012).

Os objetivos originais desta estratégia eram principalmente reduzir a quantidade de dados armazenados e de operações aritméticas (LEVIT, 1987) bem como tirar proveito do

paralelismo dos computadores vetoriais, quando associada à coloração da malha (separação de elementos vizinhos) (CAREY et al., 1988; WATHEN, 1989; DAYDE; L'EXCELLENT; GOULD, 1995). Com a popularização das GPGPU, as representantes mais atuais de processadores SIMD, houve uma retomada no interesse por solucionadores EBE (KISS et al., 2012; WU et al., 2015; YAN et al., 2017) a fim de se explorar o paralelismo disponível nessas novas arquiteturas para solução do FEM.

Na técnica EBE cada operação do tipo matriz-vetor presente no algoritmo 6 pode ser realizada em termos das matrizes $\mathbf{A^e}$ e $\mathbf{M^e}$ dos elementos. Os vetores de cada operação são obtidos por meio de uma função *gather* que busca os valores nos vetores globais e os resultados são salvos de volta por uma função *scatter*, como mostra a figura 20.

Figura 20 – Mapeamento da operação realizada sobre a matriz de um elemento



Fonte: Elaborada pelo autor

De acordo com Saad (2003) além da associação com métodos de Krylov a técnica EBE pode ser aplicada a métodos diretos, como por exemplo o multifrontal, no qual a eliminação gaussiana pode ser feita em poucos elementos por vez. Uma das dificuldades de se utilizar o método EBE é que a maioria dos precondicionadores pressupõem uma matriz de coeficientes global (KISS et al., 2012). No entanto, algumas alternativas de precondicionamento para o FEM desmontado podem ser encontradas em Dayde, L'Excellent e Gould (1995), Wu et al. (2015), Yan et al. (2017).

Enquanto a decomposição de Schwarz divide um problema em um conjunto de sub-problemas completos e independentes (granulação grossa), o método EBE com coloração atua sobre algumas operações paralelizáveis (granulação fina) de um único grande problema. Dessa forma, estas estratégias podem ser associadas a fim de se tirar proveito dos diferentes níveis de paralelismo em máquinas MIMD e SIMD.

4 Materiais e métodos

“Quem decidir se colocar como juiz da verdade e do conhecimento é naufragado pela gargalhada dos deuses”.
Albert Einstein

O conjunto de materiais e métodos deste trabalho está dividido nas etapas de *validação do problema*, *aplicação das estratégias* e *aplicação de otimizações*. Na primeira são apresentados os recursos utilizados para modelar o PVC e para verificar a viabilidade de solução por coloração em um ambiente multiprocessado. Na segunda são apresentadas as decisões corretivas utilizadas para melhorar os resultados da primeira etapa. A última seção consiste na exploração do paralelismo em nível de dados e de instrução bem como no uso de precondicionamento.

4.1 Validação do problema

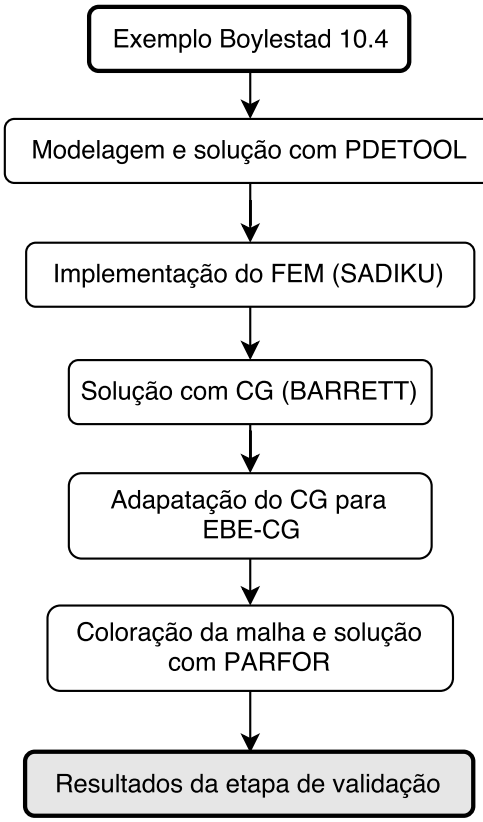
O ambiente de desenvolvimento MATLAB® foi adotado para a realização da etapa de validação. A motivação dessa escolha se deu pela facilidade da manipulação de matrizes oferecida pelo ambiente e pela existência de ferramentas dedicadas à solução de PVCs (*Partial Differential Equations Toolbox* - PDETool) e à programação paralela (*Parallel Computing Toolbox* - ParTool). A figura 21 mostra o fluxograma das atividades da primeira etapa.

4.1.1 Especificação do problema

O problema *benchmark* utilizado neste trabalho é o exemplo 10.4 do livro Análise de Circuitos de Boylestad (2011) e é ilustrado na figura 22(a). O capacitor é composto de duas placas quadradas paralelas com lado de 2 polegadas $(5,08cm)$ sendo a distância entre elas igual $\frac{1}{32}$ polegadas $(0,0794cm)$. O problema informa que a diferença de potencial entre as placas é de $48V$, dessa forma foi assumido que uma das placas possui $48V$ e que a outra está aterrada. A distribuição do campo elétrico é calculada entre as placas e no espaço ao redor. para isso é considerada uma região quadrada de $16cm$ de lado como mostra o esquema na figura 22(b). Também foi considerado que a espessura de cada placa é a metade da largura da região entre elas, ou seja, $\frac{1}{64}$ polegadas $(0,0397cm)$.

A fim de se obter um parâmetro da corretude dos resultados calculados, foi feita a simulação do problema na PDETool. Por meio dessa ferramenta é possível via programação ou interface gráfica definir a equação diferencial que rege o problema, sua a geometria, condições de contorno, a malha e a forma de exibição dos resultados. As figuras 22(c) e 22(d) mostram a geometria e a malha inicial modeladas na PDETool.

Figura 21 – Fluxo de trabalho para a validação da estratégia de paralelização do EBE-CG com coloração



Fonte: Elaborada pelo autor

4.1.2 Solução do PVC na PDETool

Inicialmente são definidas a geometria e a malha e em seguida são feitas as atribuições das condições de contorno de Dirichlet nas 4 bordas da seção transversal de cada placa. Foram atribuídos $48V$ à placa da esquerda e $0V$ à placa da direita. Para se definir a EDP do problema é necessário atribuir valores aos coeficientes da fórmula geral das EDPs elípticas, mostrada na equação 4.1.

$$-\nabla \cdot (c\nabla u) + au = f \tag{4.1}$$

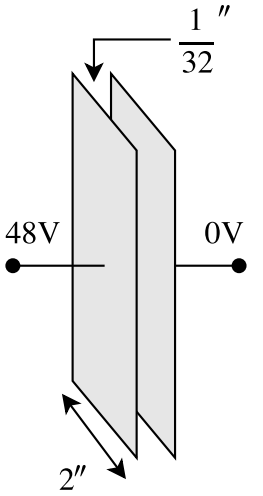
Na equação de Laplace, que modela a distribuição do potencial elétrico como apresentado na seção 1.2, apenas o coeficiente c é igual a 1 e os demais iguais a zero. Estabelecidos estes valores a PDETool resolve o PVC e apresenta graficamente os resultados, como pode ser visto nas figuras 23(a) e 23(b). A solução obtida é utilizada para avaliar, por meio do cálculo da norma do máximo (norma infinito), a corretude dos algoritmos implementados.

4.1.3 Implementação do FEM

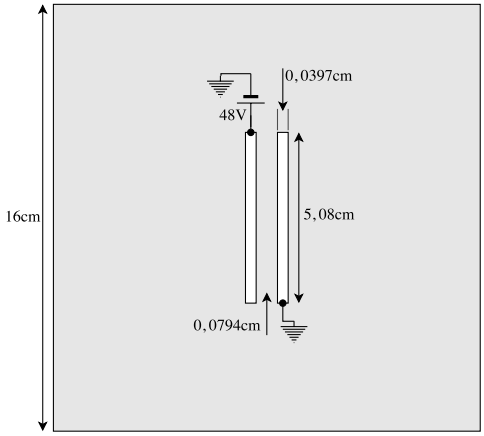
Após a obtenção dos valores de referência foi realizada a implementação do método dos elementos finitos em duas dimensões, conforme o apresentado no apêndice A. A fim

Figura 22 – Geometria do capacitor de placas paralelas

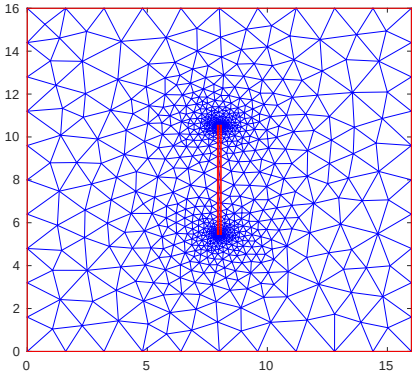
(a) Perspectiva do capacitor de placas paralelas



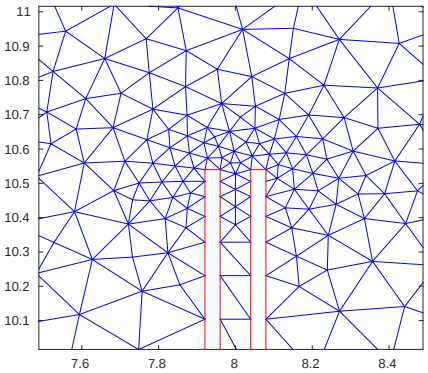
(b) Esquema da região de análise



(c) Região de análise com malha



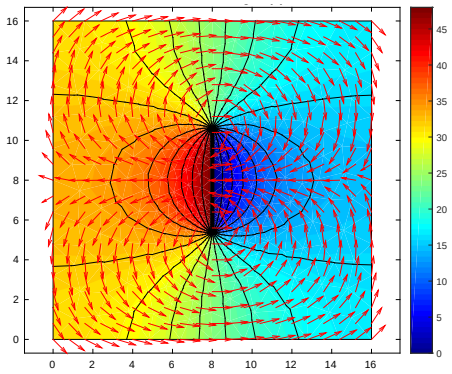
(d) Zoom sobre as placas do capacitor



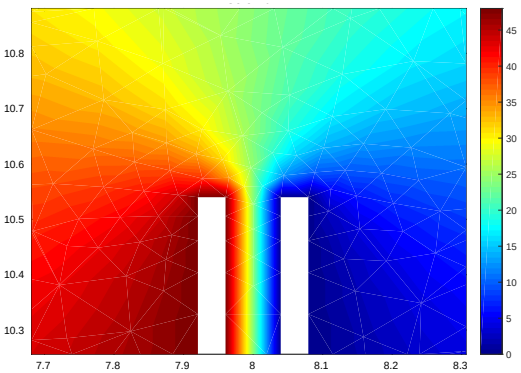
Fonte: Elaborada pelo autor

Figura 23 – Solução do PVC com a PDETool

(a) Distribuição do potencial em Volts e fluxo do campo elétrico em Volts/centímetro (vetores normalizados)



(b) Distribuição do potencial entre as placas



Fonte: Elaborada pelo autor

de não tirar proveito das funções do MATLAB®de forma a manter a portabilidade da solução implementada e a facilidade de transcrevê-la para outras linguagens, foram utilizadas nos algoritmos deste trabalho apenas as funções da PDETool relativas à geometria, geração da malha e exibição dos resultados. No algoritomo 9 as palavras grafadas em itálico são variáveis, as funções em negrito fazem parte do MATLAB®e as funções descritas nas linhas de 9 a 12 são implementadas neste trabalho e detalhadas na subseção 3.2.4. A tabela 5 contém a descrição de cada uma.

Algoritmo 9 Pseudocódigo do FEM

1: **load**($dadosGeometria$);
2: **load**($coordReferencia$);
3: **load**($dadosContorno$);
4: g = **decsg**($dadosGeometria$);
5: m = **createpde**(1);
6: **geometryFromEdges**(m,g);
7: **generateMesh**(m, 'Hmax',$valHmax$);
8: (nos,tri) = **meshToPet**($m.Mesh$);
9: C = geraMatElementares(nos, tri);
10: G = geraMatGlobal(C, tri);
11: (A, b) = atribuiContorno(G, $dadosContorno$);
12: sol = resolveSistLinear(A,b);
13: **pdeplot**(m,...);

A função `resolveSistLinear` é o ponto central deste trabalho, uma vez que a análise é feita sobre o desempenho das implementações durante solução do sistema de equações originado do FEM. Foram utilizados dois solucionadores a fim de se verificar a performance, sendo o primeiro o *solver* `assempde` da PDETool e o segundo a implementação do método dos gradientes conjugados proposta por Barrett et al. (1995).

A solução por meio de funções do MATLAB®é otimizada pela análise da estrutura do sistema, utilizando diferentes algoritmos para cada caso, de forma a tirar proveito de características como simetria e esparsidade (MATHWORKS, 2017a). Especificamente para o problema deste trabalho, cuja matriz é SPD, o `mldivide` adota a fatoração de Cholesky ou a LDL. Para sistemas esparsos, assimétricos e mal condicionados o MATLAB®recorre ao método multifrontal (SUITESPARSE, 2017).

A motivação para o uso do CG implementado por Barrett et al. (1995) está no fato de que este algoritmo é citado em diferentes trabalhos da revisão bibliográfica, como em Kiss et al. (2012), Dolwithayakul, Chantrapornchai e Chumchob (2012), Yang, Liu e Chen (2016). Além de um completo *survey* de métodos numéricos para a solução de sistemas lineares, os autores disponibilizam as implementações na linguagem do MATLAB®e em C++. O pseudocódigo 10 apresenta a implementação do CG disponibilizada por Barrett et al. (1995). Este pseudocódigo é mais próximo do código de linguagem de programação enquanto a do algoritmo 6 é mais próxima da formulação matemática.

`Função`	**Descrição**
`load`	Carrega para o *workspace* as variáveis salvas em arquivos.
`decsg`	Cria a geometria do problema por meio da associação de regiões primitivas. O espaço de análise do capacitor de placas paralelas, por exemplo é composto por 2 retângulos e 1 quadrado.
`createpde`	Instancia um modelo de PDE contendo n equações.
`geometryFromEdges`	Vincula a geometria originada da função `decsg` ao modelo de PDE gerado por `createpde`.
`generateMesh`	Cria uma malha sobre a geometria do modelo de PDE. Parâmetros adicionais podem ser incluídos para modificar a qualidade malha ou a ordem dos elementos (ver 3.2.4.2). O parâmetro `Hmax` determina o tamanho máximo das arestas dos elementos.
`meshToPet`	Obtém as matrizes de pontos, arestas e triângulos, as quais serão utilizadas na geração das matrizes elementares e da matriz global.
`geraMatElementares`	À partir dos dados da triangulação da malha, gera a matriz de cada elemento por meio das funções de aproximação e interpolação.
`geraMatGlobal`	Realiza a agregação ou o mapeamento das matrizes elementares no sistema global. A matriz resultante é esparsa e sua ordem corresponde ao número de nós da malha.
`atribuiContorno`	Atribui os m valores de contorno pré-estabelecidos e com isso realiza a redução dos sistema em m ordens, fazendo com que deixe de ser homogêneo.
`resolveSistLinear`	Consiste na aplicação de métodos numéricos para a solução eficiente de sistemas lineares esparsos.
`pdePlot`	Função utilizada para a exibição da malha e dos resultados. Conforme os parâmetros adicionais, são plotados gráficos tridimensionais, campos vetoriais e linhas de campo.

Tabela 5 – Descrição das funções utilizadas no algoritmo do FEM

4.1.4 Implementação do EBE-FEM

A implementação do EBE-FEM consiste basicamente em antecipar a solução do sistema de equações, passando da geração das matrizes dos elementos diretamente para a atribuição das condições de contorno e solução do sistema, sem que seja necessária a montagem da matriz global de coeficientes. Para realizar esta mudança de rota na solução do problema é preciso modificar tanto o algoritmo quanto a estrutura de dados utilizados.

4.1.4.1 Atribuição das condições de contorno

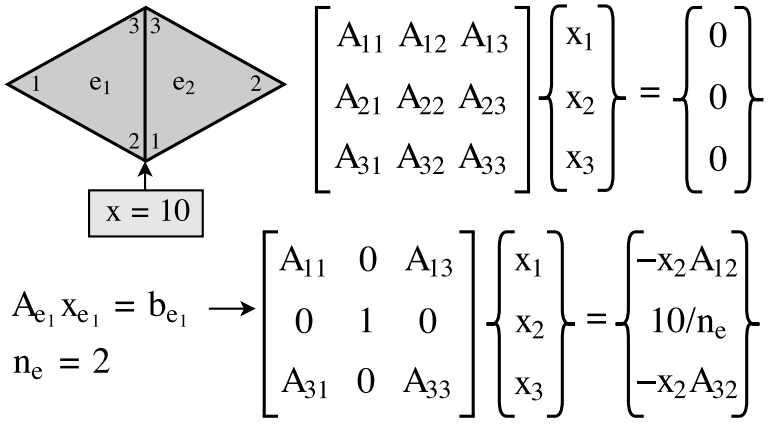
A adaptação do bloco referente à atribuição das condições de contorno (linha 11 do algoritmo 9) é baseada na estratégia proposta por Xu, Yin e Mao (2005) a qual consiste na criação de vetores $\mathbf{b}$ ou $\mathbf{rhs}$ (*right hand side*) para cada elemento. Assim sendo, ao invés de se atribuir os valores de contorno às células do vetor $\mathbf{rhs}$ global, que correspondem à cada nó da malha, a atribuição é feita nos vetores $\mathbf{b}^e$ de cada elemento. Por exemplo, se um nó

Algoritmo 10 Pseudocódigo do CG (BARRETT et al., 1995)

```
x_0 = {0}
r_0 = b − Ax_0
for i = 1,2... do
    z_{i−1} = resolve(M, 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} + β_{i−1} p_{i−1}
    end if
    q_i = A p_i
    α_i = ρ_{i−1} / p_i^T q_i
    x_i = x_{i−1} + α_i p_i
    r_i = r_{i−1} - α_i q_i
    if converge(r,tol) then
        break
    end if
end for
```

compartilhado por 2 elementos possui valor de contorno igual a 10, a atribuição é feita como uma média no $\mathbf{b}^e$ de cada um dos 2 elementos. Assim sendo, cada vetor recebe o valor de contorno igual a 5. A figura 24 contém o esquema que ilustra este exemplo. Yan et al. (2017) e Wu et al. (2015) propõe uma abordagem similar baseada não no número de conexões do nó, mas na média ponderada do valor de contorno original pelos coeficientes das matrizes dos elementos.

Figura 24 – Esquema da aplicação das condições de contorno de um elemento



Fonte: Elaborada pelo autor

4.1.4.2 Adaptação do CG

De posse das matrizes e vetores dos elementos com as condições de contorno atribuídas, é feita a adaptação no algoritmo do CG a fim de que os cálculos envolvendo matrizes

sejam feitos elemento a elemento. As operações entre vetores ou entre vetores e escalares são mantidas sem alteração visto que já são naturalmente paralelizáveis. As adaptações consistem basicamente em percorrer os N elementos realizando operações de *gather* para buscar os dados dos nós de um elemento e em seguida, realizar operações de *scatter* para distribuir os resultados nos vetores globais. As linhas 2, 4 e 12 do algoritmo 10 possuem operações matriciais e são modificadas conforme apresentado nos pseudocódigos 11, 12 e 13.

Algoritmo 11 Adaptação da declaração $r_0 = b - Ax_0$

1: $b = \{0\}$
2: **for** j = 1,2...$|tri|$ **do** ▷ Itera de 1 à quantidade triângulos na malha
3: $map = tri(j)$ ▷ Obtém os nós globais do triângulo de número j
4: $gat = b(map)$ ▷ Operação de *gather*
5: $b(map) = gat + b_e(j)$ ▷ Operação de *scatter*
6: **end for**
7: $r_0 = b$

Algoritmo 12 Adaptação da declaração z_{i-1} = resolve(M, r_{i-1})

1: $z = \{0\}$
2: **for** j = 1,2...$|tri|$ **do**
3: $map = tri(j)$
4: $M_e = precond(A_e(j))$
5: $z(map) = z(map) + M_e r(map)$ ▷ *gather-scatter*
6: **end for**

Algoritmo 13 Adaptação da declaração $q_i = Ap_i$

1: $q = \{0\}$
2: **for** j = 1,2...$|tri|$ **do**
3: $map = tri(j)$
4: $q(map) = q(map) + A_e(j)q(map)$ ▷ *gather-scatter*
5: **end for**

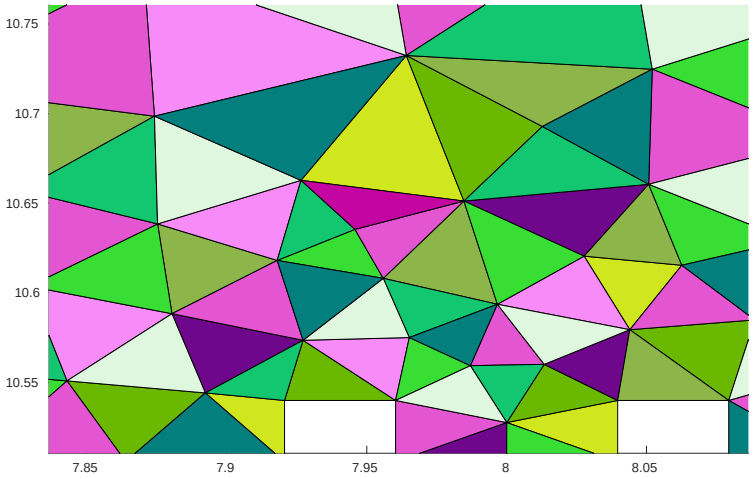
4.1.5 Coloração da malha

A coloração da malha possibilita que os trechos do CG descritos nos algoritmos 11, 12 e 13 sejam paralelizados (WATHEN, 1989; KISS et al., 2012). Assim como o processo de exclusão mútua, a coloração é uma estratégia adotada para evitar condições de corrida, as quais configuram um estado inconsistente dos dados causado por transações não atômicas. Desta forma, se por exemplo dois processos ou *threads* alteram ao mesmo tempo um dado em memória compartilhada, não se pode prever o valor final deste dado.

Para a paralelização do EBE-FEM, a coloração deve acontecer de tal forma que elementos que compartilhem um mesmo nó possuam cores distintas. Uma característica desejável num algoritmo otimizado para o EBE-FEM é que além de uma quantidade mínima de cores, deve haver equalização na quantidade de elementos que pertencem a cada cor. Por exemplo, numa malha de 1000 elementos pode ser preferível se utilizar 10 cores com 100 elementos de cada cor, ao invés de 6 cores com uma distribuição desequilibrada de elementos, como por

exemplo $\{500\ 50\ 200\ 150\ 10\ 90\}$. A figura 25 mostra um zoom na coloração da malha no topo entre as duas placas.

Figura 25 – Exemplo de coloração da malha



Fonte: Elaborada pelo autor

Após ser feita a coloração, o processamento pode ocorrer de tal forma que elementos da mesma cor seja executados simultaneamente e elementos de cores distintas sequencialmente. Como elementos de uma determinada cor não possuem vértices em comum, não há possibilidade de acontecer condição de corrida durante o acesso às variáveis globais z, r, q e d dos algoritmos 14, 15 e 16. A estrutura de dados `cores` contém as informações dos nós e elementos pertencentes a cada cor.

Algoritmo 14 Aplicação da coloração no algoritmo 11

```
b = {0}
for c = 1,2...|cores| do                    ▷ Itera de 1 à quantidade cores da coloração
    triCor = cores(c)                       ▷ Obtém os triângulos da cor c
    for j = 1,2...|triCor| do               ▷ Itera de 1 à quantidade triângulos da cor c
        idG = triCor(j)                     ▷ Obtém o identificador do triângulo na malha
        map = tri(idG)                      ▷ Obtém os nós globais do triângulo de número idG
        b(map) = b(map) + b_e(idG)
    end for
end for
r_0 = b
```

4.1.6 Paralelização do algoritmo

A ParTool, utilizada para a paralelização do algoritmo no MATLAB®, disponibiliza uma API de alto nível para a programação paralela em plataformas *multicore*, GPUs, *clusters*, *grids* e *cloud* (MATHWORKS, 2017c). Com esta ferramenta, a paralelização é feita por meio de *workers* ao invés de *threads* a fim de manter a portabilidade do algoritmo. Cada *worker* é um processo que trabalha em função do processo principal (*client*), o qual coordena a divisão de tarefas num modelo mestre-escravo. Quando necessária, a comunicação entre os *workers*

Algoritmo 15 Aplicação da coloração no algoritmo 12

```
1: z = {0}
2: for c = 1,2...|cores| do
3:     triCor = cores(c)
4:     for j = 1,2...|triCor| do
5:         idG = triCor(j)
6:         map = tri(idG)
7:         M_e = precond(A_e(idG))
8:         z(map) = z(map) + M_e r(map)
9:     end for
10: end for
```

Algoritmo 16 Aplicação da coloração no algoritmo 13

```
1: q = {0}
2: for c = 1,2...|cores| do
3:     triCor = cores(c)
4:     for j = 1,2...|triCor| do
5:         idG = triCor(j)
6:         map = tri(idG)
7:         q(map) = q(map) + A_e(idG)q(map)
8:     end for
9: end for
```

é feita por meio de troca de mensagens (MPI) . A tabela 6 contém a descrição das funções utilizadas da ParTool.

O bloco `parfor` atua de forma similar às funções SIMD, tirando proveito do paralelismo em nível de dados presente em *loops*. Mesmo com a coloração da malha, a API não permite que sejam realizadas indexações indiretas (operações *gather*), como ocorre nos algoritmos de 14 a 16. A fim de contornar este problema foi realizada uma modificação na estrutura de dados, de forma que o objeto retornado da coloração possua, para cada cor, as matrizes e o vetor rhs de seus elementos, ao invés de apenas os seus identificadores. Essa alteração elimina a necessidade de operações de *gather* e *scatter* na parte paralela do *loop*. O algoritmo 17 mostra a aplicação do `parfor` no pseudocódigo 15. A mesma adaptação é feita para os algoritmos 14 e 16.

Como pode ser visto no capítulo de resultados, o processo de paralelização via `parfor` apresentou uma performance pior que sua versão sequencial, provavelmente devido ao custo de se trabalhar com processos (*workers*) ao invés de *threads* e à granulação fina da paralelização via coloração. Na seção seguinte são apresentadas estratégias para contornar estes possíveis gargalos.

4.2 Aplicação de estratégias

Tendo em mente os resultados da etapa anterior e as características do ambiente de execução (processador *multicore*), são adotadas as estratégias a seguir, as quais são esquematizadas na figura 26.

Função	Descrição
`gcp`	*Get current parallel pool* retorna as informações sobre os *workers* ativos.
`parpool`	Cria uma nova *parallel pool*, ou seja, um conjunto de *workers* e os demais processos necessários para o funcionamento da *toolbox*.
`parfor`	Executa as iterações de um *loop* em *workers*. Para que isso seja possível as iterações devem ser independentes entre si e as estruturas de dados (coleções) utilizadas devem ter a característica de serem "fatiadas". Por exemplo, uma matriz bidimensional pode ser fatiada de 2 formas, pelas linhas ou pelas colunas.
`spmd`	A função *single program, multiple data* executa o mesmo código em diferentes *workers*, sendo que cada um opera em uma parte da coleção de dados original. Diferente do `parfor`, pode-se paralelizar um algoritmo inteiro por este método (granulação grossa), sendo também possível a comunicação entre *workers*.
`codistributed`	Cria uma coleção de dados que é distribuída entre os *workers*.
`getLocalPart`	Obtém os dados de uma determinada partição de uma coleção distribuída.
`codistributor`	Função que realiza a distribuição de uma coleção. Diferente da distribuição limitada que ocorre no *parfor* (fatias), essa função possibilita que a distribuição seja feita de diferentes formas em um conjunto de dados.
`gather`	Função que realiza a agregação de uma coleção distribuída.
`drange`	Executa um *loop* entre as partições de uma coleção. Este bloco de código faz com que os *workers* trabalhem simultaneamente em cada distribuição da coleção, mas impede a comunicação entre eles.

Tabela 6 – Descrição das funções utilizadas na paralelização do EbE-FEM

1. Paralelização com granulação mais grossa (decomposição de domínio) de forma a tirar proveito da complexidade de operações disponíveis em cada núcleo;

2. Uso de *threads* ao invés de processos para reduzir a sobrecarga de troca de contexto e concorrência por recursos.

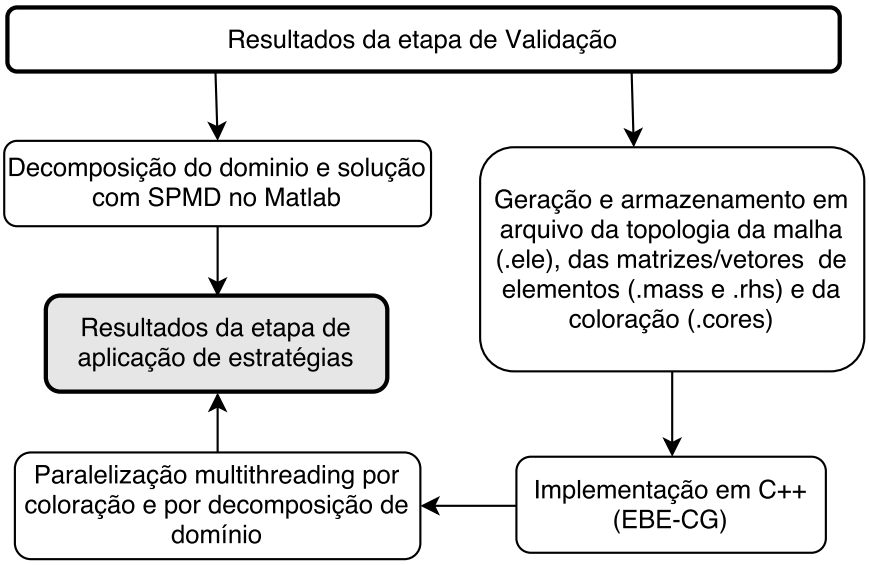
4.2.1 Troca de `parfor` por `spmd`

Como sugere Saad (2003), a paralelização de um algoritmo iterativo pode ser feita de duas formas principais, por meio da paralelização de operações do algoritmo, ou por meio de reestruturação e divisão do problema. Dos algoritmos da PDETool o `parfor` é o representante da primeira categoria e o `spmd` da segunda. Enquanto no primeiro a forma sequencial do problema é mantida, no segundo é necessária a decomposição do problema e a comunicação entre as partes.

Algoritmo 17 Aplicação do *parfor* no algoritmo 15

1: **for** c = 1,2...$|cores|$ **do** ▷ Itera de 1 à quantidade cores da coloração
2: $matElCor = cores(c).matEl$ ▷ Obtém as matrizes dos elementos da cor c
3: $rhsCor = cores(c).rhs$ ▷ Obtém os vetores rhs da cor c
4: $R = reshape(r(rhsCor), 3, [\,])$ ▷ *gather* formando uma matriz de $3 \times |matElCor|$
5: **PARFOR** ▷ Início da paralelização
6: **for** j = 1,2...$|cores(c)|$ **do** ▷ Itera de 1 à quantidade de elementos da cor c
7: $M_e = precond(matElCor(j))$
8: $Z(:) = M_e R(j)$
9: **end for**
10: **END PARFOR**
11: $z(rhsCor) = z(rhsCor) + reshape(Z, 1, [\,])$ ▷ *gather- scatter*
12: **end for**

Figura 26 – Aplicação das novas estratégias de paralelização do EBE-CG

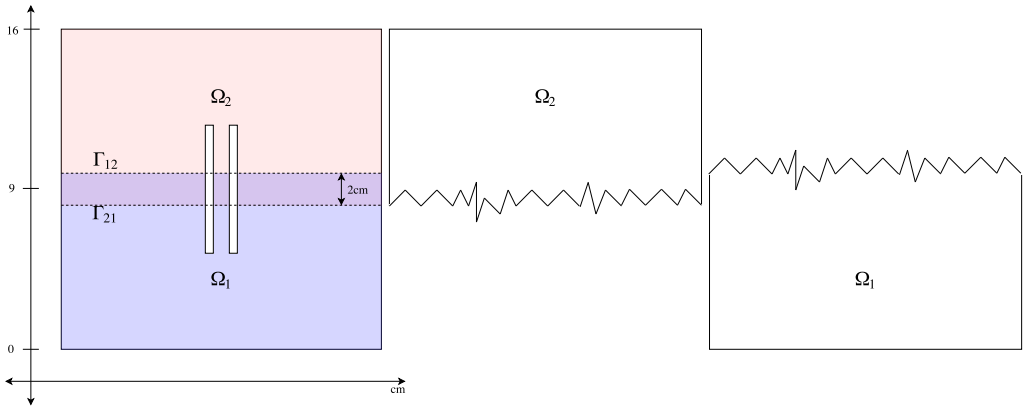


Fonte: Elaborada pelo autor

No `spmd` cada *worker* recebe uma cópia do algoritmo e uma parcela dos dados. O número de partições é restrito visto que a quantidade de *workers* é limitada ao número de núcleos físicos do processador, caso contrário haveria um *overhead* de troca de contexto. A decomposição de domínio de Schwarz, aplicada com sucesso nos trabalhos de Galante (2004) e de PALIN (2007) foi adotada para a distribuição dos dados entre os processos. Embora existam ferramentas adequadas para o particionamento de malha, tais como METIS (KARYPIS; KUMAR, 1998) e SCOTCH (CHEVALIER; PELLEGRINI, 2008), optou-se por utilizar o particionamento cartesiano em faixas horizontais (e/ou verticais), como pode ser visto na figura 27. Esta escolha se deu pela simplicidade do domínio a ser decomposto e pelo fato de as ferramentas mais comumente utilizadas não fornecerem o particionamento com sobreposição, sendo necessário um tratamento pós-partição em cada subdomínio. Para a obtenção das regiões em faixas, são especificados os limites máximos das fronteiras Γ e em seguida é feita a separação dos elementos, de acordo com a localização do seu centro de massa. É importante ressaltar que embora esta estratégia de particionamento forneça um bom balanceamento de

carga, a quantidade de elementos em cada subdomínio não é exatamente igual, uma vez que a malha não é estruturada.

Figura 27 – Exemplo de decomposição de Schwarz em faixas horizontais. Se a coordenada y do centro de massa de um elemento for menor que 8 ele pertence a Ω_1, se for maior que 10, pertence a Ω_2. Se estiver entre 8 e 10 pertence a ambos (região sobreposta)



Fonte: Elaborada pelo autor

O algoritmo 18 contém uma abstração da decomposição e distribuição dos domínios entre os *workers* bem como da comunicação entre os nós de fronteira. Este código é executado paralelamente por 2 ou mais *workers*.

Algoritmo 18 Esquema da decomposição do domínio com JSM, *spmd* e *codistributor*

1: **SPMD** ▷ Início da paralelização
2: $codDados = codistributor(formaDist)$ ▷ Início da configuração de distribuição
3: $dadosDist = codistributed(dadosGlobais, codDados)$
4: $Part = getLocalPart(dadosDist, idWorker)$
5: **for** i = 1,2...$numIterSchwarz$ **do**
6: $Part = cg(Part.A, Part.b, ...)$
7: $[frontAtual, frontNova] = getFronteira(Part, ...)$
8: **if** $dist(frontAtual, frontNova) < TOL$ **then**
9: retorno
10: **end if**
11: $[Part.A, matPart.b] = atribuiContorno(Part, frontNova, ...)$
12: **end for**
13: **END SPMD**
14: $sol = gather(Part.x)$

4.2.2 Troca de *workers* por *threads*

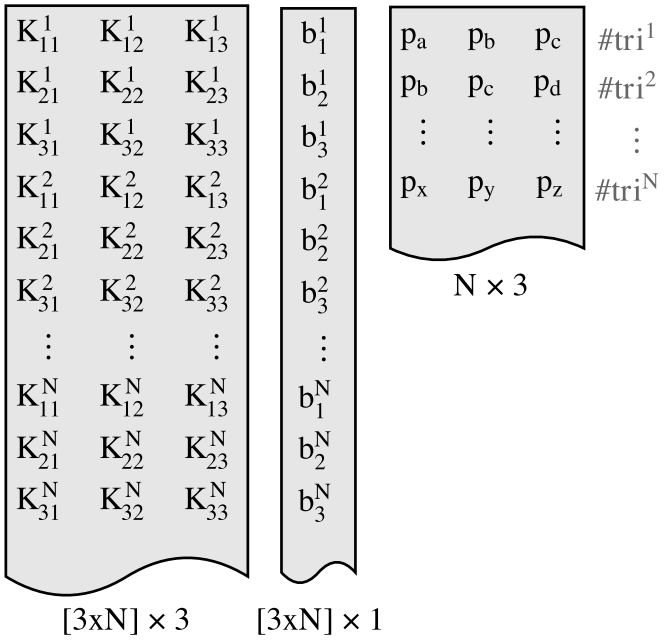
Além do aumento da granulação na paralelização do CG por meio do bloco `spmd` , é realizada uma implementação em C++ a fim de se verificar o comportamento do sistema com o uso de *threads*. De acordo com Hennessy e Patterson (2011), a programação *multithreading* possibilita a troca de contexto praticamente instantânea, enquanto a alternância entre processos pode requerer de centenas a milhares de ciclos de *clock*. Essa característica

possibilita além do paralelismo real, a concorrência entre as *threads* (existência de mais de uma *thread* por núcleo físico), o que pode ser utilizado para ocultar a latências de acesso à memória em alguns casos. Foram realizadas 3 implementações na linguagem C++, sendo a primeira sequencial, a segunda com coloração e a terceira com decomposição de domínio de Jacobi-Schwarz (JSM).

4.2.2.1 EBE-CG sequencial em C++

O primeiro passo para realizar a implementação do EBE-CG fora do ambiente do MATLAB® consiste na geração dos arquivos de texto contendo os dados do FEM e da topologia da malha. Para o algoritmo sequencial são necessárias apenas os dados relativos às matrizes dos elementos K^e, o vetor b^e de cada elemento e a relação dos triângulos da malha. A fim de favorecer o princípio da localidade, como apresentado na seção 3.1.3.3, os dados são estruturados conforme a figura 28 e são armazenados linearmente, no sentido das linhas.

Figura 28 – Estruturas de dados para a implementação do EBE-CG sequencial. Os dados das matrizes dos elementos são armazenados em um arquivo com extensão `.mass`, os vetores *rhs* em um arquivo `.rhs` e os pontos que compõem cada triângulo são armazenados em um arquivo `.ele`.



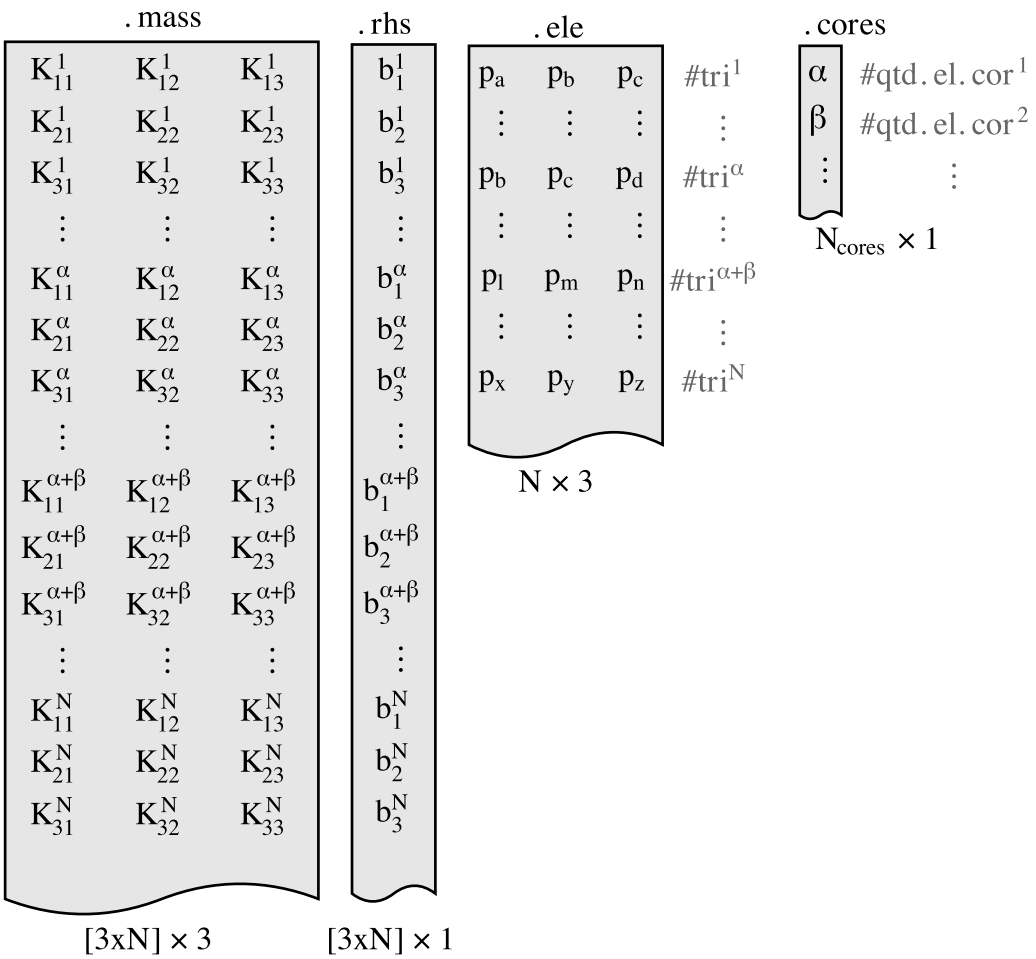
Fonte: Elaborada pelo autor

4.2.2.2 EBE-CG com coloração

O processo de coloração consiste na reordenação das estruturas de dados com extensão `.mass` e `.rhs` (ver figuras 26 e 28) de forma que os dados referentes a elementos da mesma cor estejam armazenados de forma contígua. Para a identificação do limite de cada cor, foi criada uma estrutura de dados adicional (`.cores`) que contém um vetor com a quantidade de elementos de cada cor. Tais modificações são esquematizadas na figura 29. Em cada iteração do CG as matrizes dos elementos de uma determinada cor são divididas entre

uma quantidade N de *threads* e assim executadas paralelamente. O valor de N geralmente é um múltiplo do número de núcleos físicos n, como por exemplo $2n$ ou $3n$. As *threads* em "excesso" podem ser executadas paralelamente em arquiteturas com SMT (Intel *Hyperthreading*, por exemplo) ou concorrentemente em pipelines simples.

Figura 29 – Estruturas de dados para a implementação do EBE-CG com coloração



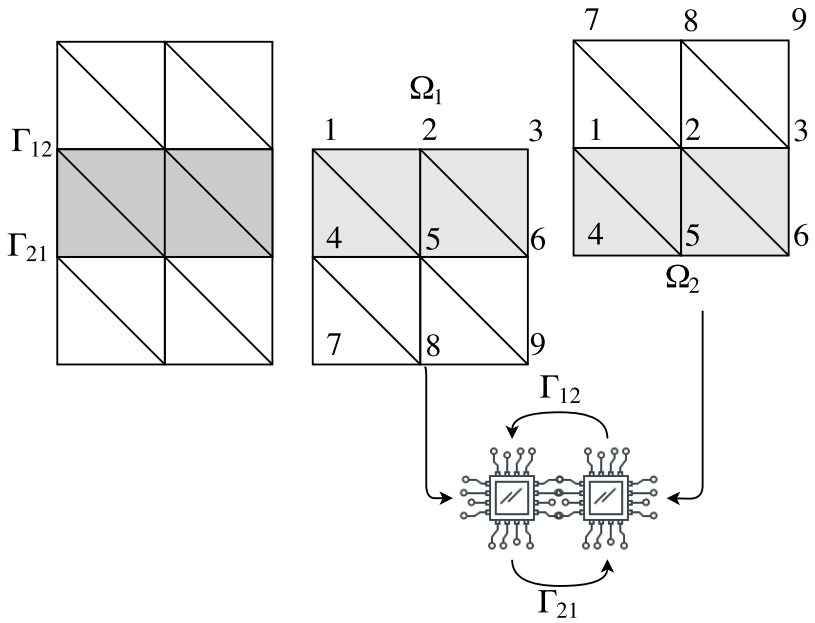
Fonte: Elaborada pelo autor

4.2.2.3 EBE-CG com decomposição de domínio

Uma vez que a decomposição de domínio possibilita a separação de um problema em k subproblemas que podem ser executados paralelamente, as estruturas de dados são as mesmas do algoritmo sequencial: um arquivo `.mass`, um `.rhs` e um `.ele` para cada subdomínio. Adicionalmente, deve-se manter um aquivo `.map` com a relação entre os identificadores dos elementos em cada submalha e os da malha original, apenas para a montagem do resultado global após o processamento. Para facilitar a comunicação entre os subdomínios, os nós das fronteiras Γ são numerados primeiro. Assim sendo, no exemplo da figura 30, os nós de Γ_{12} são numerados em cada subdomínio, em seguida os nós de Γ_{21} em cada sobdomínio, e por fim, os nós não comuns de cada um.

Assim como apresentado nos algoritmos 8 e 18, em cada iteração do método de

Figura 30 – Divisão dos domínios e comunicação entre os nós de fronteira



Fonte: Elaborada pelo autor

Jacobi-Schwarz são executados os passos de: solução do sistema, comunicação das fronteiras, checagem de convergência e atribuição das condições de contorno. Na implementação realizada, cada *thread* salva o valor de sua fronteira interna em um *buffer* para que seja lido por outra *thread*. Cada fronteira possui o seu respectivo *buffer* e cada *buffer* pode ser escrito por uma *thread* e lido por outra. Assim sendo, a sincronização é feita apenas entre pares de *threads*, possibilitando que as demais continuem sua execução.

4.3 Otimizações

Uma vez obtidos os resultados do aumento da granulosidade do problema por meio da decomposição de domínio de Jacobi-Schwarz e a troca de uma solução multiprocessos por uma *multithread* pode-se melhorar o desempenho em termos do tempo de execução e consumo de potência por meio da exploração de outros níveis de paralelismo ou por meio da aceleração da convergência com precondicionadores.

4.3.1 Paralelismo em nível de dados

Ao se trabalhar com dados estruturados é possível se obter vantagem tanto em termos de paralelismo quanto de termos de `cache`, de forma que o código deve ser produzido tendo em mente não apenas a arquitetura do processador, mas também a hierarquia de memória. Para tirar proveito do paralelismo em si, são empregadas no algoritmo do CG as instruções multimídia dos conjuntos `sse` e `avx` por meio das bibliotecas `emmintrin` e `immintrin` da Intel e é feita a eliminação das dependências transportadas por `loop`. Para aproveitar o príncípio de localidade espacial e temporal da memória são adotadas as técnicas de `strip-mining` e

redução de `stride`. Maiores detalhes dessas estratégias são apresentados nas seções 3.1.3.3 e 3.1.3.1.

4.3.2 Paralelismo em nível de instrução

Como foi apresentado na seção 3.1.1 o ILP pode ser explorado principalmente por meio de *flags* de compilação, desdobramento de *loop* e eliminação de dependências de nome. São aplicadas sobre a solução encontrada as duas últimas estratégias, uma vez que estas são independentes da linguagem e do compilador e portanto, portaveis para outras linguagens.

4.3.3 Precondicionamento

A última forma de redução do tempo de execução não tem a ver com a arquitetura do processador ou com a hierarquia de memória, mas sim com a natureza do problema modelado ou até mesmo com a forma de aquisição dos dados (inclusão de ruídos). Embora existam muitos tipos de precondicionadores, conforme apresentado na seção 3.4, a maioria deles pressupõe que o sistema esteja montado (KISS et al., 2012) em sua forma global, o que não acontece quando se trabalha com EBE-FEM. A fim de se ter uma noção do efeito do precondicionamento na convergência é aplicado no algoritmo final o precondicionador de Jacobi. Um eficiente precondicionador (também chamado de EBE) para sistemas desmontados foi introduzido por (HUGHES; LEVIT; WINGET, 1983) e é detalhado nos trabalhos de Dayde, L'Excellent e Gould (1995), Carey et al. (1988) e Levit (1987).

5 Conclusão

“Inteligência é a habilidade de evitar fazer o trabalho,
e mesmo assim conseguir ter o trabalho realizado”.
Linus Torvalds

Neste capítulo são apresentados os resultados das implementções, as limitações encontradas e as possibilidades de trabalhos futuros. Convém neste ponto retomar o objetivo geral, que consiste na investigação e implementação de uma solução paralela pelo Método dos Elementos Finitos (FEM) de um problema de valor de contorno (PVC). Visando atingir este objetivo em um ambiente *multicore*, parte-se do trabalho de Kiss et al. (2012) o qual consiste em uma modificação do método EBE-BiCGStab de forma que as matrizes dos elementos nunca são armazenadas, mas recalculadas a cada iteração. Esta solução se mostrou mais eficiente que o EBE-BiCGStab tradicional para o processamento em GPU, uma vez ela minimiza a movimentação de dados. Ainda segundo o autor, os ganhos de performance e economia no consumo energético só são maximizados quando a solução implementada se adequa à arquitetura do hardware.

5.1 Resultados

O ambiente de execução adotado neste trabalho possui sistema operacional Ubuntu 16.04 LTS, 64 bits e hierarquia de memória composta por uma cache L1 de $32KiB$, uma L2 de $256KiB$, uma L3 de $3MiB$ e memória principal SODIMM DDR3 síncrona de $2GiB$ e $1600MHz$. O processador é um Intel Core $i3-4010U$ com frequência de $1.70GHz$, 2 núcleos físicos e microarquitetura Haswell com extensão do conjunto de instruções para AVX-256 e SSE4. Cada núcleo apresenta suporte à *multithreading* do *hardware* por meio da tecnologia *Hyperthreading* (SMT), sendo 2 linhas de execução para cada.

5.1.1 Resultados da validação do problema

Os parâmetros de referência em termos de precisão e tempo de execução foram obtidos à partir da solução do problema pela função `assempde` da PdeTool e pelas funções de contagem de tempo `tic` e `toc`. Os resultados obtidos são mostrados na tabela 7.

Antes da exploração do paralelismo disponível no ambiente *multicore*, foi feita a verificação do impacto do armazenamento ou não das matrizes dos elementos no tempo de exe-

Qtd. Triângulos	Qtd. Pontos	Resolução	CPU (%)	Tempo (s)	Tam. (MB)
3190	1.700	0,6	30	0,0989	0,9
88.522	44.692	0,1	30	2,1473	9
1.127.292	565.157	0,028	30	46,9111	177,8

Tabela 7 – Parâmetros de referência obtidos com a função `assempde`

Algoritmo	**CPU (%)**	**Tempo (s)**
EBE-CG (Kiss)	30	13,2032
EBE-CG	30	1,9821

Tabela 8 – Comparação entre o EBE-CG sequencial sem e com armazenamento de matrizes. A resolução utilizada é a de $0,6$

Algoritmo	**Resol.**	**Tempo (s)**	**Iter. (CG)**	**CPU (%)**	**Norma Máx.**
EBE-CG	0,1	394,2595	721	25	0,0019
EBE-CG + `parfor`		583,2284	721	65	0,0019

Tabela 9 – Comparação entre a solução sequencial e com coloração

cução. Como pode ser visto na tabela 8, abordagem convencional do EBE-CG (com armazenamento de matrizes) apresentou um tempo de execução consideravelmente menor. Embora a estratégia de não armazenar as matrizes dos elementos tenha se mostrado efetiva quando aplicada em sistemas com GPU (KISS et al., 2012), ela não apresenta ganhos (*speedup*) em uma implementação sequencial ou mesmo numa implementação paralela executada em um sistema com poucos núcleos.

Uma vez verificado que é preferível o armazenamento das matrizes dos elementos, é empregada a paralelização por coloração utilizando processos (*workers*) por meio do bloco de iterações `parfor` da ParTool. A não ser que se diga o contrário, são adotados na paralelização 2 processos ou *threads* e uma acurácia de $1e^{-6}$ na análise de convergência do CG.

Foram realizados testes com as malhas de resolução (tamanho máximo de arestas) $0,6$, $0,1$ e $0,028$. Como foi verificado o mesmo comportamento do *speedup* em cada uma para cada caso de teste e sabendo que a análise da escalabilidade depende apenas do aumento do número de *threads* e do *speedup* gerado (ver seção 3.1.5.2), optou-se por apresentar os resultados relativos a apenas uma das malhas, no caso a de resolução $0,1$.

A norma do máximo (ou norma infinito) fornece a máxima distância entre 2 vetores e é utilizada para comparar a solução obtida com a solução da PdeTool. Nas implementações com decomposição de domínio esta norma tende a ser maior, devido à "costura" gerada entre os domínios ao final da execução. À medida em que são realizadas iterações do JSM esta "costura" tende a ser suavizada. A tabela 9 apresenta os resultados do EBE-CG sequencial e com coloração para a malha de resolução $0,1$. Foram utilizadas 13 cores, 7 com 6810 elementos, 4 com 6809 e 2 com 6808, sendo esta uma distribuição bem balanceada.

Após adaptações no código e na estrutura de dados da coloração foi verificado na documentação da ParTool (MATHWORKS, 2017b), a existência dos comandos `parfor` e `spmd` dentro de *loops* não é recomendada visto que a cada iteração do *loop* externo ocorre troca de mensagens entre o processo *client* e os *workers*. Além disso, a documentação informa que iterações que apresentam operações simples e baixo tempo de execução não devem ser paralelizadas pela *toolbox* devido ao custo adicional de memória e comunicação para se criar e manter os *workers*.

O tempo de execução e o percentual de uso da CPU do algoritmo paralelizado foram

Resol.	Tempo (s)	Iter.	CPU (%)	Norma Máx.	Speedup	Eficiência
0,1	293,225	1	65	1,3789	1,44	0,72

Tabela 10 – Ganhos de performance com o JSM e `spmd`

Resolução	Tempo (s)	Iterações	CPU (%)	Norma Máx.
0,1	13,0292	720	25	0,0289
0,028	741,8785	2296	25	0,0299

Tabela 11 – Implementação sequencial em C++

respectivamente cerca de $1,5$ e $2,6$ vezes maiores que a versão sequencial sendo necessário o uso de outras estratégias.

5.1.2 Resultados da aplicação de estratégias

O excesso de comunicação encontrado na execução com o `parfor` pode ser contornado com o aumento da granulação e/ou com o uso de *threads*.

A viabilidade da primeira estratégia é verificada ainda no ambiente do MATLAB® , por meio do bloco `spmd` da PdeTool associada à decomposição de domínio do JSM, ao invés do uso de coloração. Como é possível criar apenas 2 *workers*, a malha é particionada como apresentado no esquema da figura 27. A cada iteração do método de Jacobi-Schwarz é realizada a troca de informações relativas à fronteira. É dito que o método converge se a distância euclidiana entre o valor anterior de uma fronteira e o valor a ser recebido for menor que $1e^{-2}$. Como o particionamento é feito de forma simétrica, o método converge logo na primeira iteração do JSM, ou seja, o sistema de cada subdomínio é resolvido em paralelo via CG e em seguida as soluções são unificadas, sem a necessidade de um reprocessamento. A tabela 10 contém os dados do tempo de execução para a malha de resolução $0,1$ com sobreposição de $1cm$ $(6,32\%)$.

A aplicação da decomposição de domínio e o uso do bloco `spmd` foram responsáveis por um ganho de velocidade de 34%. O outro caminho investigado é a utilização de *threads* ao invés de processos e para isso é realizada a implementação em C/C++.

Como pode ser observado na tabela 11, o EBE-CG sequencial do MATLAB® é cerca de 30 vezes mais lento que o implementado em C++. Essa discrepância se dá pelo fato de terem sido utilizadas estruturas de dados complexas (`cell` e `struct`) no MATLAB® ao invés de vetores unidimensionais, como foi feito no C++. Ao se utilizar células para armazenar os dados das matrizes dos elementos e dos vetores *rhs* pode perder a contiguidade de memória e assim o benefício do princípio da localidade.

A tabela 12 apresenta o resultado da paralelização com granulação fina em C++ por meio da coloração e a tabela 13 o resultado da paralelização com granulação grossa e decomposição de domínio de Jacobi-Schwarz. Novamente, como o domínio é particionado simetricamente, a convergência ocorre logo na primeira iteração do JSM. Como era de se esperar, o algoritmo com granulação grossa, mais adequado para sistemas MIMD, possui melhor desempenho.

Resol.	Tempo (s)	Iter.(CG)	CPU (%)	Norma Máx.	Speedup	Eficiência
0,1	5,7232	730	40	0,0279	2,28	1,14
0,028	296,45	2296	40	0,0159	2,52	1,26

Tabela 12 – Implementação *multithreading* com coloração em C++

Resol.	Tempo (s)	Iter. (JSM)	CPU (%)	Norma Máx.	Speedup	Eficiência
0,1	2,4425	1	50	0,3123	5,34	2,17
0,028	133,4312	1	65	0,2991	5,56	2,28

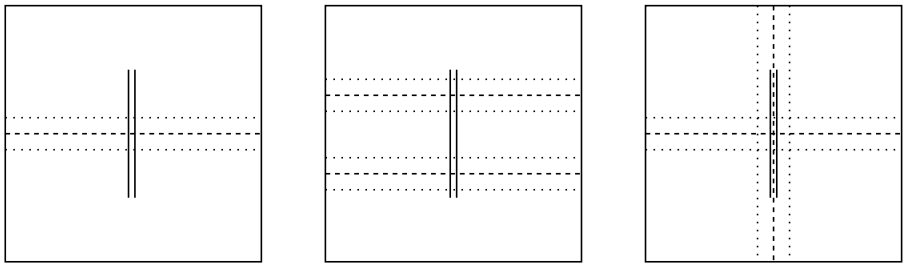
Tabela 13 – Implementação *multithreading* com decomposição de domínio em C++

Nº. Th.	Tempo (s)	Iter. (CG)	CPU (%)	Norma Máx.	Speedup	Eficiência
3	5,3842	729	65	0,0212	2,4198	0,8066
4	5,1218	752	70	0,0252	2,5498	0,6359
5	5,9925	726	80	0,0302	2,1742	0,4348

Tabela 14 – Aumento do número de *threads* no EBE-CG com coloração em C++. Dados referentes à malha de resolução $0,1$.

A extrapolação do número de *threads* é apresentada nas tabelas 14 e 15. Como pode ser visto na primeira, a performance tende a cair à medida em que o número de *threads* ultrapassa o número de núcleos *Hyperthreading* do processador, uma vez que neste caso, deixa de existir o paralelismo real e as *threads* passam a concorrer pela execução. Na paralelização com decomposição de domínio a distribuição adequada dos dados é de extrema importância. Como foi adotado o particionamento da malha orientado pelos eixos do plano cartesiano, pode ocorrer o desbalanceamento na distribuição dos nós de contorno (placas do capacitor), como pode ser visto nas figuras 31 e 32 para o particionamento em 3 subdomínios. Esta má distribuição ocasiona uma lentidão na convergência do método, uma vez podem ser necessárias mais iteraçãos do JSM que o número de *threads*, o que torna a performance do código paralelizado pior que a do sequencial.

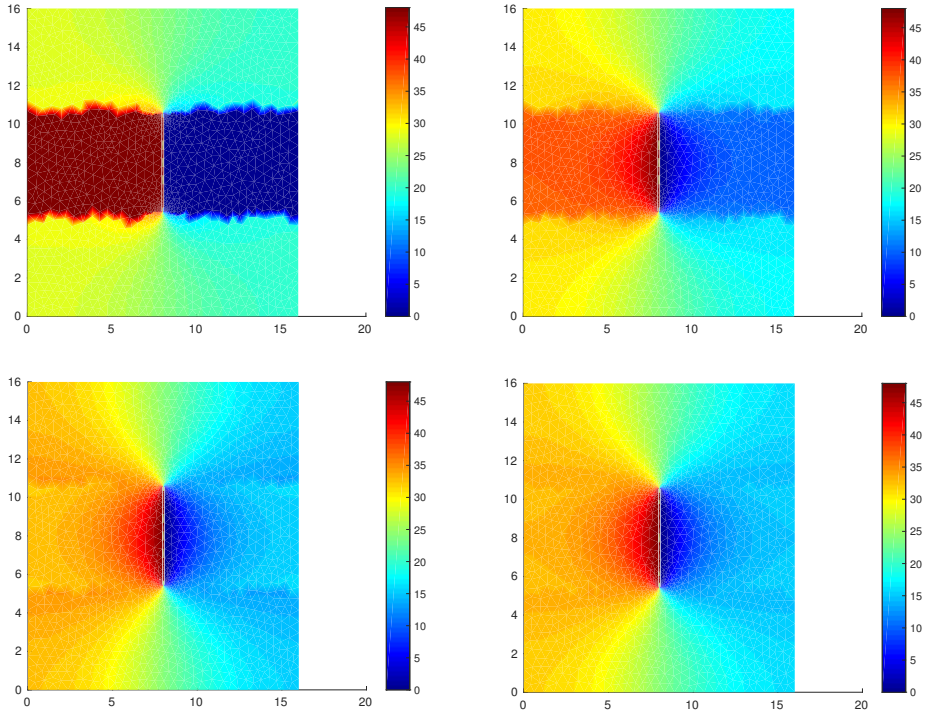
Figura 31 – Partição do domínio em 2, 3 e 4 subdomínios



Fonte: Elaborada pelo autor

Se for considerado o caso em que a decomposição do domínio distribui de forma balanceada as condições de contorno, para que haja pouca ou nenhuma comunicação entre os domínios, o problema pode ser considerado fortemente escalável (ver seção 3.1.5.2), uma vez que à medida em que se aumenta o número de *threads* para um tamanho fixo do problema, o tempo de execução tende a diminuir e a eficiência a se manter.

Figura 32 – Imagem da 1^a, 10^a, 19^a e 28^a iteração do JSM para 3 domínios. A cada iteração do JSM os sistemas de cada subdomínio são solucionados via CG. Devido à má distribuição dos valores de contorno, ocorre um excesso de comunicação, sendo necessárias muitas iterações do JSM para se obter uma solução satisfatória



Fonte: Elaborada pelo autor

Nº. Th.	**Tempo (s)**	**Iter. (JSM)**	**CPU (%)**	**Norma Máx.**	**Speedup**	**Eficiência**
3	15,94	30	70	0,3125	-	-
4	1,5252	1	100	0,4852	8,6060	2,1515

Tabela 15 – Aumento do número de *threads* no EBE-CG com o JSM em C++. Dados referentes à malha de resolução $0,1$.

5.1.3 Aplicação de otimizações

Embora o uso de *threads* e processos seja a forma mais comum de se explorar o paralelismo do hardware, existem ainda os paralelismos em nível de instrução (ILP) e em nível de dados (DLP), os quais podem ser explorados sem se alterar a estrutura sequencial do programa. Para fazer uso do ILP o código sequencial foi otimizado por meio do desdobramento de *loops* e renomeação de variáveis. O DLP é explorado utilizando-se explicitamente as instruções multimídia dos conjuntos `sse` e `avx` à partir das bibliotecas `emmintrin` e `immintrin` da Intel. A tabela 16 contém os resultados da otimização do algorimo sequencial.

Ao se comparar os resultados da tabela 16 com os das tabelas 12 e 14 é possível notar que os ganhos de performance utilizando-se apenas otimizações de código são maiores que os ganhos com o uso de *multithreading* e coloração. Além da execução mais rápida, o código otimizado apresenta maior eficiência energética, pois utiliza apenas 30% da CPU, enquando o EBE-CG com coloração utiliza de 50 a 70%. Embora a redução no consumo de potência em

Otim.	Tempo (s)	Iter. (CG)	CPU (%)	Norma Máx.	Speedup	Eficiência
ILP	4,9051	720	30	0,0281	2,6563	2,6563
DLP	4,6532	720	30	0,0290	2,8001	2,8001

Tabela 16 – Performance das otimizações aplicadas ao código sequencial. Dados referentes à malha de resolução $0,1$.

Tempo (s)	Iter. (JSM)	CPU (%)	Norma Máx.	Speedup	Eficiência
2,2101	1	50	0,3128	5,8955	2,9477

Tabela 17 – Associação entre TLP, DLP e ILP de forma a maximizar a eficiência energética. Dados referentes à malha de resolução $0,1$.

Sobrep. (cm; %)	Tempo (s)	CPU (%)	Norma Máx.	Speedup	Eficiência
0,4; 2,5	2,0935	50	0,9828	6,2236	3,1118
0,2 ;1,24	2,0563	50	0,9125	6,3362	3,1681

Tabela 18 – Redução da sobreposição e uso do precondicionador de Jacobi. Dados referentes à malha de resolução $0,1$.

detrimento do tempo de execução seja uma forma de subutilização de recursos, ela é uma característica desejável para a economia no consumo no consumo de energia em *clusters* bem como para o aumento da duração de baterias em dispositivos móveis (YANG; CHENG, 2015).

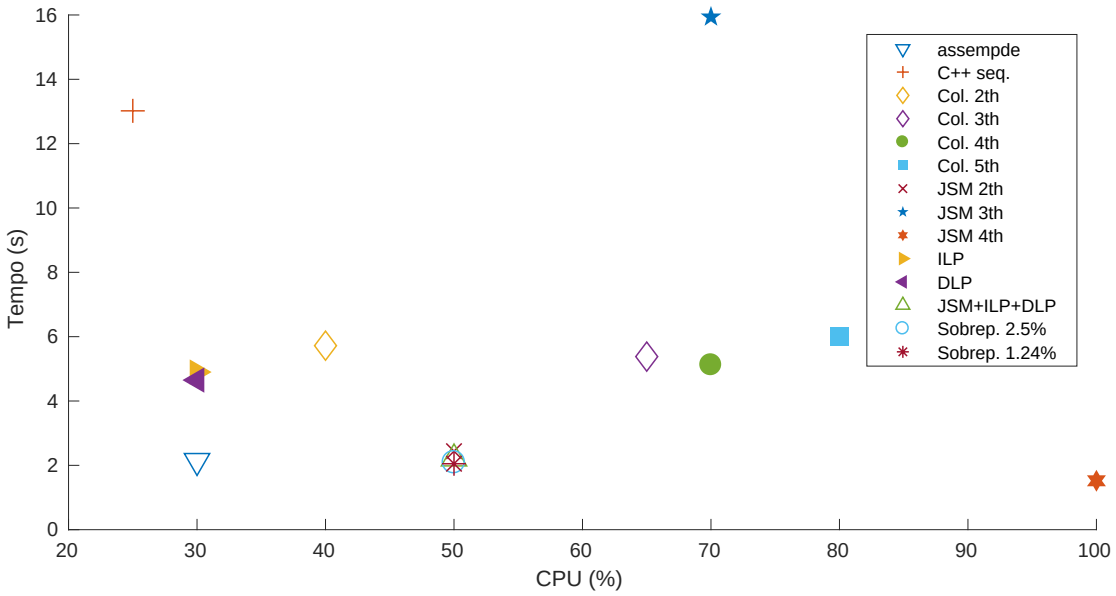
Considerando que a eficiência energética pode ser vista como a razão apresentada na equação 5.1, a decomposição em 2 subdomínios é mais eficiente em termos de velocidade e economia de energia que a de 4 subdomínios. Enquanto para 2 subdomínios a razão é igual a $10,68$, para 4 ela é igual a $8,6$. Ao se aplicar as otimizações de ILP e DLP ao algoritmo de decomposição de domínio obtém-se os resultados da tabela 17. O valor de eficiência energética para este caso é igual a $11,8$.

$$Efic.\ Energética = \frac{Speedup \times 100}{CPU} \tag{5.1}$$

As últimas estratégias adotadas para o ganho de performance consistem na redução da região de sobreposição e na aplicação de um precondicionador, de forma a reduzir o número de iterações necessárias no CG. A região de sobreposição utilizada nos resultados é de $1cm$ e possui $6,32\%$ dos elementos da malha de resolução $0,1$ e $6,22\%$ da de resolução $0,028$. Os resultados do uso do precondicionador de Jacobi aplicado junto à redução da região de sobreposição são apresentados na tabela 18. O precondicionador de Jacobi se mostrou pouco efetivo, com redução de aproximadamente 7% do número de iterações do CG e com a inclusão de maior erro na aproximação.

Uma compilação dos resultados para malha de resolução $0,1$ é apresentada na figura 33. Como os eixos representam o percentual de CPU e o tempo de execução, os melhores resultados (melhor eficiência energética) estão mais próximos da origem. A figura 34 contém os resultados obtidos na implementação. O valor obtido para o campo elétrico E foi de $60,45kV/m$ próximo ao valor da solução exata $60,5kV/m$ conforme apresentado no exemplo

Figura 33 – Dados das execuções para a malha 0.1. Quanto mais próximo da origem, maior a eficiência energética

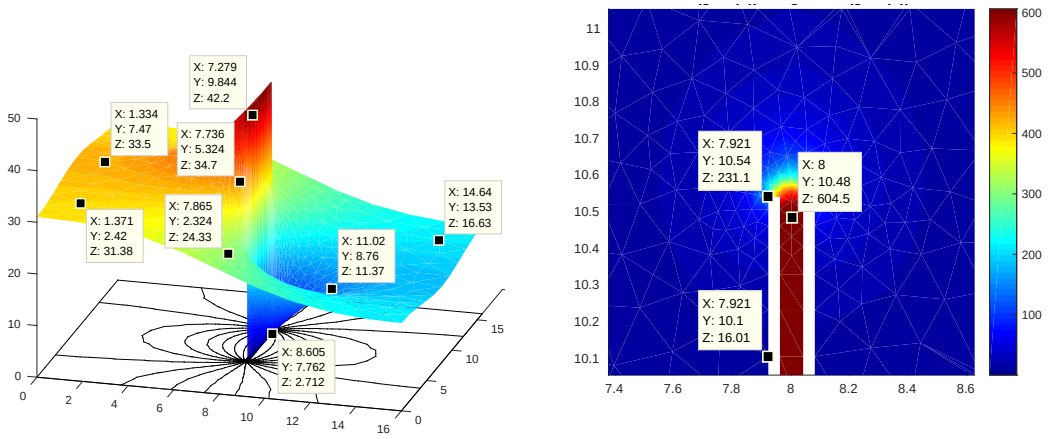


Fonte: Elaborada pelo autor

Figura 34 – Resultados obtidos na solução paralela do PVC

(a) Valores da distribuição do potencial

(b) Valores do campo elétrico



Fonte: Elaborada pelo autor

10.4 de Boylestad (2011).

5.2 Considerações e limitações

Retomando os objetivos iniciais do trabalho, foram apresentados neste capítulo os resultados experimentais da exploração dos níveis de paralelismo de um computador multiprocessado. No capítulo 4 foi demonstrado o processo de paralelização do algoritmo do FEM por meio da estratégia EBE associada à coloração da malha e à decomposição de domínio de Jacobi-Schwarz. Como mostra a figura 33, se tratando de eficiência energética (boa relação

entre tempo de execução e uso da CPU), tanto o algoritmo sequencial com DLP (paralelização do tipo SIMD) quanto as versões que utilizam decomposição pelo JSM com 2 *threads* apresentaram resultados satisfatórios. Levando em conta apenas o tempo de execução, o JSM com 2 e 4 *threads* apresentou melhor performance até mesmo que a função `assempde` do MATLAB®. Em relação à acurácia, o método com DLP apresentou melhores resultados, visto que não é preciso particionar a malha e então "costurá-la" novamente. A maior precisão nos resultados do JSM e consequentemente a suavização das fronteiras pode ser obtida diminuindo-se o valor da tolerância na comparação das fronteiras e assim iterando mais de uma vez.

A Principal limitação do trabalho está imprecisão de alguns resultados, visto que a cada execução é retornado um valor ligeiramente diferente do anterior, sobretudo devido ao sistema de cache ou à execução simultânea de outros programas, como por exemplo, o editor de texto e a IDE. Para a malha com resolução de $0,1$ foi tomada a média de 3 execuções, mas nenhum cálculo de estatística (como por exemplo o desvio padrão) foi realizado. Para malhas mais densas, como é o caso da resolução $0,028$ a variação em cada execução foi menos significativa. Além disso, o percentual de uso da CPU varia ao longo de uma execução, sendo utilizado para os cálculos apenas o máximo valor identificado.

Uma segunda limitação é a restrição da solução para a clase de problemas positivos definidos (elipticos). A solução de problemas hiperbólicos e parabólicos pode ser feita por meio da implementação do BiCGStab.

A última limitação consiste no fato de que os resultados foram obtidos utilizando-se um ambiente *multicore* (único chip com múltiplos núcleos) e não um sistema multiprocessado genérico (com mais de 1 chip). Esta escolha se justifica pela tentativa de se obter resultados próximos aos que seriam observados no processamento em dispositivos móveis, tais como celulares e tablets.

5.3 Trabalhos futuros

Uma vez que os resultados deste trabalho foram satisfatórios abre-se espaço para que este possa ser utilizado como referência para trabalhos futuros, tais como os que seguem:

- Implementação e paralelização da geração de malha e das etapas de pré-processamento e pós-processamento do FEM;
- Aplicação da solução implementada à outras linguagens de programação nativamente paralelas, tais como as citadas na seção 3.1.2.2;
- Associação com o MPI do algoritmo do EBE-CG proposto, de forma a expandir a solução para sistemas de memória distribuída;
- Utilização de precondicionadores mais eficientes que o de Jacobi e expansão da solução para resolver outras classes de PDE, como por exemplo as com dimensão temporal;
- Investigar a paralelização de outros métodos de solução de PDE como por exemplo os métodos sem malha e o FEM com sobreposição;

- Investigar otimizações para o JSM ou ainda novas estratégias de decomposição de domínio como por exemplo a FETI;
- Investigar o impacto da exploração do paralelismo da CPU em sistemas heterogêneos.

Referências

AHAMED, A.-K. C.; MAGOULÈS, F. Conjugate gradient method with graphics processing unit acceleration: CUDA vs OpenCL. *Advances in Engineering Software*, v. 47, n. 1, p. 164–169, 2016. ISSN 0965-9978. Disponível em: <http://dx.doi.org/10.1016/j.advengsoft.2011.12.013>. Citado nas páginas 15, 20 e 48.

AL., A. J. et. *Concurrency*. 2017. Acesso em 03/11/2017. Disponível em: <https://elixirschool.com/en/lessons/advanced/concurrency/>. Citado na página 30.

AMD. *Produtos AMD*. 2017. <http://www.amd.com/pt-br/products>. Acesso em: 19/07/2017. Citado na página 15.

ANZT, H. et al. Preconditioned Krylov solvers on GPUs. *Parallel Computing*, Elsevier B.V., v. 0, p. 1–13, 2016. ISSN 01678191. Disponível em: <http://dx.doi.org/10.1016/j.parco.2017.05.006>. Citado nas páginas 15 e 48.

ARMKEIL. *Loop unrolling in C code*. 2015. Disponível em: <http://www.keil.com/support/man/docs/armcc/armcc_chr1359124222660.htm>. Citado na página 27.

BANCILA, M. *C++11 threads, locks and condition variables*. 2013. Acesso em 01/11/2017. Disponível em: <https://www.codeproject.com/Articles/598695/Cplusplus-threads-locks-and-condition-variables>. Citado na página 29.

BARNEY, B. *OpenMP*. Lawrence Livermore National Laboratory, 2017. Disponível em: <https://computing.llnl.gov/tutorials/openMP/>. Citado na página 29.

BARNEY, B. *POSIX Threads Programming*. Lawrence Livermore National Laboratory, 2017. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/>. Citado na página 29.

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 10, 47, 48, 49, 54, 56, 57, 63 e 65.

BLANDY, J. *Why Rust?* [S.l.]: O'Reilly Media, 2015. Citado na página 30.

BOEHMER, S. et al. Numerical simulation of electrical machines by means of a hybrid parallelisation using MPI and OpenMP for finite-element method. *IET Science, Measurement & Technology*, v. 6, n. 5, p. 339, 2011. ISSN 17518822. Disponível em: <http://digital-library.theiet.org/content/journals/10.1049/iet-smt.2011.0126>. Citado na página 20.

BOYCE, W. E.; DIPRIMA, R. C. *Equacões Diferenciais Elementares e Problemas de Valores de Contorno*. [S.l.]: LTC, 2010. ISBN 978521617563. Citado na página 37.

BOYLESTAD, R. L. *Análise de Circuitos*. [S.l.]: Pearson, 2011. ISBN 978-85-64574-20-5. Citado nas páginas 17, 60 e 82.

CAREY, G. F. et al. Element-by-element vector and parallel computations. *Communications in Applied Numerical Methods*, v. 4, n. 3, p. 299–307, 1988. ISSN 0748-8025. Disponível em: <http://doi.wiley.com/10.1002/cnm.1630040303>. Citado nas páginas 20, 59 e 75.

CHEVALIER, C.; PELLEGRINI, F. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 34, n. 6-8, p. 318–331, jul. 2008. ISSN 0167-8191. Disponível em: <http://dx.doi.org/10.1016/j.parco.2007.12.001>. Citado na página 70.

CHOU, C. Y.; CHEN, K. T. Performance Evaluations of Different Parallel Programming Paradigms for Pennes Bioheat Equations and Navier-Stokes Equations. p. 503–508, 2016. Citado na página 21.

CPLUSPLUS. *The Features of C++ as a Language*. 2017. Acesso em 01/11/2017. Disponível em: <http://www.cplusplus.com/info/description/>. Citado na página 29.

DAYDE, M. J.; L'EXCELLENT, J.-Y.; GOULD, N. I. M. On the Use of Element-by-Element Preconditioners to Solve Large Scale Partially Separable Optimization. *Computing*, 1995. Citado nas páginas 15, 20, 59 e 75.

DESAI, J. A. C. S. *Introduction to the Finite Element Method: A Numerical Method for Engineering Analysis*. [S.l.]: Van Nostrand Reinhold Company, 1972. Citado nas páginas 36, 38, 39 e 41.

DEVELOPERS, T. R. P. *Concurrency*. 2011. Acesso em 03/11/2017. Disponível em: <https://doc.rust-lang.org/book/first-edition/concurrency.html>. Citado na página 30.

DOLEAN, V.; JOLIVET, P. An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation. 2016. Citado na página 58.

DOLWITHAYAKUL, B.; CHANTRAPORNCHAI, C.; CHUMCHOB, N. An efficient asynchronous approach for Gauss-Seidel iterative solver for FDM/FEM equations on multi-core processors. *JCSSE 2012 - 9th International Joint Conference on Computer Science and Software Engineering*, p. 357–361, 2012. Citado na página 63.

EVANS, L. C. *Partial Differential Equations*. 19. ed. [S.l.]: American Mathematical Society, 1998. Citado na página 44.

FAIRES, R. L. B. J. D. *Análise Numérica*. [S.l.]: CENGAGE Learning, 2008. ISBN 978-85-221-0601-1. Citado nas páginas 10, 15, 38, 47, 48, 49, 53 e 55.

FRANCO, N. B. *Cálculo numérico*. [S.l.]: Prentice Hall Brasil, 2006. ISBN 8576050870, 9788576050872. Citado nas páginas 10, 47, 48, 49, 50, 51 e 53.

FURTADO, M. *Notas de EDP2*. Brasilia: [s.n.], 2012. Disponível em: <http://www.mat.unb.br/~furtado/homepage/notas-edp2.pdf>. Citado na página 44.

GALANTE, G. Métodos de decomposição de domínios para a solução paralela de sistemas de equações lineares. Acesso em 05/11/2017. 2004. Citado na página 70.

GUO, X. et al. Developing a scalable hybrid MPI/OpenMP unstructured finite element model. *Computers and Fluids*, Elsevier Ltd, v. 110, p. 227–234, 2014. ISSN 00457930. Disponível em: <http://dx.doi.org/10.1016/j.compfluid.2014.09.007>. Citado na página 16.

HE, L. et al. 2-D electromagnetic modelling by finite element method on GPU. v. 127, p. 9026–9036, 2016. Citado na página 15.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728. Citado nas páginas 15, 23, 24, 25, 26, 27, 28, 30, 31, 32, 33, 34 e 71.

HU, J.; QUIGLEY, S. F.; CHAN, A. An element-by-element preconditioned conjugate gradient solver of 3d tetrahedral finite elements on an FPGA coprocessor. *Proceedings - 2008 International Conference on Field Programmable Logic and Applications, FPL*, p. 575–578, 2008. Citado nas páginas 20 e 21.

HUEBNER, K. H. et al. *The Finite Element Method for Engineers*. [S.l.]: John Wiley & sons, 2001. ISBN 978-0-471-37078-9. Citado nas páginas 38 e 41.

HUGHES, T.; LEVIT, I.; WINGET, J. Element-by-element implicit algorithms for heat conduction. *Journal of Engineering Mechanics*, v. 109, n. 2, p. 576–585, 1983. ISSN 07339399. Citado nas páginas 21 e 75.

IBM. *Processador Power9 da IBM*. 2016. <http://www.ibmsystemsmag.com/power/businessstrategy/competitiveadvantage/POWER9-Plans/>. Acesso em: 19/07/2017. Citado na página 15.

INTEL. *Intel Streaming SIMD Extensions tecnologia*. 2017. Acesso em 05/11/2017. Disponível em: <https://www.intel.com.br/content/www/br/pt/support/articles/000005779/processors.html>. Citado na página 31.

INTEL. *Processadores Intel*. 2017. <https://www.intel.com/content/www/us/en/products/processors.html>. Acesso em: 19/07/2017. Citado na página 15.

IWASHITA, T. et al. Software framework for parallel BEM analyses with H-matrices. *IEEE CEFC 2016 - 17th Biennial Conference on Electromagnetic Field Computation*, Elsevier B.V., v. 108, n. June, p. 2200–2209, 2017. ISSN 18770509. Disponível em: <http://dx.doi.org/10.1016/j.procs.2017.05.263>. Citado nas páginas 15 e 16.

JIN, J. *The Finite Element Method in Electromagnetics*. [S.l.]: John Wiley & sons, 2002. ISBN 0471438189. Citado nas páginas 36, 37, 38, 39, 40, 41, 42, 47 e 50.

KAPUSTA, P. et al. Acceleration of image reconstruction in 3D Electrical Capacitance Tomography in heterogeneous, multi-GPU system using sparse matrix computations and Finite Element Method. 2016. Citado na página 16.

KARYPIS, G.; KUMAR, V. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. [S.l.], 1998. Citado na página 70.

KISS, I. et al. High locality and increased intra-node parallelism for solving finite element models on GPUs by novel element-by-element implementation. In: *2012 IEEE Conference on High Performance Extreme Computing, HPEC 2012*. [S.l.: s.n.], 2012. ISBN 9781467315760. Citado nas páginas 15, 16, 18, 20, 21, 45, 57, 58, 59, 63, 66, 75, 76 e 77.

LAWLOR, O. *Memory Speed and 2D Arrays*. 2006. Acesso em 05/11/2017. Disponível em: <https://www.cs.uaf.edu/2006/fall/cs301/lecture/11_01_memory.html>. Citado na página 33.

LEVIT, I. Element by element solvers of order n. v. 27, n. 3, p. 357–360, 1987. Citado nas páginas 20, 58 e 75.

MATHWORKS. *Documentation: mldivide*. 2017. Acesso em: 03/08/2017. Disponível em: <https://www.mathworks.com/help/matlab/ref/mldivide.html>. Citado na página 63.

MATHWORKS. *Nested parfor loops and for loops*. 2017. <https://www.mathworks.com/help/distcomp/nested-parfor-loops-and-for-loops.html>. Acesso em: 03/08/2017. Citado na página 77.

MATHWORKS. *Perform parallel computations on multicore computers, GPUs, and computer clusters*. 2017. <https://www.mathworks.com/products/parallel-computing.html>. Acesso em: 03/08/2017. Citado na página 67.

NVIDIA. *Produtos NVIDIA*. 2017. <http://www.nvidia.com.br/page/products.html>. Acesso em: 19/07/2017. Citado na página 15.

ORACLE. *Interface Runnable*. 2017. Acesso em 01/11/2017. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>. Citado na página 29.

ORACLE. *Lesson: Concurrency*. 2017. Acesso em 01/11/2017. Disponível em: <https://docs.oracle.com/javase/tutorial/essential/concurrency/>. Citado na página 29.

PACHECO, P. *An Introduction to Parallel Programming*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123742605. Citado nas páginas 16, 24, 25, 26, 27, 28, 29, 30, 34, 35 e 36.

PALIN, M. F. Técnicas de decomposição de domınio em computação paralela para simulação de campos eletromagnéticos pelo método dos elementos finitos. Acesso em 05/11/2017. 2007. Citado na página 70.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface*. 4th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123744938, 9780123744937. Citado nas páginas 16, 23, 24, 25, 26, 27, 28, 30, 31, 32, 35 e 36.

PLATAFORMATEC. *Processes*. 2017. Acesso em 03/11/2017. Disponível em: <https://elixir-lang.org/getting-started/processes.html>. Citado na página 30.

POWERS, D. L. *Boundary Value Problems and Partial Differential Equations*. [S.l.]: Elsevier, 2006. ISBN 139780125637381. Citado na página 37.

REDDY, J. N. *An Introduction to the Finite Element Method*. [S.l.]: McGraw Hill, 2006. ISBN 0071244735. Citado nas páginas 36 e 45.

REINDERS, J. *Intel AVX-512 Instructions*. 2017. Acesso em 05/11/2017. Disponível em: <https://software.intel.com/en-us/blogs/2013/avx-512-instructions>. Citado na página 31.

SAAD, Y. *Iterative Methods for Sparse Linear Systems*. 2nd. ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. ISBN 0898715342. Citado nas páginas 21, 52, 53, 56, 57, 58, 59 e 69.

SADIKU, M. *Elementos de Eletromagnetismo*. Bookman, 2004. ISBN 9788536302751. Disponível em: <https://books.google.com.br/books?id=qUCCGQAACAAJ>. Citado na página 17.

SADIKU, M. N. O. *Numerical Techniques in Electromagnetics*. [S.l.]: CRC Press, 2001. ISBN 0-8493-1395-3. Citado nas páginas 38, 40, 50 e 90.

SHEWCHUK, J. R. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Pittsburgh: School of Computer Science Carnegie Mellon University, 1994. Citado nas páginas 50, 51, 52, 53, 54, 55 e 57.

STRANG, G. Piecewise Polynomials and the Finite Element Method. *Bulletin of the American Mathematical Society*, v. 79, n. 6, p. 1128–1137, 1973. ISSN 0002-9904. Citado na página 42.

STRANG, G. *Álgebra Linear e suas aplicações*. Massachusets: Cengage Learning, 2013. Citado nas páginas 47, 48, 49, 50, 52, 53, 54, 55 e 56.

SUITESPARSE. *Sparse matrix algorithms and software*. 2017. Acesso em 05/11/2017. Disponível em: <http://faculty.cse.tamu.edu/davis/research.html>. Citado na página 63.

TANENBAUM, A. S.; AUSTIN, T. *Structured computer organization*. 6th ed. ed. [S.l.]: Pearson Education, 2012. Citado nas páginas 23, 25, 26, 27, 28, 30, 32, 33, 34 e 35.

VOLAKIS, J. L.; CHATTERJEE, A.; KEMPEL, L. C. *Finite Element Method for Electromagnetics*. [S.l.]: John Wiley & sons, 1998. ISBN 0780334256. Citado nas páginas 17, 42, 44, 47 e 57.

WATHEN, A. J. An analysis of some element-by-element techniques. *Computer Methods in Applied Mechanics and Engineering*, v. 74, n. 3, p. 271–287, 1989. ISSN 00457825. Citado nas páginas 16, 59 e 66.

WU, D. et al. GPU Acceleration of EBE Method for 3-D Linear Steady Eddy Current Field. 2015. Citado nas páginas 16, 21, 22, 45, 59 e 65.

XU, J.; YIN, W. Y.; MAO, J. Capacitance extraction of high-density 3D interconnects using finite element method. *Asia-Pacific Microwave Conference Proceedings, APMC*, v. 2, p. 3–5, 2005. Citado nas páginas 21 e 64.

YAN, X. et al. Research on Preconditioned Conjugate Gradient Method Based on EBE-FEM and the Application in Electromagnetic Field Analysis. v. 53, n. 6, 2017. Disponível em: <http://www.ieee.org/publications{_}standards/publications/rights/index.h>. Citado nas páginas 21, 59 e 65.

YANG, B.; LIU, H.; CHEN, Z. Preconditioned GMRES solver on multiple-GPU architecture. *Computers and Mathematics with Applications*, Elsevier Ltd, v. 72, n. 4, p. 1076–1095, 2016. ISSN 08981221. Disponível em: <http://dx.doi.org/10.1016/j.camwa.2016.06.027>. Citado nas páginas 48 e 63.

YANG, X.; CHENG, K. T. *Scalable Augmented Reality on Mobile Devices: Applications, Challenges, Methods, and Softwarey*. 2th. ed. San Francisco, CA, USA: CRC Press, 2015. Citado nas páginas 16 e 81.

YAO, L. et al. Parallel implementation and performance comparison of BiCGStab for massive sparse linear system of equations on GPU libraries. *Proceedings - 2015 IEEE 12th International Conference on Ubiquitous Intelligence and Computing, 2015 IEEE 12th International Conference on Advanced and Trusted Computing, 2015 IEEE 15th International Conference on Scalable Computing and Communications, 20*, p. 603–608, 2015. Citado nas páginas 15 e 21.

ZIENKIEWICZ, R. L. T. J. Z. C. *The Finite Element Method its Basis & Fundamentals*. [S.l.]: Elsevier, 2005. ISBN 0750663200. Citado nas páginas 36, 38, 39, 40, 42, 44, 46 e 50.

APÊNDICE A – Método dos elementos finitos em 2 dimensões

Neste apêndice é apresentada a solução de um modelo de elementos finitos. O exemplo desenvolvido foi adaptado do exemplo 6.1 de Sadiku (2001). Deseja-se saber o valor do potencial elétrico nos pontos a e b da placa submetida a uma diferença de potencial de $10V$, como mostra a figura 35a.

O primeiro passo consiste em realizar a triangulação da malha. Uma possibilidade de malha é apresentada na figura 35b. A estrutura de dados tri contém a informação sobre quais vértices formam cada elemento enquanto a estrutura nos possui as coordenadas de todos os vértices da geometria do problema.

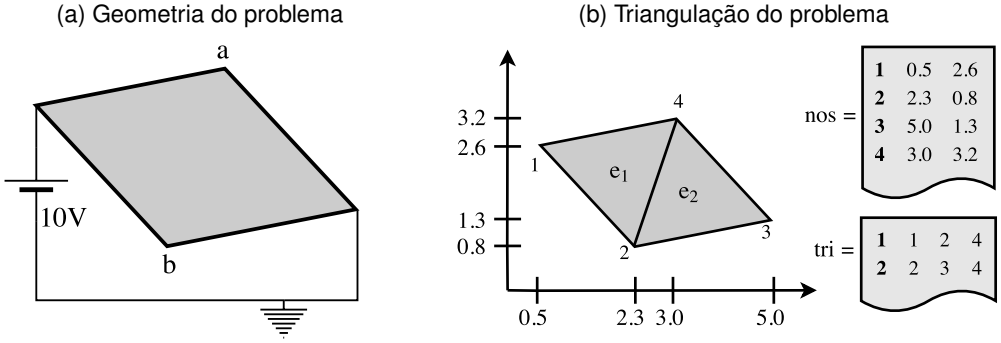
O passo seguinte é a definição da função de interpolação. A aproximação da distribuição do potencial num dado elemento pode ser feita por uma função linear, como mostra a equação A.1. Desta forma, os valores de V em cada nó podem ser obtidos pelo sistema de equações A.2

$$\tilde{V^e} = a^e + b^e x + c^e y \tag{A.1}$$

$$\begin{Bmatrix} \tilde{V_1^e} \\ \tilde{V_2^e} \\ \tilde{V_3^e} \end{Bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{Bmatrix} a \\ b \\ c \end{Bmatrix} \tag{A.2}$$

Deseja-se então resolver o problema encontrando os valores dos coeficientes a, b e c que fazem a aproximação de V. Dada a simplicidade o sistema ele pode ser resolvido invertendo-se a matriz por meio do cálculo dos determinantes e cofatores. Os valores da trans-

Figura 35 – Exemplo do FEM em 2 dimensões



Fonte: Elaborada pelo autor

posta da matriz de cofatores são dados na equação A.3 e o determinante na equação A.4. A equação A.5 contém a substituição na equação A.1 dos valores dos coeficientes calculados.

$$cof(A^e)^T = \begin{bmatrix} (x_2y_3 - x_3y_2) & (x_3y_1 - x_1y_3) & (x_1y_2 - x_2y_1) \\ (y_2 - y_3) & (y_3 - y_1) & (y_1 - y_2) \\ (x_3 - x_2) & (x - 1 - x_3) & (x_2 - x_1) \end{bmatrix} \tag{A.3}$$

$$det(A^e) = (x_1y_2 - x_2y_1) + (x_3y_1 - x_1y_3) + (x_2y_3 - x_3y_2) \tag{A.4}$$

$$\tilde{V^e} = \begin{Bmatrix} 1 & x & y \end{Bmatrix} \begin{Bmatrix} a \\ b \\ b \end{Bmatrix} = \begin{Bmatrix} 1 & x & y \end{Bmatrix} \frac{cof(A^e)^T}{det(A^e)} \begin{Bmatrix} \tilde{V_1^e} \\ \tilde{V_2^e} \\ \tilde{V_3^e} \end{Bmatrix} = \sum_{j=1}^{3} N_j^e(x,y)\tilde{V_j^e} \tag{A.5}$$

Obtidas as funções de interpolação N^e, a próxima etapa consiste na montagem do sistema de equações do problema, começando pela definição das matrizes elementares. A distribuição do potencial é modelada pela equação de Laplace $\nabla^2 V = 0$. Substituindo esta equação em 3.15 e adotando-se o método de Galerkin, tem-se a integral A.6.

$$\iint N_i^e \nabla^2 V^e dxdy = R_i^e \qquad i = 1, 2, 3. \tag{A.6}$$

Aplicando a integração por partes fazendo $u = N$ e $dv = \nabla^2 V$, tem-se como resultado da relação $\int udv = uv - \int vdu$ a equação A.7.

$$\iint \nabla V^e \nabla N_i^e dxdy = R_i^e \qquad i = 1, 2, 3. \tag{A.7}$$

Substituindo V^e por sua aproximação obtida em (A.5) obtém-se a fórmula da matriz elementar nas equações A.8 e A.9.

$$\iint \sum_{j=1}^{3} \nabla N_j^e \tilde{V_j^e} \nabla N_i^e dxdy = \sum_{j=1}^{3} \iint \left(\nabla N_i^e \nabla N_j^e\right) \tilde{V_j^e} dxdy = R_i^e \qquad i = 1, 2, 3. \tag{A.8}$$

$$K_{ij}^e = \iint \left(\nabla N_i^e \nabla N_j^e\right) dxdy \tag{A.9}$$

De forma a agilizar os cálculos das matrizes elementares define-se na equação A.10 a partir dos elementos das equações A.3 e A.4 as variáveis P, Q e A. P e Q são obtidas a partir das simplificações nos cálculos e A é a área do triângulo. A equação A.11 mostra como os coeficientes podem ser calculados após a simplificação

$$\begin{gathered} P_1 = (y_2 - y_3) \qquad P_2 = (y_3 - y_1) \qquad P_3 = (y_1 - y_1) \\ Q_1 = (x_3 - x_2) \qquad Q_2 = (x_1 - x_3) \qquad Q_3 = (x_2 - x_1) \\ A = \tfrac{1}{2}(P_2Q_3 - P_3Q_2) \end{gathered} \tag{A.10}$$

$$K_{ij}^{e} = \frac{1}{4A}(P_i P_j + Q_i Q_j) \tag{A.11}$$

As matrizes dos elementos e a matriz global são apresentadas nas equações A.12 e A.13. Após a definição das matrizes elementares é realizado o mapeamento de seus coeficientes para a matriz global. Enquanto a matriz elementar é da ordem do número de nós do elemento, a matriz global de coeficientes é da ordem do número de nós da malha.

$$K^1 = \begin{pmatrix} 0,560 & -0,285 & -0,274 \\ -0,285 & 0,592 & -0,3066 \\ -0,274 & -0,306 & 0,580 \end{pmatrix} \; K^2 = \begin{pmatrix} 0,620 & -0,257 & -0,362 \\ -0,257 & 0,509 & -0,252 \\ -0,362 & -0,252 & 0,615 \end{pmatrix} \tag{A.12}$$

$$K = \begin{pmatrix} 0,560 & -0,285 & 0 & -0,274 \\ -0,285 & 1,213 & -0,257 & -0,669 \\ 0 & -0,257 & 0,509 & -0,252 \\ -0,274 & -0,669 & -0,252 & 1,195 \end{pmatrix} \tag{A.13}$$

A atribuição dos valores de contorno consiste na redução da ordem na matriz A.13 por meio da eliminação dos graus de liberdade (nós) com valores pré estabelecidos (contorno). As equações A.14 e A.15 exibem o sistema reduzido e a solução.

$$\begin{pmatrix} 0,560 & -0,285 & 0 & -0,274 \\ -0,285 & 1,213 & -0,257 & -0,669 \\ 0 & -0,257 & 0,509 & -0,252 \\ -0,274 & -0,669 & -0,252 & 1,195 \end{pmatrix} \begin{Bmatrix} 10 \\ V^2 \\ 0 \\ V^4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix} \tag{A.14}$$

$$\begin{pmatrix} 1,213 & -0.257 \\ -0,669 & -0.252 \end{pmatrix} \begin{Bmatrix} V^2 \\ V^4 \end{Bmatrix} = \begin{Bmatrix} 2.858 \\ 2.741 \end{Bmatrix} \Rightarrow \begin{Bmatrix} V^2 \\ V^4 \end{Bmatrix} = \begin{Bmatrix} 5.241 \\ 5.227 \end{Bmatrix} \tag{A.15}$$