Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Simulação Paralela do Método dos Elementos Finitos Para a Atividade Elétrica Cardíaca

Gilmar Ferreira da Silva Filho

JUIZ DE FORA
NOVEMBRO, 2017

Simulação Paralela do Método dos Elementos Finitos Para a Atividade Elétrica Cardíaca

Gilmar Ferreira da Silva Filho

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Bernardo Martins Rocha

JUIZ DE FORA

NOVEMBRO, 2017

Simulação Paralela do Método dos Elementos Finitos Para a Atividade Elétrica Cardíaca

Gilmar Ferreira da Silva Filho

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Bernardo Martins Rocha
Doutor em Modelagem Computacional pelo LNCC

JUIZ DE FORA
28 DE NOVEMBRO, 2017

Ao meu irmão.

À minha avó e aos pais, pelo apoio e sustento.

Resumo

A modelagem computacional é uma poderosa ferramenta para o entendimento das mais diversas áreas do conhecimento. O comportamento do coração humano é alvo de intensos estudos e por conta disso as técnicas de aquisição de dados são cada vez mais poderosas, contribuindo para tornar as soluções numéricas para os modelos mais custosas. A modelagem da atividade eletromecânica do tecido cardíaco apresenta grande complexidade, dada as suas características multi-escala e multi-física, e exige a interação de diferentes modelos. Esses fatores tornam a utilização de computação de alto desempenho, com *cluster* de computadores e programação paralela, uma abordagem viável. O presente trabalho busca usar ferramentas de programação paralela para melhorar o desempenho da simulação do método dos elementos finitos para a atividade eletromecânica do coração, de tal forma que problemas de grande escala, abordando geometrias realísticas e detalhadas do coração demandem um menor tempo de execução.

Palavras-chave: modelagem computacional, *cluster*, coração, programação paralela

Abstract

Computer modeling is a powerful tool to the understanding of the most diverse areas of knowledge. Human heart behavior is the subject of intense study and because of that the data acquisition techniques are increasingly powerful, helping to make numerical solutions for the models more expensive. Electromechanical modeling of the heart tissue presents great complexity, given its multi-scale and multi-phisical features, and demands the interaction of different models. These factors make the use of high performance computing, with computer cluster and parallel programming, a viable approach. The present work aims to use parallel programming tools to enhance the performance of the simulation of finite elements method for the electromechanical heart activity, so that large scale problems, approaching realistic and detailed geomotries of the heart demand a shorter runtime.

Keywords: computer modelling, cluster, heart, parallel programming

Agradecimentos

A todos os meus parentes, pelo encorajamento e apoio.

Ao professor Bernardo pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"Qualquer realização é uma servidão. Obriga a uma realização mais elevada."

Albert Camus

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

EDOs	Equações diferenciais ordinárias
EDPs	Equações diferenciais parciais
MEF	Método dos Elementos Finitos
MPI	*Message Passing Interface*
PETSc	*Portable, Extensible Toolkit for Scientific Computation*
GPU	*Graphics Processing Unit*
CUDA	*Cumpute Unified Device Architecture*
XML	*eXtensible Markup Language*
XDMF	*eXtensible Data Model and Format*
LNCC	Laboratório Nacional de Computação Científica

1 Introdução

A modelagem computacional é uma poderosa ferramenta para a análise e entendimento de problemas das mais diversas áreas do conhecimento. Um modelo matemático que descreve em detalhes um determinado fenômeno pode ser obtido a partir do conhecimento de princípios físicos e matemáticos que o regem. Soluções numéricas das equações que descrevem o modelo podem então ser alcançadas.

A sofisticação dos problemas com os quais a humanidade tem se deparado, aliada ao crescente poder de obtenção de dados tornam os modelos matemáticos cada vez mais complexos e geram a necessidade da obtenção de resultados cada vez mais refinados e em menor tempo, o que torna as soluções numéricas ainda mais complexas e computacionalmente custosas.

1.1 Apresentação do tema

A computação de alto desempenho se refere à utilização de supercomputadores ou *clusters* de computadores para a realização de tarefas de alto custo computacional. Um *cluster* de computadores pode ser descrito como um conjunto de computadores ligados em uma rede de tal forma que possam ser vistos como uma única máquina, tornando possível a utilização do poder de processamento de múltiplas máquinas para a execução de uma tarefa. A utilização de técnicas de computação paralela com *cluster* de computadores configura um poderoso aliado à modelagem computacional, possibilitando a obtenção de resultados em menor intervalo de tempo.

O conhecimento do comportamento do coração e o poder das técnicas de aquisição de dados (eletrocardiograma, ultrassom, etc.) são cada vez maiores (Sainte-Marie et al, 2006), aliado a isso a modelagem do comportamento elétrico do coração é alvo de crescente interesse médico-científico, dada a sua grande importância para a compreensão de diversos fenômenos, que ainda não são bem entendidos, associados ao comportamento fisiológico do coração. O fenômeno de interesse apresenta uma grande complexidade devido as suas

características multi-escala e multi-física, e exige a interação de diferentes modelos, por isso, a utilização da computação paralela se mostra uma ferramenta extremamente útil para o problema.

Um importante método para a resolução dos modelos que descrevem a atividade elétrica cardíaca é o método dos elementos finitos (MEF) (Zhu et al, 2005; Saad et al, 2003; Rincon et al, 2013). Esse método é um procedimento numérico para determinar soluções aproximadas de equações diferenciais parciais (Zhu et al, 2005). De maneira sucinta, o MEF particiona um domínio em diversos subdomínios, chamados de elementos e, então, aproxima uma equação diferencial através de uma formulação variacional, resultando em um sistema linear de equações a serem calculadas.

O simulador Cardiax (Rocha et al, 2014) é uma implementação de um modelo eletromecânico acoplado capaz de realizar simulações computacionais da atividade eletromecânica cardíaca. Esse modelo é formado por modelos da eletrofisiologia (a nível celular) e modelos para a deformação e contração do tecido cardíaco (atividade elétrica e mecânica). A implementação computacional do modelo é feita utilizando métodos numéricos adequados e eficientes para a sua solução. Um dos métodos numéricos utilizados para os modelos da atividade elétrica e mecânica é o MEF. Uma ferramenta importante para a implementação paralela do simulador é o PETSc. O *Portable, Extensible Toolkit for Scientific Computation* (PETSc) é um conjunto de estruturas de dados e rotinas que provê blocos de construção, que definem soluções de equações diferenciais parciais e problemas relacionados, para a implementação de códigos de aplicações em larga escala em computadores de alta performance (Balay et al, 2016, 1997). O PETSc utiliza o padrão *Message-Passing Interface* (MPI) (Balaji et al, 2014) para as comunicações com passagem de mensagem. MPI é uma biblioteca para passagem de mensagens em computação paralela, proposto como um padrão por um vasto comitê de fornecedores, implementadores e usuários.

1.2 Problema

A execução do MEF no simulador Cardiax para problemas de grande escala, com geometrias mais reais e detalhadas do coração, pode exigir extremo esforço computacional.

Esses problemas podem levar ao consumo inviável de tempo, podendo executar por horas ou até dias, ou podem ser grandes a ponto de não serem comportados na memória de uma única máquina. A abordagem da computação paralela é uma forma de atacar esse problema.

A implementação do MEF envolve a discretização de um domínio em uma malha composta por elementos e nós, além da resolução de uma série de rotinas numéricas, como integrações numéricas e montagem de matrizes e vetores. Em uma abordagem paralela, com passagem de mensagens, do MEF é necessária a distribuição dos subconjuntos da malha de elementos para cada processador. Essa distribuição deve ser feita de maneira a minimizar o número de elementos adjacentes designados a diferentes processos, para que seja minimizado o custo de comunicação entre processadores, e de maneira a designar uma quantidade igual de elementos para cada processo, para que as computações entre os processadores estejam balanceadas (George et al, 2013). Essa tarefa pode ser realizada de maneira eficiente pelo pacote METIS (George et al, 2013) de particionamento de domínios. Além disto, a biblioteca PETSc permite a paralelização das diversas rotinas numéricas envolvidas pelo MEF sem exigir muito esforço de programação.

1.3 Justificativa

O coração humano é constantemente alvo de intensos estudos. O seu entendimento é de grande importância por ser um órgão vital. Doenças cardiovasculares vitimizam pessoas no mundo todo e representam um problema não só para a qualidade de vida humana, mas financeiramente (*World Health Organization*, 2017). Uma forma de melhor entender esse órgão e encontrar soluções para os problemas que o envolvem é a formulação de modelos matemáticos e simulação computacional dos fenômenos que englobam a atividade cardíaca.

A abordagem da modelagem computacional da atividade eletromecânica do coração, utilizando modelos matemáticos e simulações no computador pode ser de grande valia para o avanço nesta área, uma vez que através desta é possível obter resultados de experimentos caros, complexos ou inviáveis de se realizar. Porém, muitas vezes problemas mais realísticos demandam alto poder computacional, o que pode levar algumas soluções de

modelos a serem inviáveis, por tomarem muito tempo ou por serem muito grandes.

O simulador Cardiax, quando submetido a problemas de grande escala, se mostra inviável por tomar muito tempo para resolvê-los, portanto uma implementação paralela de seus resolvedores numéricos acompanhada da execução em um *cluster* de computador se mostra pertinente, afinal o ganho de desempenho proporcionado por tal abordagem poderá tornar problemas inviáveis computáveis em tempo satisfatório.

1.4 Objetivos

O objetivo deste trabalho é aprimorar, por meio de computação paralela, a simulação do MEF para a atividade elétrica cardíaca no simulador Cardiax.

A princípio, objetiva-se realizar a paralelização da solução numérica para um problema mais simples, como o problema de Poisson em três dimensões (Saad et al, 2003). Em seguida, busca-se a paralelização da componente responsável pela simulação da atividade elétrica cardíaca. Com estes objetivos, espera-se principalmente o ganho de desempenho para problemas de grande porte na simulação do MEF no simulador Cardiax.

Por fim, será feita a análise dos resultados obtidos, comparando o desempenho do código paralelo com o código serial, por meio de medidas como o *speedup* obtido, a eficiência e escalabilidade dos problemas paralelizados.

1.5 Estrutura do trabalho

O segundo capítulo do trabalho discute os modelos e métodos pertinentes à simulação do MEF para a atividade elétrica cardíaca, apresentando um panorama dos principais conceitos pertinentes e abordando alguns trabalhos relacionados. O terceiro capítulo aborda a metodologia da implementação paralela do simulador cardíaco, detalhando as ferramentas utilizadas e apresentando a implementação do simulador e as modificações conduzidas no processo de paralelização do mesmo. O quarto capítulo apresenta os resultados obtidos com os experimentos conduzidos para a validação da implementação paralela, discutindo a natureza e as implicações dos mesmos. Finalmente, o quinto capítulo conclui o trabalho apresentando as conclusões do autor e descrevendo ideias para trabalhos futuros.

2 Modelos e métodos

Este trabalho objetiva aprimorar, por meio de computação paralela, a simulação do MEF para a atividade eletromecânica cardíaca. Para isto, este capítulo busca revisar parte da bibliografia relacionada e apresenta um panorama dos principais conceitos presente neste trabalho.

2.1 Equações diferenciais parciais

Equações diferenciais parciais modelam matematicamente fenômenos da natureza ao relacionarem diversas derivadas parciais de quantidades físicas, como temperatura, energia, momento, velocidade, etc. Nestas equações a variável de interesse é influenciada por outras variáveis independentes e todas estas variáveis representam alguma grandeza relacionada ao fenômeno de interesse. EDPs são capazes de descrever os mais diversos fenômenos físicos, como a difusão do calor, propagação de ondas, eletroestática, eletrodinâmica, dinâmica dos fluidos, etc. Muitas vezes fenômenos totalmente distintos podem ser modelados pelas mesmas EDPs.

Uma das equações diferenciais parciais mais comuns, encontrada em diversas áreas é a equação de Poisson (Saad et al, 2003). A equação de Poisson é uma EDP elíptica que pode descrever, dentre vários outros fenômenos, a distribuição do potencial elétrico em um domínio Ω, fenômeno de interesse neste trabalho. A equação de Poisson é dada por:

$$-\Delta u(x) = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = f, \text{ para } x = (x_1, x_2) \text{ em } \Omega, \tag{2.1}$$

onde Ω é um domínio delimitado em $\mathbb{R}^2$ e x_1 e x_2 são variáveis no espaço. A equação deve ser satisfeita para os pontos localizados no interior do domínio Ω.

Outro aspecto importante são as condições que devem ser satisfeitas no contorno Γ de Ω. Essas condições são chamadas de condições de contorno e as mais comuns são a condição de Dirichlet $u(x) = \phi(x)$ e a condição de Neumann $\nabla u(x) \cdot \vec{n} = g(x)$.

A partir da discretização de EDPs, soluções aproximadas de EDPs podem ser

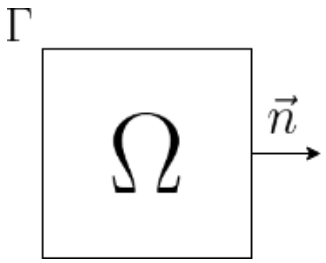


Figura 2.1: Domínio Ω para a equação de Poisson

obtidas computacionalmente por diferentes métodos, como o método das diferenças finitas (LeVeque et al, 1998) ou o método dos elementos finitos.

2.2 Modelagem da atividade elétrica cardíaca

É sabido que doenças cardíacas vitimizam milhares de pessoas no mundo todo. A modelagem computacional do coração pode proporcionar dados importantes para a sua melhor compreensão. A seguir apresenta-se uma visão sucinta deste tópico.

2.2.1 Uma visão geral

Um modelo matemático muito utilizado, é o conjunto de equações de bidomínio (Plonsey et al, 1988). Quase sempre presente em modelos elétricos cardíacos, o modelo bidomínio é o mais completo conjunto de EDPs que descreve a atividade elétrica cardíaca (Vigmond et al, 2008; Nash e Panfilov et al, 2004; Sundnes et al, 2007). Trata-se de um modelo contínuo para a propagação do potencial de ação no tecido cardíaco, no qual assume-se que o tecido cardíaco é dividido em dois domínios separados, o intracelular e o extracelular.

Como os modelos são descritos por conjuntos de equações, a simulação computacional leva à necessidade de métodos de discretização, como o MEF, e os sistemas lineares obtidos devem ser resolvidos com a utilização de métodos numéricos.

2.2.2 Modelo monodomínio

Geralmente o modelo bidomínio é utilizado em simulações complexas do comportamento elétrico cardíaco, porém é um modelo de difícil resolução e análise dada a sua complexidade matemática e onerosa simulação computacional. O modelo monodomínio é uma simpli-

ficação que torna simulações computacionais mais tangíveis. Este modelo é obtido ao se tomar hipóteses simplificadoras sobre o modelo bidomínio. Basicamente, é possível simplificar o sistema de EDPs provenientes do modelo bidomínio em uma equação parabólica, descrevendo apenas as dinâmicas do potencial transmembrânico do tecido cardíaco, como explica Sundnes et al (2007).

As células do músculo cardíaco são capazes de responder ativamente a um estímulo elétrico. Como estas células são conectadas, uma célula estimulada pode passar o sinal elétrico adiante para células em sua vizinhança. Esta habilidade possibilita a completa ativação do coração. Quando as células estão em repouso há uma diferença de potencial através da membrana celular. Nestas condições, o potencial intracelular é negativo e o potencial extracelular é positivo. Quando as células são estimuladas eletricamente elas despolarizam, o que é um fenômeno que ocorre muito rapidamente, e na sequencia um processo de repolarização recupera a diferença de potencial inicial. Este ciclo é chamado de potencial de ação. O modelo monodomínio é um modelo matemático que descreve essa diferença de potencial através da membrana celular, também chamado de potencial transmembrânico (Sundnes et al, 2007).

As equações do monodomínio são então, EDPs do tipo reação-difusão e descrevem a difusão do estímulo elétrico através do tecido cardíaco. A descrição matemática das equações é dada por:

$$\chi C_m \frac{\partial v}{\partial t} = \nabla \cdot (M_i \nabla v) - \chi I_{ion}, \tag{2.2}$$

onde v descreve o potencial transmembrânico, χ é a razão superfície-volume das células cardíacas, C_m é a capacitância da membrana celular, I_{ion} é a densidade total de corrente iônica e M_i é o tensor de condutividade.

Para a completa descrição do modelo especificam-se as condições iniciais e de contorno apropriadas para $v = v(x,t)$ e $\eta = \eta(x,t)$:

$$v(x,0) = v_0(x) \tag{2.3}$$

$$\eta(x,0) = \eta_0(x) \tag{2.4}$$

Assumindo que o tecido é isolado em seu contorno, são impostas as condições de Neumann,

ou seja, condições de contorno de fluxo nulo sobre v:

$$\vec{n} \cdot \sigma \nabla v = 0, \tag{2.5}$$

onde $\vec{n}$ é um vetor unitário normal à superfície do tecido.

2.2.3 O simulador Cardiax

O simulador Cardiax foi desenvolvido por Rocha et al (2014) e define um modelo eletromecânico acoplado composto por um modelo celular eletromecânico, que fornece o potencial elétrico da membrana celular que inicia a ativação elétrica no tecido, e um modelo a nível do tecido para a simulação da atividade elétrica e mecânica. O modelo é implementado por meio de métodos numéricos adequados e eficientes. Problemas complexos, de geometria detalhada e malhas bem refinadas podem ser computacionalmente caros para a simulação no Cardiax; espera-se reduzir essa demanda computacional por meio de computação paralela. No capítulo 3, mais detalhes sobre a implementação do simulador serão apresentados.

2.3 O Método dos elementos finitos

Diversos fenômenos físicos e problemas da engenharia, fisiologia, entre outros, são comumente modelados por meio de sistemas de equações diferenciais parciais (EDPs) acoplados a sistemas de equações ordinárias (EDOs). As EDPs na maioria dos casos de interesse não possuem solução analítica e, portanto, devem ser discretizadas, ou seja, aproximadas por equações envolvendo um número finito de elementos (Saad et al, 2003; Zhu et al, 2005; Rincon et al, 2013). Um dos métodos mais comuns para a discretização de EDPs é o MEF, o qual será melhor discutido nas subseções a seguir.

2.3.1 Formulação variacional

Seja o domínio Ω, com contorno $\Gamma = \Gamma_D \bigcup \Gamma_N$. Considere o problema suplementado pelas condições de contorno de Dirichlet sobre Γ_D e Neumann sobre Γ_N:

$$-\Delta u = f \text{ em } \Omega. \tag{2.6}$$

Para que o MEF seja aplicável neste problema, é necessário que o mesmo seja expressado numa forma mais conveniente para que seja possível aproximar a solução da equação. Para extrair sistemas de equações que produzem a solução, é comum utilizar a formulação variacional ou formulação fraca do problema.

Seja $L^2(\Omega) = \left\{v : \Omega \to \mathbb{R};\ \int_\Omega |v|^2 dx < \infty\right\}$. Integrando no domínio Ω e multiplicando o problema por uma função v, obtemos:

$$-\int_\Omega (\Delta u + f)v\, dx = 0. \tag{2.7}$$

Utilizando o teorema da divergência, a equação (2.7) pode ser reescrita como:

$$\int_\Omega \nabla u \cdot \nabla v\, dx - \int_\Gamma \nabla u \cdot \vec{n} v\, ds = \int_\Omega f v\, dx. \tag{2.8}$$

A partir desta formulação fraca, podemos definir o seguinte problema: dado $f \in L^2(\Omega)$, encontrar $u \in U$, tal que

$$\int_\Omega \nabla u \cdot \nabla v\, dx - \int_\Gamma \nabla u \cdot \vec{n} v\, ds = \int_\Omega f v\, dx,\ \forall v \in V.$$

Onde $U = \{u \in L^2(\Omega),\ \nabla u \in L^2(\Omega);\ u|_{\Gamma_D} = \phi\}$ e $V = \{v \in L^2(\Omega),\ \nabla v \in L^2(\Omega);\ v|_{\Gamma_D} = 0\}$. As condições de Dirichlet, conhecidas como condições essenciais, são impostas no espaço, enquanto as condições de Neumann, também chamadas de condições naturais, são impostas diretamente na formulação variacional.

Como $\Gamma = \Gamma_D \bigcup \Gamma_N$, podemos reescrever o problema variacional como:

$$\int_\Omega \nabla u \cdot \nabla v\, dx - \int_{\Gamma_D} \nabla u \cdot \vec{n} v\, ds - \int_{\Gamma_N} \nabla u \cdot \vec{n} v\, ds = \int_\Omega f v\, dx,\ \forall v \in V.$$

Como todo $v \in V$ se anula no contorno de Dirichlet e como $\nabla u \cdot \vec{n} = g$ sobre Γ_N, então:

$$\int_\Omega \nabla u \cdot \nabla v\, dx - \int_{\Gamma_N} g v\, ds = \int_\Omega f v\, dx,\ \forall v \in V.$$

Desta forma, o problema variacional pode ser representado da seguinte forma:

$$\text{Achar } u \in U, \text{ satisfazendo } a(u,v) = f(v), \forall v \in V. \tag{2.9}$$

A forma bilinear $a(.,.)$ é dada por

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, dx$$

e o funcional linear $f(.)$ por

$$f(v) = \int_\Omega fv \, dx - \int_{\Gamma_N} gv \, ds.$$

A partir da forma variacional, é possível aproximar uma solução do problema, desde que o espaço de soluções seja aproximado por um espaço de dimensão finita.

2.3.2 Método de Galerkin

O método dos elementos finitos consiste da aproximação da formulação fraca por um problema de dimensão finita obtido ao se substituir o espaço V por um subespaço de funções que são definidas como polinômios de baixo grau em elementos do domínio original.

O método de Galerkin aproxima o espaço V com o subespaço V_m gerado por um conjunto de m elementos do espaço de Hilbert $H_0^1(\Omega) = \{v \in H^1(\Omega);\ v|_\Gamma = 0\}$, onde $H^1(\Omega) = \{v \in L^2(\Omega);\ \nabla v \in [L^2(\Omega)]^2\}$. Desta forma, temos que:

$$V_m = \{\varphi_1, \varphi_2, \varphi_3, ..., \varphi_m\}, \tag{2.10}$$

onde $\{\varphi_i, i \in N\}$ é uma base de $H_0^1(\Omega)$. Desta forma, o objetivo é encontrar uma solução aproximada $u^h = u^h(x)$ do problema (2.9).

O espaço de dimensões finitas V_m é então definido como o espaço de todas as funções que são lineares por partes e contínuas na região aproximada $\Omega_h = \bigcup_{i=1}^m E_i$, onde E_i são os elementos da malha que representa o domínio aproximado. A Figura 2.2 ilustra uma malha que representa um domínio aproximado qualquer.

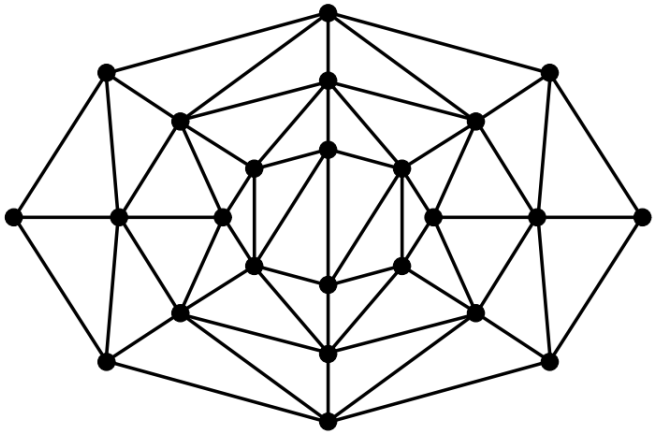


Figura 2.2: Exemplo de malha (Saad et al, 2003)

Aproximando o problema (2.9) por:

$$\text{Achar } u^h \in V_m, \text{ satisfazendo } a(u^h, v^h) = f(v^h), \ \forall v^h \in V_m. \tag{2.11}$$

Temos que a solução aproximada u^h pode ser representada como:

$$u^h = \sum_{i=1}^{m} u_i \varphi_i, \text{ onde } \varphi_i \in V_m. \tag{2.12}$$

Para se obter a solução aproximada u^h é necessário determinar os coeficientes $u_i \in \mathbb{R}$.

Escolhendo $v^h = \varphi_j$, podemos reescrever o problema (2.11) como:

$$\text{Achar } u_i \in \mathbb{R}, \text{ tal que } \sum_{i=1}^{m} a(\varphi_i, \varphi_j) u_i = f(\varphi_j), \ j = 1, 2, 3, ..., m. \tag{2.13}$$

Este problema é equivalente à forma matricial:

$$Ku = F,$$

onde $u = \{u_i\}$ é o vetor de incógnitas e $K_{ij} = a(\varphi_i, \varphi_j)$, $i, j = 1, 2, 3, ..., m$ e $F_j = f(\varphi_j)$, $j = 1, 2, 3, ..., m$. A matriz K é chamada matriz de rigidez e o vetor F é chamado vetor fonte ou vetor de carga.

No contexto do problema que está sendo abordado, a matriz de rigidez K e o

vetor fonte F são dados por:

$$K_{ij} = \int_{\Omega_h} \nabla\varphi_i \cdot \nabla\varphi_j \, dx, \; i,j = 1,2,3,...,m$$
$$F_j = \int_{\Omega_h} f\varphi_j \, dx, \; j = 1,2,3,...,m$$

a montagem de K e F é feita adicionando-se adequadamente as contribuições locais, definidas a nível de elemento por:

$$K^e_{ij} = \int_E \nabla\varphi^e_i \cdot \nabla\varphi^e_j \, dx, \; i,j = 1,2,3,...,n$$
$$F^e_j = \int_E f\varphi^e_j \, dx, \; j = 1,2,3,...,n$$

onde $\Omega_h = \bigcup_{i=1}^{m} E_i$, $K = \sum_{i=1}^{m} K^e$, $F = \sum_{i=1}^{m} F^e$ e n é o número de nós do elemento E.

2.3.3 Funções de interpolação

Para o cálculo da matriz global K_{ij} e do vetor F_i é necessário definir explicitamente as funções φ_i, base do subespaço V_m (Rincon et al, 2013). As funções φ_i são escolhidas como funções polinomiais, para as quais as condições de fronteira são satisfeitas.

O objetivo principal na escolha das funções de base é fazer com que a matriz de rigidez seja uma matriz esparsa e que o sistema linear resultante seja bem condicionado (Rincon et al, 2013).

Assumindo um problema de uma dimensão, as funções φ_i escolhidas são funções de interpolação linear por partes que satisfazem a seguinte condição:

$$\varphi_i(x_j) = \begin{cases} 1, \text{ se } i = j \\ 0, \text{ se } i \neq j \end{cases} \tag{2.14}$$

onde $x_j \in [a,b]$ é o nó da malha que representa o domínio aproximado, para um domínio $\Omega = [a,b]$. Tomando m divisões em $[a,b]$, define-se o passo $h_j = x_{j+1} - x_j$, $j = 1,2,3,...,m$. No caso em que os nós são equidistantes, $h_j = h = \frac{b-a}{m}$.

Em cada nó j, define-se a função φ_j satisfazendo a condição 2.10, de tal forma que:

$$\varphi_j(x) = \begin{cases} \frac{x-x_{j-1}}{h_{j-1}}, \forall x \in [x_{j-1}, x_j] \\ \frac{x_{j+1}-x}{h_j}, \forall x \in [x_j, x_{j+1}] \\ 0, \forall x \notin [x_{j-1}, x_{j+1}] \end{cases} \tag{2.15}$$

As funções de base locais φ_j^e são definidas em cada elemento $E = [x_1^e, x_2^e]$, porém são análogas às funções globais e podem ser representadas na forma linear , para cada elemento como:

$$\varphi_1^e = \frac{1}{h}(x_2^e - x), \forall x \in [x_1^e, x_2^e]$$
$$\varphi_2^e = \frac{1}{h}(x - x_1^e), \forall x \in [x_1^e, x_2^e]$$
$$\varphi_j^e = 0, \forall x \in [x_1^e, x_2^e].$$

assumindo que os nós são equidistantes. Assim, a matriz local K_{ij}^e para o caso linear é dada por:

$$K_{ij}^e = \begin{bmatrix} K_{11}^e & K_{12}^e \\ K_{21}^e & K_{22}^e \end{bmatrix}$$

onde:

$$K_{ij}^e = \int_E \frac{d\varphi_i^e}{dx}\frac{d\varphi_j^e}{dx}\, dx,\ i, j = 1, 2$$

e o vetor local F_j^e é dado por:

$$F_j^e = \begin{bmatrix} F_1^e \\ F_2^e \end{bmatrix}$$

onde:

$$F_j^e = \int_E f\varphi_j^e\, dx + \varphi^e(b)g,\ j = 1, 2$$

assumindo um domínio $\Omega = [a, b]$.

Por fim, a montagem da matriz de rigidez global K e do vetor fonte global F é feita através da soma em blocos das matrizes e vetores locais, levando em consideração as modificações necessárias dadas as imposições das condições de contorno do problema.

Através da resolução do sistema $Ku = F$ é possível obter, usando o MEF, a solução aproximada do problema considerado. Existem também os problemas transientes. Estes seguem o mesmo método apresentado para serem discretizados e resolvidos pelo MEF, porém inclui-se o tratamento da passagem de tempo por meio da integração numérica no tempo das resoluções dos sistemas.

2.4 Computação paralela

Diversas das técnicas numéricas até agora discutidas, como o MEF, os modelos bidomínio e monodomínio e as soluções numéricas de sistemas lineares esparsos, muitas das vezes exigem um poder computacional que vai além daquele oferecido por uma única máquina, de tal forma que a obtenção de resultados se torna inviável, ou por tomarem tempo demais ou por ser impossível computar os resultados em uma única máquina. A computação paralela é uma forma eficaz de atacar este problema e. por isso, é o foco principal deste trabalho. Este capítulo abordará conceitos pertinentes para a computação paralela, bem como arquiteturas e ferramentas pertinentes ao presente trabalho.

2.4.1 O que é computação paralela?

Pode-se dizer que a computação paralela é quando o poder de várias unidades de processamento é utilizado para computarem em concorrência um mesmo problema que está dividido entre cada uma dessas unidades.

Para entender como um problema é dividido para várias unidades de processamento, também chamadas de núcleos, define-se a idéia de programas seriais e programas paralelos, ou seja, programas que são escritos para aproveitarem do poder de processamento de apenas um núcleo e programas destinados a executarem em paralelo. Segundo Pacheco et al (2011), existem duas abordagem para se obter paralelismo em um programa, o paralelismo por tarefa e o paralelismo por dados. Foca-se nesse trabalho no conceito do paralelismo por dados, onde o que é divido entre os núcleos são os dados e não a tarefa que cada um deve executar.

Além disto, Saad et al (2003) define 6 formas de paralelismo: múltiplas unidades

funcionais, *pipelining*, processamento vetorial, múltiplas *pipelines* de vetor, multiprocessamento e computação distribuída. Também são apresentados os três principais modelos de arquiteturas paralelas: o modelo de memória compartilhada, modelos de dado paralelo ou SIMD (*Single Instruction Multiple Data*) e o modelo de memória distribuída com passagem de mensagem. Tem-se interesse na forma de paralelismo de multiprocessamento e computação distribuída e na arquitetura de memória distribuída com passagem de mensagem.

A arquitetura de memória distribuída com passagem de mensagens segue a forma de paralelismo da computação distribuída, que é uma forma geral de um sistema de multiprocessamento, onde as unidades de processamento são computadores ligados por uma LAN (*Local Area Network*) e a comunicação entre eles é feita por passagem de mensagens. O MPI é um exemplo de um padrão que define uma biblioteca para a passagem de mensagens entre os nós de uma arquitetura desse tipo e um exemplo dessa arquitetura é o *cluster* de computadores.

Os dois principais tipos de sistemas paralelos são os sistemas de memória compartilhada e os sistemas de memória distribuída.

Em um sistema de memória compartilhada os núcleos podem compartilhar o acesso à memória do computador. A princípio, cada núcleo pode ler e escrever em cada local de memória. Pode ser necessário coordenar os acessos dos núcleos aos locais compartilhados da memória para evitar condições de corrida.

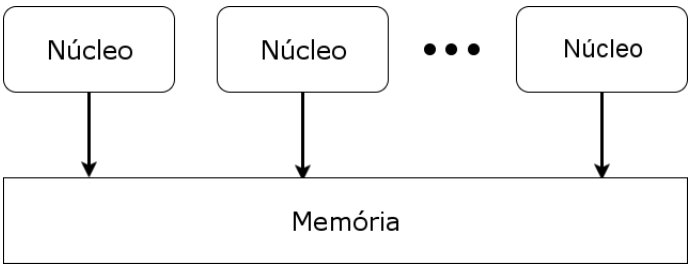


Figura 2.3: Sistema de memória compartilhada (adaptado de Pacheco et al (2011))

Em um sistema de memória distribuída, cada núcleo possui sua própria memória e os núcleos devem se comunicar explicitamente através de passagem de mensagens por uma rede, por exemplo.

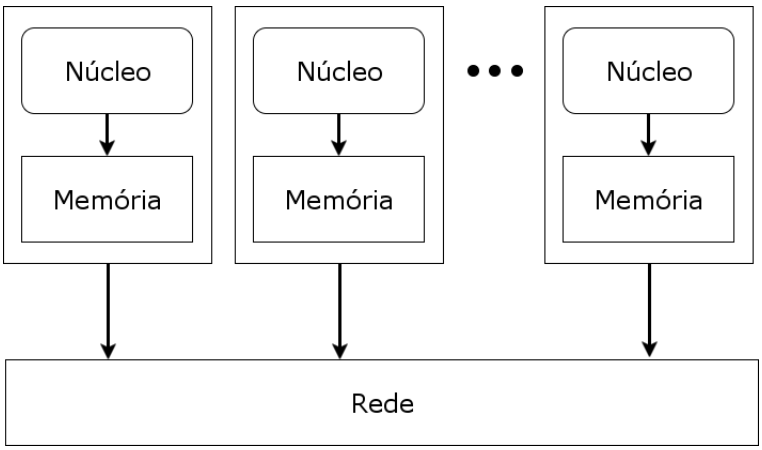


Figura 2.4: Sistema de memória distribuída (adaptado de Pacheco et al (2011))

2.4.2 Métricas

Dentro da computação paralela, é importante entender os conceitos de *speedup*, eficiência e escalabilidade, bem como a lei de Amdahl.

O *speedup* nada mais é do que o ganho de aceleração com a paralelização em relação ao tempo serial de um programa. Se chamarmos o tempo serial de T_{serial} e o tempo paralelo de $T_{paralelo}$ e o *speedup* de S, temos que:

$$S = \frac{T_{serial}}{T_{paralelo}}. \tag{2.16}$$

Se S é igual ao número de processadores executando o programa, então o *speedup* é dito linear. Existem casos especiais em que o *speedup* pode ser maior do que linear, estes casos são chamados de *speedup* super-linear. Uma das causas mais comuns do *speedup* super-linear é o chamado efeito *cache*. Processadores podem ter diferentes tamanhos de memória *cache* e é relativamente comum em computação paralela que uma maior parte ou todo o dado sendo processado caiba na *cache* dos processadores, reduzindo drasticamente o tempo de acesso à memória, consequentemente ocasionando um *speedup* super-linear.

A lei de Amdahl diz, em suma, que a menos que todo um programa serial seja paralelizável, seu *speedup* será limitado pela quantidade de tarefa serial contida no programa (Pacheco et al, 2011).

A eficiência de um programa paralelo é a razão entre o *speedup* e o número de processadores executando o programa (Pacheco et al, 2011). Se chamarmos a eficiência

de E e o número de processadores executando o programa de p, então:

$$E = \frac{S}{p} \tag{2.17}$$

Por fim, diz-se que um programa paralelo é escalável se, com o aumento do número de processadores usados pelo programa, for possível encontrar uma taxa de aumento correspondente no tamanho do problema de maneira a se obter uma eficiência constante. Um programa é fortemente escalável, se com o aumento do número de processadores, não for necessário aumentar o tamanho do problema para manter a eficiência constante. Um programa é dito fracamente escalável, se ao aumentar o número de processadores a uma taxa, for necessário um aumento no tamanho do problema na mesma taxa, para manter a eficiência constante (Pacheco et al, 2011).

Todos estes conceitos são de importantes para a análise do ganho de desempenho no processo de paralelização de um programa, pois eles fornecem dados que podem ajudar a direcionar o foco da paralelização no caminho mais eficiente possível.

2.4.3 PETSc

Métodos como o MEF e os resolvedores numéricos de sistemas lineares envolvem rotinas numéricas que são custosas em relação ao tempo de execução no computador. Implementá-las para que sejam executadas em paralelo é uma solução para o problema do alto custo de tempo de execução, porém tal implementação tende a ser complexa e exige grande esforço de programação.

O PETSc é uma biblioteca que pode ser usada em códigos escritos em Fortran, C, C++ e MATLAB, e que provê um conjunto de estruturas de dados e módulos para a implementação em paralelo da solução numérica de EDPs e problemas relacionados, utilizando o padrão MPI para as comunicações por passagem de mensagens. O PETSc ainda possui um conjunto de resolvedores numéricos em paralelo de sistemas lineares e não lineares (Balay et al, 2016). É natural reconhecer que uma ferramenta como o PETSc ameniza o esforço e a complexidade de programação exigidos para a implementação paralela das rotinas numéricas. Mais detalhes sobre os módulos e estruturas do PETSc são discutidos no capítulo 3.

2.5 Trabalhos relacionados

O campo da computação paralela vem sendo amplamente pesquisado. Independente de qual o contexto em que é aplicada, o objetivo é sempre o de obter melhor desempenho quando comparado a abordagem serial.

As equações do modelo bidomínio usadas para a descrição da atividade elétrica do tecido cardíaco são computacionalmente custosas. Buscando uma forma eficiente de resolver as equações do modelo bidomínio, dos Santos et al (2005) discutem técnicas paralelas de precondicionamento para os métodos de solução das equações, implementando diferentes precondicionadores em paralelo utilizando o PETSc, e comparando os resultados testados em um *cluster* de computadores.

É comum o uso de técnicas de precondicionamento em métodos numéricos para a solução de sistemas de equações para a obtenção de resultados mais precisos. Segundo dos Santos et al (2005), o estágio de solução do sistema linear associado à parte elíptica do modelo bidomínio é a etapa mais custosa. Para a obtenção de desempenho computacional nesta etapa de solução do sistema linear, dos Santos et al (2005) analisa a utilização de três diferentes tipos de precondicionadores (*incomplete lower upper factorization*, *additive Schwartz method* e multigrid) para o método do gradiente conjugado, propondo uma implementação em paralelo com a utilização da biblioteca PETSc. A implementação dos três precondicionadores foi feita de maneira a reduzir a comunicação entre processos e as rotinas numéricas paralelizadas com as implementações presentes na biblioteca PETSc. Como resultado, dos Santos et al (2005) destacam a superioridade do precondicionador *multigrid*, apesar do baixo ganho de *speedup* em todas as três abordagens.

Recentemente, novas arquiteturas paralelas surgiram. Uma dessas novas arquiteturas consiste da utilização de GPUs (*Graphic Processing Units*) para a computação paralela. Alguns trabalhos atuais analisam a utilização desta abordagem para a solução de modelos cardíacos. Amorim e dos Santos et al (2013) compararam os resultados da execução da solução das equações do modelo bidomínio nessa arquitetura com a execução sequencial em um único processador, e com a arquitetura de memória distribuída utilizando *cluster* de computadores.

Para a solução das tarefas do modelo bidomínio, Amorim e dos Santos et al

(2013) utilizam o padrão *OpenGL* para abordar o paralelismo utilizando GPUs. O padrão *OpenGL* faz uso do modelo de programação por *Stream*. Uma *Stream* é um conjunto de registros que necessitam de computação similar e uma *Stream* é computada por um *kernel* (Amorim e dos Santos et al, 2013), que pode ser visto como um bloco de código a ser computado na GPU.

Similarmente à arquitetura de memória compartilhada, no paralelismo com uso de GPUs também existe o problema de perda de desempenho devido à comunicação. Enquanto em memória compartilhada existe o problema de comunicação entre unidades de processamento, no caso do paralelismo com GPUs o problema se dá por conta da necessidade de transferências de dados do processador para a GPU e da GPU para o processador. Para a solução do sistema não linear de equações diferenciais ordinárias utilizando GPU, Amorim e dos Santos et al (2013) propõem que se mantenham todas as variáveis na memória da GPU e apenas os vetores que guardam as soluções sejam transferidos entre o processador e a GPU, de tal forma que todas as computações necessárias para a obtenção da solução possam ser feitas com apenas um *kernel*.

Um problema associado ao paralelismo por GPUs é o suporte limitado das GPUs atuais a operações com precisão dupla de ponto flutuante. Esse empecilho leva a erros numéricos na execução dos métodos. Amorim e dos Santos et al (2013) atentam para este problema e mostram que em sua implementação para o método do gradiente conjugado, a execução sem precisão dupla acaba levando à não convergência do método. Sugerem então que alguns cálculos sejam feitos no processador, o que gera uma pequena perda de desempenho.

O uso de paralelismo com GPUs é promissor. Em seus resultados, Amorim e dos Santos et al (2013) obtêm um ganho de desempenho considerável em sua implementação, que se mostra 1,4 vezes mais rápida que a implementação utilizando *clusters* de computadores compostos de 16 unidades de processamento discutida em (dos Santos et al, 2005).

Ainda no campo da paralelização com GPUs, Mena et al (2015) segue uma abordagem diferente à do *OpenGL*. Através do uso da arquitetura CUDA ou *Compute Unified Device Archtecture*, Mena et al (2015) obtém aceleração na solução da atividade elétrica

do coração.

Mena et al (2015) sugerem uma implementação CUDA que consiste de dois subprogramas: um subprograma a ser executado no processador e um subprograma a ser executado na GPU. O subprograma do processador prepara as estruturas necessárias para a execução na GPU e move os dados da memória do processador para a memória da GPU. O subprograma da GPU executa as computações necessárias em paralelo. Mena et al (2015) ressaltam que em seu modelo, dois estágios contribuem para o alto custo computacional: a solução de um sistema de equações diferenciais ordinárias e um sistema linear associado a uma EDP parabólica.

Assim como sugerido por Amorim e dos Santos et al (2013), Mena et al (2015) propõem, para a solução do sistema de equações diferenciais parciais, que todas as variáveis residam sempre na memória da GPU, e que apenas o vetor de soluções seja transferido entre as memórias de processador e GPU, otimizando, desta forma, o ganho de performance. Alguns modelos mais complexos possuem uma quantidade muito grande de variáveis, o que pode impedir que essas sejam todas mantidas na memória da GPU. Neste caso, é necessário que exista transferência de grupos de variáveis entre a memória da GPU e a memória do processador. Estas transferências introduzem perda de performance, Mena et al (2015) ressaltam que uma implementação otimizada deve considerar a melhor compensação entre a divisão dos grupos de variáveis, o uso de memória e a perda de performance obtida com as transferências de dados entre GPU e processador.

Os resultados obtidos por Mena et al (2015) mostram enorme potencial para a tecnologia empregada, obtendo resultados até 50 vezes mais rápidos que a implementação sequencial em problemas tridimensionais.

3 Implementação computacional

Neste capítulo serão apresentados o método de implementação paralela do simulador cardíaco. A primeira seção detalha as ferramentas numéricas e de computação paralela utilizadas na implementação do trabalho. A segunda seção descreve as implementações iniciais, conduzidas com o objetivo de familiarizar-se com a biblioteca PETSc. A terceira seção apresenta os detalhes referentes à implementação do código do simulador Cardiax, explicando a utilização do METIS para o particionamento dos domínios utilizados nos experimentos, bem como a implementação do PETSc no código para garantir a eficiência nas rotinas numéricas e para se obter o paralelismo desejado. Além disso, são detalhadas as alterações na implementação sequencial, conduzidas no processo de paralelização do código.

3.1 Ferramentas de computação científica e de computação paralela

No código paralelo do Cardiax diversas ferramentas e bibliotecas auxiliam na implementação das estruturas e rotinas numéricas. Dentre elas destacam-se o PETSc e o METIS. Como já mencionado anteriormente, o PETSc fornece uma série de estruturas de dados e módulos para a implementação paralela da solução numérica de EDPs. O METIS é um pacote para o particionamento de grafos irregulares e malhas de elementos finitos. Esta seção apresenta os detalhes pertinentes destas ferramentas no intuito de melhor entender a implementação paralela do Cardiax como um todo.

3.1.1 MPI

O simulador Cardiax em sua implementação paralela é um sistema de memória distribuída com passagem de mensagens. O padrão deste modelo utilizado no simulador é o MPI (mais especificamente a implementação MPICH (Balaji et al, 2014)).

Como já mencionado, o MPI define um padrão para o modelo de passagem de mensagens, no qual dados são movidos de um processo a outro através de operações cooperativas em cada processo. O MPI provê também operações coletivas, operações de acesso à memória remota, criação dinâmica de processos e operações de entrada e saída paralelas. Todas as operações são expressas como funções, subrotinas ou métodos.

Um programa MPI deve inicializar o ambiente de passagem de mensagens, definindo um ou mais comunicadores (coleção de processos que podem trocar mensagens (Pacheco et al, 2011)) e o número de processos, realizar as passagens de mensagens e finalizar o ambiente de comunicação. Apesar de qualquer programa paralelo com passagem de mensagens poder ser desenvolvido apenas com funções básicas de envio e recebimento de mensagens, o MPI provê diversas outras operações complexas. Vale mencionar algumas funções básicas do MPI:

- *MPI_Init*: inicializa o ambiente de execução do MPI.
- *MPI_Finalize*: finaliza o ambiente de execução do MPI.
- *MPI_Send*: envia mensagem de um processo a outro.
- *MPI_Recv*: recebe a mensagem de um processo.
- *MPI_Comm_size*: determina o tamanho do grupo de processos associado ao comunicador.
- *MPI_Comm_rank*: determina o ranque (número) do processo no comunicador.

3.1.2 PETSc

A resolução de sistemas lineares de grande escala resultantes das discretizações do MEF e as estruturas de dados referentes a estes sistemas presentes no código do Cardiax são implementados através do PETSc. Trata-se de uma biblioteca organizada hierarquicamente, como mostra a Figura 3.1 que permite a usuários o emprego do nível de abstração que melhor se ajusta a um problema em particular. Com o uso de técnicas de programação orientada a objetos, a biblioteca fornece grande flexibilidade e robustez.

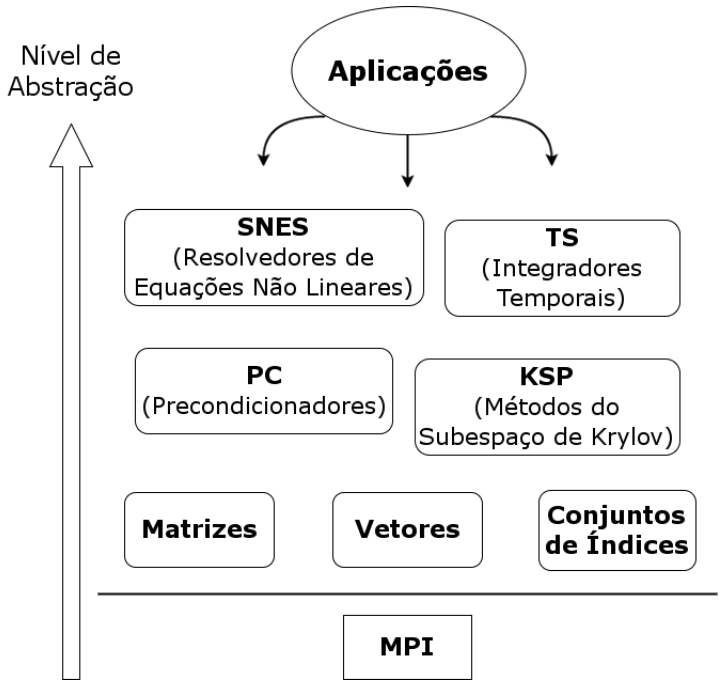


Figura 3.1: Organização dos módulos da biblioteca PETSc (adaptado de Balay et al (2016))

O PETSc é composto de diversos módulos e cada um manipula uma família de objetos e operações. Através do MPI, o PETSc combina o modelo de passagem de mensagem e algoritmos numéricos implementados em paralelo, para garantir que suas bibliotecas sejam capazes de lidar com as complexas trocas de mensagens necessárias durante a coordenação das computações presentes em aplicações de solução númerica de EDPs. Desta forma pode-se obter alto desempenho e facilidade de implementação em uma aplicação científica (Balay et al, 1997).

Alguns dos módulos e operações do PETSc lidam com:

- conjuntos de índices (*IS*);
- vetores (*Vec*);
- matrizes (*Mat*);
- produtos matriz-vetor paralelos;
- computação paralela de matrizes Jacobianas;
- métodos iterativos e subespaço de Krylov (gradiente conjugado) para a solução de sistemas lineares (*KSP*);

- diversos precondicionadores;
- entre outros.

Cada módulo consiste de uma interface abstrata e uma ou mais implementações utilizando estruturas de dados particulares.

3.1.3 METIS

Os algoritmos implementados no METIS são baseados no paradigma de particionamento multinível de grafos (George et al, 2013). O custo de comunicação entre processos resultante de um particionamento *k-way* (Karypis et al, 1998) usado pelo METIS geralmente depende do volume total de comunicação, da quantidade máxima de dados que um processo precisa receber e enviar e do número de mensagens que um processo precisa receber e enviar (George et al, 2013). O particionamento obtido pelo METIS minimiza diretamente o volume total de comunicação e provê suporte para a minimização do número de mensagens enviadas e recebidas por um processo. O objetivo do particionamento do METIS para malhas de elementos finitos é então, a minimização da comunicação e o balanceamento de carga entre processos.

O METIS provê uma série de programas de linha de comando para computar o particionamento de grafos irregulares e malhas, além de uma API que pode ser usada para invocar seus vários algoritmos a partir de programas em *C/C++* e *Fortran*. Foi utilizado neste trabalho, o programa de linha de comando *mpmetis*, que computa o particionamento de uma malha. O programa recebe como entrada os nós de cada elemento de uma malha, converte a malha em um grafo dual ou um grafo nodal e, então, computa o particionamento *k-way* da malha para seus elementos e nós. O arquivo de entrada para o programa consiste de uma linha-cabeçalho com a informação sobre o tamanho e o tipo da malha e n linhas contendo os nós que formam cada elemento da malha. A saída do programa é um arquivo com a partição dos elementos da malha e um arquivo com a partição dos nós da malha.

3.2 Implementações iniciais

A parte inicial do trabalho consistiu na familiarização com as bibliotecas PETSc. Para isso foram implementados dois códigos simples que fazem uso das principais implementações e estruturas do PETSc, medindo-se o *speedup* obtido com a sua execução em paralelo.

O primeiro código implementado foi uma simples rotina de multiplicação entre uma matriz esparsa e um vetor. Através desta implementação é possível perceber como o PETSc é eficiente para a paralelização de rotinas numéricas, visto que com o uso de suas estruturas abstratas de vetores e matrizes, bem como a própria rotina de multiplicação, a preocupação com a paralelização do código é praticamente eliminada, uma vez que a própria implementação do PETSc se encarrega de utilizar o MPI para conduzir a paralelização do código.

A operação de multiplicação matriz e vetor é uma das operações mais custosas em métodos iterativos para sistemas esparsos de equações lineares. Na execução do MEF sistemas dessa natureza surgem frequentemente, portanto, verificar a eficiência das bibliotecas do PETSc para essa operação é importante.

O código em *C/C++* a seguir demonstra como é simples a implementação paralela de uma multiplicação entre matriz e vetor utilizando o PETSc.

```
/* Comentarios neste formato omitem trechos
   de codigo para fins de simplicidade */
/* Inclui as bibliotecas do petsc */
int main()
{
    /* Declara as variaveis e inicializa PETSc e MPI*/
    // cria matriz
    MatCreate(PETSC_COMM_WORLD,&A);
    MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n);
    MatSetType(A,type);
    MatMPIAIJSetPreallocation(A,5,NULL,5,NULL);
    // preenche a matriz
    MatGetOwnershipRange(A,&Istart,&Iend);
```

```
        /* Rotina para o preenchimento da matriz */
        // monta matriz
        MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
        // cria vetores
        VecCreate(PETSC_COMM_WORLD,&X);
        VecSetSizes(X,PETSC_DECIDE,m*n);
        VecSetType(X,VECMPI);
        VecSet(X, one); // preenche o vetor X
        VecCreate(PETSC_COMM_WORLD,&Y);
        VecSetSizes(Y,PETSC_DECIDE,m*n);
        VecSetType(Y,VECMPI);
        // monta vetores
        VecAssemblyBegin(X);
        VecAssemblyEnd(X);
        VecAssemblyBegin(Y);
        VecAssemblyEnd(Y);
        double tic, tac;
        tic = MPI_Wtime();
        int k;
        MatMult(A,X,Y); // multiplica Y = A*X
        tac = MPI_Wtime();
        if(rank == 0) printf("wall time: %e\n", tac-tic);
        /* Destroi os objetos petsc e finaliza */
        return 0;
    }
```

A implementação consiste na inicialização do PETSc, que automaticamente inicializa o MPI, em seguida é criada a matriz paralela A, utilizando a estrutura abstrata do PETSc, a matriz é então preenchida e montada. A chamada à função *MatMPIAIJSetPreallocation()* torna a matriz paralela, alocando memória para as estruturas locais da matriz. A mesma coisa é feita para os vetores X e Y, sendo que Y não é preenchido,

uma vez que é o vetor que armazenará a resposta da multiplicação. Por fim, chama-se a rotina de multiplicação do PETSc que se encarrega de conduzir a multiplicação em paralelo de A por X e guardar em Y o resultado da multiplicação. O algoritmo termina com a desalocação das estruturas abstratas da memória e com a finalização do PETSc (que finaliza automaticamente o MPI). Na implementação paralela utilizando o PETSc, estruturas como vetores e matrizes são declarados e preenchidos levando-se em conta seus tamanhos globais, portanto as rotinas de montagem de matriz e de vetor é que se encarregam da passagem de mensagens dos componentes não locais, o que garantirá a paralelização destas estruturas entre os processos em execução.

Esta implementação garante escalabilidade e eficiência na execução em paralelo da multiplicação matriz-vetor, devido a forma como o PETSc se aproveita da natureza esparsa da matriz, reduzindo a comunicação entre processos o máximo possível. A mesma operação poderia ser implementada na arquitetura de memória distribuída apenas com o uso do MPI. Assuma que a matriz A é distribuída por linhas para um conjunto de processadores de tal forma que cada processador possua uma série de linhas adjacentes da matriz A e os elementos correspondentes do vetor X. Em uma implementação simples da multiplicação matriz-vetor com MPI, cada processo poderia informar aos outros processos os seus respectivos elementos do vetor X, assim todos os processadores teriam uma cópia com todos os elementos do vetor X. Cada processo conduziria a multiplicação matriz-vetor localmente e, por fim, todos os processos informariam os seus resultados para um processador que reuniria tudo no vetor resposta Y. Essa implementação é vantajosa, pois codificar a comunicação é simples e cada processador sabe exatamente quais mensagens deve passar e receber. A desvantagem é que este código não é escalável, afinal a quantidade de comunicação cresce com o aumento do número de colunas da matriz A, e memória é desperdiçada, uma vez que cada processador precisa guardar uma cópia completa do vetor X (Balay et al, 1997). A implementação com o PETSc contorna todos esses problemas e ainda elimina os esforços de programação que seriam necessários para uma implementação escalável e eficiente com o MPI.

O segundo código estudado é baseado em um exemplo disponível na biblioteca `src/ksp/pc/examples/tutorials/ex2.c` que resolve um sistema linear $Ax = b$ em para-

lelo com o uso de métodos iterativos (Balay et al, 2016; Saad et al, 2003). O PETSc possui um objeto chamado *KSP* que implementa diversos métodos iterativos do subespaço de Krylov. O código implementado utiliza a implementação do gradiente conjugado com o precondicionador de Jacobi (Saad et al, 2003) presentes no PETSc.

O código em *C/C++* a seguir ilustra a implementação paralela da resolução de um sistema linear utilizando o PETSc.

```
/* Comentarios neste formato omitem trechos
   de codigo para fins de simplicidade */
/* Inclui as bibliotecas do petsc */
int main(int argc,char **args)
{
    /* Declara variaveis e inicializa PETSc e MPI */
    // cria matriz A
    MatCreate(PETSC_COMM_WORLD,&A);
    MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n);
    MatSetFromOptions(A);
    MatMPIAIJSetPreallocation(A,5,NULL,5,NULL);
    MatSeqAIJSetPreallocation(A,5,NULL);
    MatSeqSBAIJSetPreallocation(A,1,5,NULL);
    // preenche a matriz
    MatGetOwnershipRange(A,&Istart,&Iend);
    /* Rotina para o preenchimento da matriz */
    // monta a matriz
    MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
    // cria vetores
    VecCreate(PETSC_COMM_WORLD,&b);
    VecSetSizes(b,PETSC_DECIDE,m*n);
    VecDuplicate(b,&x);
    /* preenche vetor do lado direito b */
    // monta vetores
    VecAssemblyBegin(b);
```

```
        VecAssemblyEnd(b);
        VecAssemblyBegin(x);
        VecAssemblyEnd(x);
        // cria o resolvedor linear
        KSPCreate(PETSC_COMM_WORLD,&ksp);
        // determina matriz do sistema e de precondicionamento
        KSPSetOperators(ksp,A,A);
        // obtem parametros do resolvedor linear
        KSPSetFromOptions(ksp);
        // resolve o sistema linear
        KSPSolve(ksp,b,x);
        /* Destroi os objetos petsc e finaliza */
        return 0;
    }
```

O algoritmo começa com a inicialização do PETSc, em seguida é criada a matriz que define o sistema linear A, a matriz é então preenchida e montada em paralelo. O mesmo é feito aos vetores b e x, sendo que b não é preenchido, pois é o vetor resposta. O algoritmo cria então o objeto *ksp* que implementa o contexto dos resolvedores lineares do subespaço Krylov e, na sequencia, o objeto *ksp* recebe a matriz A e a matriz do precondicionador que vão determinar os operadores associados ao sistema linear. Por fim, o objeto *ksp* resolve o sistema linear $Ax = b$ e depois os objetos são destruídos para que a memória seja desalocada e então o PETSc é finalizado. O contexto do resolvedor linear pode receber em tempo de execução qual o método iterativo e qual precondicionador para a resolução do sistema linear deve ser utilizado.

Bem como na implementação da multiplicação matriz-vetor, a implementação paralela da resolução de um sistema linear com o PETSc é extremamente simples e não exige esforço de programação com MPI, permitindo focar apenas na aplicação. Todas as complicações de programação inerentes à codificação com o MPI são abstraídas pelos objetos do PETSc que garantem boa escalabilidade e eficiência ao código paralelo. Os resultados obtidos com as implementações descritas nesta seção são apresentados no próximo capítulo.

3.3 Código Cardiax

A implementação do MEF para a modelagem da atividade elétrica cardíaca é complexa, dada a natureza multi-escala e multi-física do problema. Sendo assim, o que normalmente ocorre é a utilização de vários códigos para a simulação do problema como um todo. A implementação do Cardiax acopla as diversas etapas do MEF, utilizando técnicas de orientação a objetos e as ferramentas de computação científica e computação paralela apresentadas.

O código sequencial do Cardiax foi desenvolvido por Rocha et al (2014). A implementação é feita com a linguagem de programação *C++*, o que possibilita a aplicação de técnicas de orientação a objetos para dividir os componentes do problema em módulos, além de permitir a criação de abstrações de alto nível para os complexos problemas inerentes aos modelos matemáticos e ao MEF.

A implementação define classes que abstraem os detalhes e estruturas referentes ao MEF e aos modelos matemáticos que envolvem o problema. Estas classes foram agrupadas em módulos que formam os principais componentes do sistema. Na Figura 3.2 um diagrama ilustra a organização dos módulos no código. Os módulos são mostrados de baixo pra cima explicitando a dependência entre os mesmos, bem como a dependência das bibliotecas externas PETSc e Armadillo (Sanderson et al, 2010).

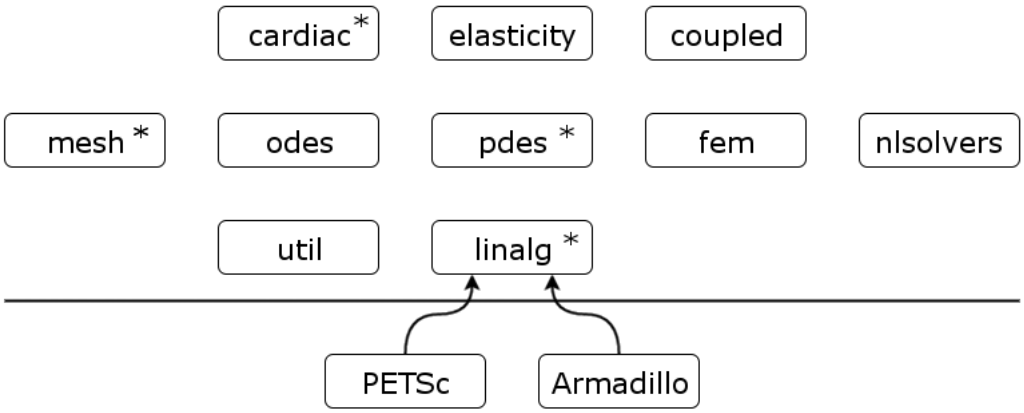


Figura 3.2: Módulos Cardiax (adaptado de Rocha et al (2014))

Uma vez que o presente trabalho focou na paralelização da atividade elétrica do coração, nem todos os módulos do sistema Cardiax foram paralelizados e, portanto, os

módulos pertinentes, ou seja, que sofreram algum tipo de paralelização, são os módulos: *linalg*, *mesh*, *pdes* e *cardiac*. No entanto, visto que os módulos possuem dependências entre si, uma breve explicação de cada um dos módulos é apresentada na subseção a seguir.

3.3.1 Módulos

O módulo *util* implementa funcionalidades básicas de leitura e escrita de arquivos, leitura de argumentos de entrada pela linha de comando, classes que controlam o passo no tempo de alguns métodos, dentre outras operações mais gerais.

O módulo *linalg* é um dos módulos fundamentais do sistema como um todo. Este módulo encapsula as bibliotecas externas PETSc e Armadillo e implementa as operações de álgebra linear computacional. O PETSc foi utilizado para definir desde estruturas como vetores e matrizes, a métodos para a solução de sistemas de equações lineares através dos métodos do subespaço Krylov. A biblioteca Armadillo foi utilizada para realizar operações básicas com matrizes e vetores pequenos (como por exemplo as matrizes de elementos dos MEF) e também para a realização de operações com tensores. Na paralelização do código este módulo recebeu especial atenção no sentido de definir as estruturas do PETSc como estruturas paralelas.

O módulo *mesh* é responsável pela implementação das classes que representam a malha computacional utilizada no MEF. Portanto o módulo lida também com as diversas operações relacionadas à malha, assim como operações de entrada e saída de arquivos de malha.

O módulo *odes* implementa métodos implícitos e explícitos para a solução numérica de modelos celulares representados através de sistemas de EDOs, bem como os próprios modelos. Já o módulo *pdes* implementa a representação de modelos pelas equações de Poisson e outras.

No módulo *fem* é feita a implementação das variadas classes para a solução numérica de EDPs utilizando o MEF. São representadas nesse módulo as classes para o tratamento de graus de liberdade, integração numérica e funções base do MEF. No módulo *nlsolvers* são feitas as implementações de de métodos resolvedores de sistemas de

equações não-lineares.

As classes que implementam os modelos que representam a atividade elétrica do coração, ou seja, os modelos monodomínio e bidomínio, são implementadas no módulo *cardiac*. Já o módulo *elasticity* implementa as classes para a representação do problema da elasticidade não-linear e equações da atividade mecânica no coração. Por fim, o módulo que implementa o modelo eletromecânico acoplado do coração é o *coupled*. Esse módulo acopla as classes dos módulos *cardiac* e *elasticity* e implementa a solução do problema eletromecânica através da solução de cada subproblema.

3.3.2 Paralelização da implementação sequencial

Os esforços de implementação mais importantes deste trabalho se concentram na paralelização do código sequencial do Cardiax. As alterações e implementações conduzidas na implementação sequencial são descritas a seguir.

Para paralelizar o MEF para a simulação da atividade elétrica cardíaca, paralelizaram-se os códigos dos módulos pertinentes ao problema. Como a paralelização é fortemente embasada na biblioteca PETSc, a primeira etapa da paralelização consistiu no encapsulamento de estruturas e funções paralelas do PETSc. Em seguida, foi feita a implementação de códigos de leitura e escrita em paralelo. E, finalmente conduziu-se a paralelização dos códigos dos modelos de Poisson e monodomínio, equações (2.1) e (2.2).

O módulo *linalg* implementa as operações de álgebra linear computacional relevantes ao MEF através do encapsulamento das estruturas e rotinas do PETSc e Armadillo. Na implementação sequencial, as estruturas de matrizes e vetores são definidas de maneira sequencial. Foi feita então a definição e criação de matrizes e vetores paralelos do PETSc, e com isso todas as operações envolvendo essas estruturas ao longo do código do Cardiax são conduzidas paralelamente através do PETSc. Além disto, foi feito o encapsulamento da estrutura de conjuntos de índices *IS* do PETSc e de suas operações, para o posterior uso no mapeamento da solução global para a solução local dos problemas.

Uma parte importante da paralelização é a garantia de que os dados estão representados de forma local (cada processo tem a sua porção dos dados) ou global (todos os processos têm todo o dado) quando necessário. Para isto, foi implementado um código

```
<?xml version="1.0"?>
<mesh celltype="tetrahedron" dim="3">
    <!-- DEFINIÇÃO DA MALHA -->
    <nodes size="9953">
        <!-- NÓS DA MALHA -->
        <node id="0" x="1.793987" y="-1.138025" z="1.711825"/>
        <node id="1" x="1.833126" y="-1.136228" z="1.443619"/>
        <node id="2" x="1.681251" y="-1.339227" z="1.513785"/>
        ...
    </nodes>
    <elements size="47320">
        <!-- ELEMENTOS DA MALHA -->
        <element id="0" v0="4010" v1="4011" v2="4012" v3="4013"/>
        <element id="1" v0="4011" v1="1994" v2="4014" v3="2684"/>
        <element id="2" v0="4012" v1="4014" v2="4015" v3="4016"/>
        ...
    </elements>
    <boundary celltype="triangle" dim="2">
        <!-- ELEMENTOS DO CONTORNO DA MALHA -->
        <element id="0" marker="10" v0="0" v1="1" v2="2"/>
        <element id="1" marker="10" v0="1" v1="3" v2="4"/>
        <element id="2" marker="10" v0="3" v1="0" v2="5"/>
        ...
    </boundary>
</mesh>
<poisson>
    <!-- CONDIÇÕES DE CONTORNO -->
    <dirichlet>
        <node id="10" marker="10" value="0.0" />
        <node id="11" marker="11" value="1.0" />
    </dirichlet>
</poisson>
```

Figura 3.3: Exemplo de XML de entrada para o problema de Poisson

para a leitura dos dados de entrada de maneira a garantir a representação local dos dados. Tanto o problema de Poisson quanto o problema monodomínio recebem como entrada um arquivo XML (Bray et al, 1997) que define a malha que representa o problema a ser resolvido. As Figuras 3.3 e 3.4 ilustram exemplos de arquivos de entrada para o problema de Poisson e monodomínio, respectivamente. Os arquivos contêm informações sobre os nós e elementos da malha, bem como sobre condições de contorno, condições iniciais e parâmetros do problema.

Na paralelização da entrada e dados, é feita a extração dos elementos-nós do arquivo XML que são então representados em um arquivo de entrada para o METIS. Com o uso do programa de particionamento de malhas *mpmetis* do METIS obtém-se um arquivo de particionamento, que define qual processo vai ler quais elementos e nós e dados referentes a estes elementos. Logo, em sua versão paralela, os problemas de Poisson e monodomínio recebem como entrada o arquivo XML de malha e o arquivo de particionamento do METIS separadamente.

A leitura começa com o processo 0 lendo o arquivo de particionamento e o re-

```
<?xml version="1.0"?>
<mesh celltype="hexahedron" dim="3">
    <!-- DEFINIÇÃO DA MALHA -->
    <nodes size="4305">
        <!-- NÓS DA MALHA -->
        <node id="0" x="-10000.0" y="-3500.0" z="1500.0"/>
        <node id="1" x="-10000.0" y="-3500.0" z="1000.0"/>
        <node id="2" x="-10000.0" y="-3000.0" z="1000.0"/>
        ...
    </nodes>
    <elements size="3360">
        <!-- ELEMENTOS DA MALHA -->
        <element id="0" v0="0" v1="1" v2="2" v3="3" v4="4" v5="5" v6="6" v7="7"/>
        <element id="1" v0="1" v1="8" v2="9" v3="2" v4="5" v5="10" v6="11" v7="6"/>
        <element id="2" v0="8" v1="12" v2="13" v3="9" v4="10" v5="14" v6="15" v7="11"/>
        ...
    </elements>
    <element_data type="fiber_transversely_isotropic">
        <!-- VALORES DE FIBRA DOS ELEMENTOS -->
        <element id="0">
            <fiber>1.0,0.0,0.0</fiber>
        </element>
        <element id="1">
            <fiber>1.0,0.0,0.0</fiber>
        </element>
        ...
    </element_data>
</mesh>
<electrophysiology>
    <!-- ESTIMULO INICIAL E PARAMÊTROS DO PROBLEMA -->
    <stimuli number="1">
        <stim start="0.0" duration="2.0" value="-35.714"
                          x0="-10000" x1="-8500"
                          y0="-3500" y1="-2000"
                          z0="-1500" z1="0"/>
    </stimuli>
    <parameters>
        <surface_to_volume> 0.14 </surface_to_volume>
        <theta_method> 0.5 </theta_method>
        <pcgtol> 1.0e-16 </pcgtol>
        <maxnz> 50 </maxnz>
        <sigma_l> 0.0001334 </sigma_l>
        <sigma_t> 0.0000176 </sigma_t>
        <sigma_n> 0.0000176 </sigma_n>
    </parameters>
</electrophysiology>
```

Figura 3.4: Exemplo de XML de entrada para o problema Monodomínio

presentando em um vetor, de maneira que o índice i do vetor representa o elemento i da malha e o valor guardado no índice representa a qual processo pertence aquele elemento. Todos os processos precisam da informação contida neste vetor para que saibam qual elemento devem ler do arquivo XML, desta forma o processo 0 transmite este vetor para os demais processos através de uma operação de *broadcast* do MPI. A partir do vetor de particionamento, cada processo conta quantos elementos terá de ler, para fins de alocação de memória e de representação local da malha.

Na sequencia os processos seguem para a leitura dos elementos do arquivo XML. É importante que os elementos sejam lidos antes dos nós, pois cada processo tem apenas a informação do particionamento dos elementos, com os elementos lidos basta consultar os nós que compõem cada elemento local e ler apenas estes nós. Na leitura dos elementos, cada processo lê o tipo de elemento da malha (por exemplo: tetraedro, hexaedro, etc.) e para cada elemento no arquivo XML extrai o número do elemento e verifica no vetor de particionamento se deve ou não guardar aquele elemento na estrutura local da malha.

Com os elementos lidos e representados na estrutura da malha, cada processo segue com a leitura dos nós. Para isso, descobre-se primeiro quais são os nós locais a serem lidos. Cada processo verifica os nós que compões os elementos locais e lê apenas estes do arquivo XML. Se o arquivo define fibras para os elementos, cada processo lê apenas aquelas que são definidas para os elementos locais. Da mesma forma, se o arquivo define elementos de contorno, cada processo verifica se o elemento é composto por nós locais, se sim o elemeno é lido. Com isto, tem-se a implementação paralela da entrada de dados. Com esta implementação a malha do problema é representada em porções locais, facilitando e otimizando operações posteriores na execução do MEF.

Visto que a simulação do MEF para atividade elétrica cardíaca pode lidar com problemas de larga escala e por consequência, gerar soluções que envolvem grande quantidade de dados, faz-se necessário uma forma eficiente de representar esses dados. Foi implementada uma escrita de dados utilizando arquivos XDMF (Elias et al, 2010) para a representação dos dados em paralelo. Neste formato de representação de dados paralelos, os dados são guardados em arquivos binários, no formato HDF5 (Key et al, 2011) enquanto a informação necessária para acesso aos dados é descrita no arquivo XDMF que é baseado no formato XML. A Figura 3.5 exemplifica um arquivo deste tipo para um problema estacionário.

Para conduzir a escrita dos dados de saída, cada processo escreve em um arquivo HDF5 os dados referentes às soluções locais do problema. Em seguida, o processo 0 coleta os números de nós e elementos pertencentes às malhas locais de todos os processos, assim como os nomes dos arquivos HDF5 de cada processo, para então escreves o arquivo XDMF que descreve a malha de saída e "aponta" para os arquivos HDF5.

Com a definição das estruturas do PETSc em paralelo, grande parte da execução do código sequencial já implementado já obteve certo nível de paralelização, visto que as operações de álgebra linear do PETSc já utilizadas passam agora a executar em paralelo. Portanto, foi necessário atentar apenas para alguns detalhes na implementação sequencial da simulação do MEF.

Tanto para o problema de Poisson quanto para o problema monodomínio, o primeiro detalhe a receber atenção é na etapa de montagem do sistema de equações do MEF.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="3.0">
    <Domain>
        <Grid Name="cube_hex100_02p_proc0" GridType="Collection" CollectionType="Spatial">
            <!-- Problema estacionário (apenas um passo de tempo) -->
            <Time TimeType="Single" Value="0.00"/>
            <!-- Malha de saída do processo 0 -->
            <Grid Name="rank_0" GridType="Uniform">
                <!-- Topologia ou conectividade dos nós -->
                <Topology TopologyType="Hexahedron"
                          NumberOfElements="684630">
                    <DataItem Format="HDF" DataType="Int" Dimensions="684630 8">
                        cube_hex100_02p_proc0.h5:/topology/connectivity
                    </DataItem>
                </Topology>
                <!-- Coordenadas dos nós -->
                <Geometry GeometryType="XYZ">
                    <DataItem Dimensions="711280 3" NumberType="Double" Precision="8" Format="HDF">
                        cube_hex100_02p_proc0.h5:/geometry/coordinates
                    </DataItem>
                </Geometry>
                <!-- Potencial transmembrânico resultante -->
                <Attribute Name="vm" AttributeType="Scalar" Center="Node">
                    <DataItem ItemType="HyperSlab" Dimensions="1 711280" Type="HyperSlab">
                        <DataItem Dimensions="3 2" Format="XML">
                            0 0
                            1 1
                            1 711280
                        </DataItem>
                        <DataItem Name="Points" Dimensions="1 711280" Format="HDF">
                            cube_hex100_02p_proc0.h5:/vertex_field/vm
                        </DataItem>
                    </DataItem>
                </Attribute>
                <!-- Vetor deslocamento resultante -->
                <Attribute Name="displacement" AttributeType="Vector" Center="Node">
                    <DataItem ItemType="HyperSlab" Dimensions="1 711280 3" Type="HyperSlab">
                        <DataItem Dimensions="3 3" Format="XML">
                            0 0 0
                            1 1 1
                            1 711280 3
                        </DataItem>
                        <DataItem Name="Points" Dimensions="1 711280 3" Format="HDF">
                            cube_hex100_02p_proc0.h5:/vertex_field/displacements
                        </DataItem>
                    </DataItem>
                </Attribute>
            </Grid>
        </Grid>
        <!-- Malhas de saída dos demais processos (truncado) -->
        ...
    </Domain>
</Xdmf>
```

Figura 3.5: Exemplo de XDMF de saída para um problema estacionário

Foi necessário trocar a criação da matriz de rigidez e vetores fonte e solução para as novas estruturas paralelas. Estas estruturas são definidas com seus tamanhos globais, mesmo na implementação paralela.

Como a solução do sistema linear do MEF é feita através de estruturas globais, em alguns pontos da implementação é necessária a obtenção dos dados locais para conduzir a escrita da solução local nos problemas de Poisson e monodomínio. Para isto, utilizaram-se as estruturas de conjunto de índices *IS* para mapear a estrutura global para um vetor local.

4 Resultados

Neste capítulo serão apresentados os resultados obtidos com os experimentos conduzidos para a simulação do MEF na atividade elétrica cardíaca. A primeira seção descreve o ambiente computacional onde foram conduzidos os experimentos, as próximas seções detalham os experimentos. A princípio foram feitos experimentos simples objetivando-se a familiarização e testes com a ferramenta PETSc, em seguida, foram conduzidos experimentos com o problema de Poisson em três dimensões e, por fim, foram feitos os experimentos com o modelo elétrico cardíaco monodomínio. Serão apresentados os resultados relacionados ao ganho de desempenho na execução do MEF quando implementado em paralelo.

4.1 Ambiente computacional

Os experimentos iniciais com a biblioteca PETSc foram conduzidos no *cluster* do Programa de Pós Graduação de Modelagem Computacional da UFJF. As máquinas, ou nós de execução, utilizadas possuem dois processadores AMD 6272, cada um com 16 núcleos (32 núcleos no total), e 32 GB de memória RAM. Todas as máquinas executam o sistema operacional Linux.

Posteriormente, os experimentos com o problema Poisson e com o problema monodomínio foram conduzidos no *cluster* Altix-XE do Laboratório Nacional de Computação Científica (LNCC, 2017). O *cluster* é formado por máquinas com 2 Processadores Intel Xeon E5520 2.27GHz Quad Core, totalizando 8 núcleos em cada máquina. As máquinas possuem 24 GB de memória RAM e executam o sistema operacional Linux.

4.2 Experimentos iniciais com PETSc

A parte inicial dos experimentos consistiu na execução dos códigos da operação de multiplicação matriz-vetor e da resolução de um sistema linear, com o intuito de verificar a

instalação da biblioteca e a eficiência das estruturas e operações da biblioteca PETSc, bem como familiarizar-se com a mesma.

O programa que testa a multiplicação matriz-vetor foi executado no *cluster* do Programa de Pós Graduação de Modelagem Computacional da UFJF. Foram feitos testes para diferentes dimensões de matrizes e número de processos. A Tabela 4.1 mostra os tempos de execução para cada dimensão de matriz e número de processos, e entre parenteses o *speedup* obtido. As medidas de tempo foram obtidas com o comando *MPI_WTime* do MPI. Foram feitas 5 execuções para cada teste e então, tomada a média aritmética dos tempos medidos.

Tabela 4.1: Tempo de execução multiplicação matriz-vetor em segundos (*speedup*)

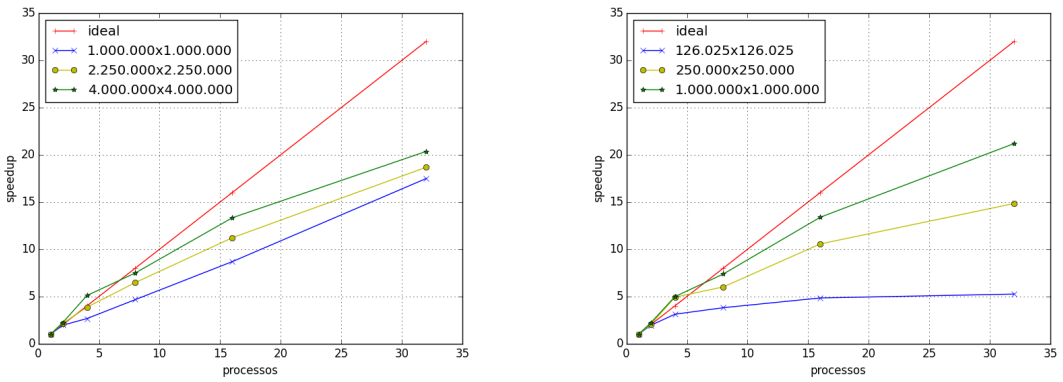
Processos	1000000×1000000	2250000×2250000	4000000×4000000
1	20.09 (1.000)	54.28 (1.000)	105.8 (1.000)
2	10.30 (1.950)	25.46 (2.131)	47.73 (2.216)
4	7.566 (2.655)	14.07 (3.857)	20.78 (5.091)
8	4.305 (4.666)	8.389 (6.470)	14.13 (7.487)
16	2.313 (8.685)	4.841 (11.21)	7.939 (13.32)
32	1.149 (17.48)	2.907 (18.67)	5.198 (20.35)

É possível notar que quanto menor a dimensão da matriz pior o *speedup*, o que é de se esperar, visto que nesse caso a demanda computacional é mais baixa e, portanto, o ganho de desempenho com o paralelismo é menos significativo. Com o aumento do número de processos o custo da comunicação se mostra mais expressivo para problemas de menores dimensão. Aumentando-se as dimensões da matriz a aceleração melhora, de tal forma que é possível notar que o programa é fracamente escalável, uma vez que com o aumento do número de processos a eficiência se mantém constante somente se aumentarmos as dimensões da matriz. A Tabela 4.2 mostra a eficiência calculada para os testes.

Tabela 4.2: Eficiência da multiplicação matriz-vetor.

Processos	1000000×1000000	2250000×2250000	4000000×4000000
1	1.000	1.000	1.000
2	0.975	1.065	1.108
4	0.663	0.964	1.272
8	0.583	0.808	0.935
16	0.542	0.700	0.832
32	0.546	0.583	0.636

Para as matrizes maiores ocorre uma aceleração super-linear para dois e quatro processos. Uma possibilidade para explicar a ocorrência desta aceleração é a ocorrência intensa de acerto de *cache*. A Figura 4.1(a) ilustra o desempenho do código obtido.



(a) (b)

Figura 4.1: *Speedups*: (a) multiplicação matriz-vetor e (b) resolução do sistema linear.

O experimento com a resolução de um sistema linear também foi conduzido no *cluster* do Programa de Pós Graduação de Modelagem Computacional da UFJF. Da mesma forma, foram feitos testes com diferentes tamanhos de sistema e números de processos. O sistema foi resolvido com o método do gradiente conjugado com o precondicionador de Jacobi, especificados em tempo de execução através das *flags* `-ksp_type cg -pc_type jacobi` para o precondicionador. A Tabela 4.3 mostra o tempo de execução para os diferentes tamanhos de sistema e entre parênteses o *speedup* calculado. A Figura 4.1(b) ilustra o gráfico dos *speedups*. Foram feitas 5 execuções e, então, mediu-se a média dos tempos obtidos. O tempo foi obtido através da *flag* `-log_view` disponível pelo PETSc para análise de desempenho.

Tabela 4.3: Tempo de execução sistema linear em segundos (*speedup*).

Processos	126025×126025	250000×2250000	1000000×1000000
1	3.732 (1.000)	14.36 (1.000)	113.6 (1.000)
2	1.931 (1.932)	7.119 (2.017)	51.86 (2.190)
4	1.198 (3.115)	2.922 (4.915)	22.81 (4.978)
8	0.980 (3.808)	2.390 (6.009)	15.40 (7.376)
16	0.771 (4.837)	1.360 (10.55)	8.486 (13.38)
32	0.710 (5.250)	0.968 (14.83)	5.362 (21.18)

Assim como no teste da multiplicação matriz-vetor, na resolução do sistema linear também é possível notar que o *speedup* melhora com o aumento da dimensão do sistema,

o que configura o programa como um programa fracamente escalável. O comportamento semelhante é esperado, já que na resolução de um sistema linear por métodos iterativos de Krylov a operação de maior demanda é a multiplicação matriz-vetor. Nota-se, também, a ocorrência de *speedup* super-linear, e o motivo é análogo ao apresentado para o caso da multiplicação matriz-vetor. A Tabela 4.4 mostra a eficiência calculada.

Tabela 4.4: Eficiência da solução do sistema linear em paralelo.

Processos	126025×126025	250000×2250000	1000000×1000000
1	1.000	1.000	1.000
2	0.966	1.008	1.095
4	0.778	1.228	1.244
8	0.476	0.751	0.922
16	0.302	0.659	0.836
32	0.164	0.463	0.662

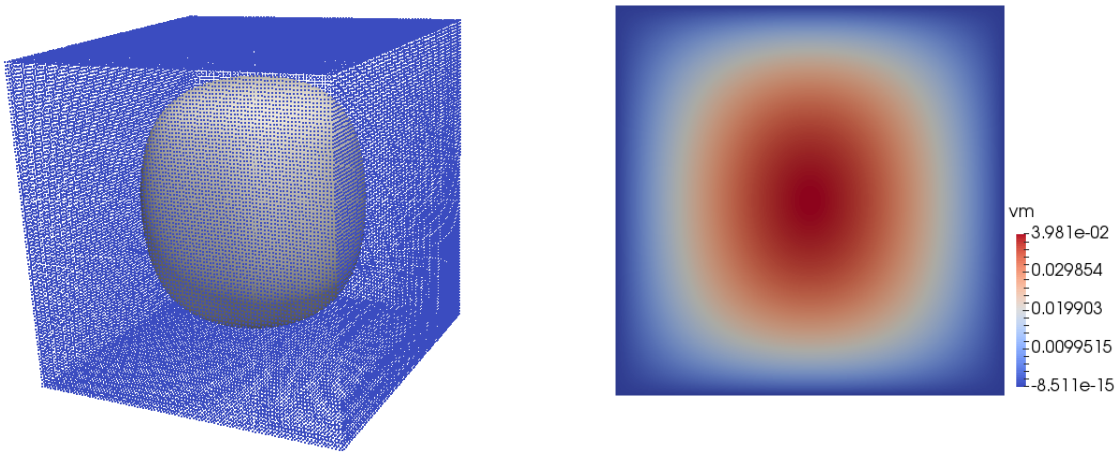
4.3 Experimentos com o problema Poisson

Os resultados para os experimentos com o problema Poisson foram obtidos no *cluster* Altix-XE do LNCC. As simulações consistiram na resolução do problema de Poisson em três dimensões. A resolução do sistema linear foi feita com o método do gradiente conjugado com o precondicionador de Jacobi. Foram feitos testes para três malhas com elementos do tipo hexaedro lineares de diferentes refinamentos. A malha 1, menos refinada, possui 1000000 elementos e 1030301 nós, a malha 2 possui 2744000 elementos e 2803221 nós e a malha 3 é a mais refinada e possui 8000000 elementos e 8282705 nós. A Figura 4.2(a) mostra o contorno da região e a Figura 4.2(b) mostra a distribuição espacial do potencial pela malha.

Tabela 4.5: Tempo de execuçao problema Poisson em segundos (*speedup*)

Processos	malha 1	malha 2	malha 3
1	21.88 (1.000)	59.67 (1.000)	- (-)
2	11.24 (1.947)	37.44 (1.594)	205.6 (1.000)
4	9.398 (2.328)	21.46 (2.780)	117.7 (1.747)
8	6.074 (3.602)	10.90 (5.476)	79.49 (2.586)

Para cada malha, o problema foi simulado de 1 a 8 processos e os tempos calculados, bem como os *speedups* obtidos são mostrados na Tabela 4.5. Como a malha 3 é muito



(a) (b)

Figura 4.2: Malha para os experimentos com o problema Poisson: (a) pontos do domínio e contorno da região da solução e (b) solução do problema Poisson para a distribuição do potencial *vm*

grande, o *cluster* não foi capaz de simular o problema quando executado sequencialmente, pois não houve memória suficiente. Isso ilustra a importância da computação paralela, afinal a simulação do problema para a mesma malha foi viabilizada quando executada em paralelo. Como não foi possível medir um tempo sequencial para esta malha, os *speedups* foram calculados assumindo o tempo de execução para dois processos como o tempo sequencial.

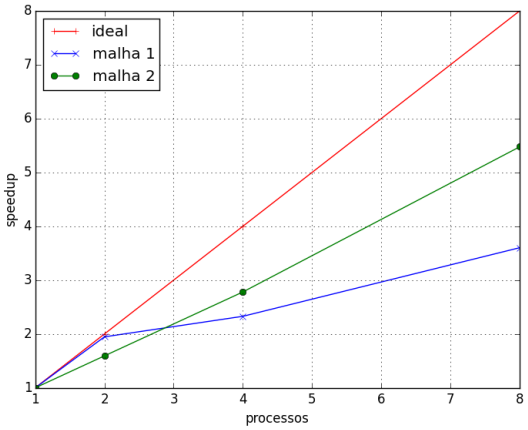


Figura 4.3: *Speedup* problema Poisson

A simulação do problema para a malha 1 apresentou um *speedup* bem próximo de 2 para a execução com 2 processos, mas em seguida a eficiência decaiu drasticamente. Um

possível motivo para este comportamento é que, como a malha possui poucos elementos, ocorreram muitos acertos de cache na execução com dois processos e, como são apenas dois processos, o custo da comunicação não contribuiu significantemente para o tempo de execução. O gráfico do *speedup* para as malhas 1 e 2 é mostrado na Figura 4.3. O código se mostra eficiente, pois com o aumento do tamanho do problema obteve-se aumento da eficiência, com exceção do caso de 2 processos. A Tabela 4.6 mostra a eficiência calculada.

Tabela 4.6: Eficiência problema Poisson

Processos	malha 1	malha 2
1	1.000	1.000
2	0.973	0.796
4	0.582	0.695
8	0.450	0.684

4.4 Experimentos com o modelo Monodomínio

Os experimentos com o modelo cardíaco monodomínio também foram conduzidos no *cluster* Altix-XE do LNCC. Para analisar a paralelização do simulador Cardiax com um problema mais realístico, foram feitas simulações para a resolução de um problema transiente para o potencial transmembrânico através de uma malha biventricular com 1645376 elementos e 318335 nós. O problema foi simulado para 100 passos de tempo e os sistemas lineares também foram resolvidos com o gradiente conjugado com o precondicionador de Jacobi. A Figura 4.4 mostra a solução do problema em quatro passos de tempos.

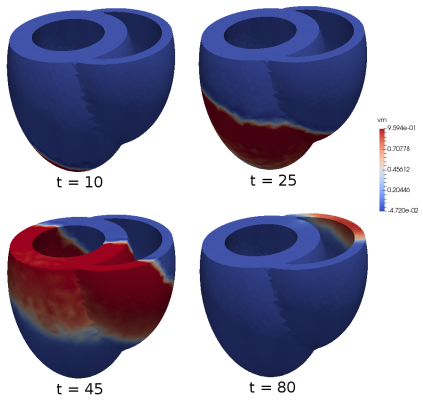
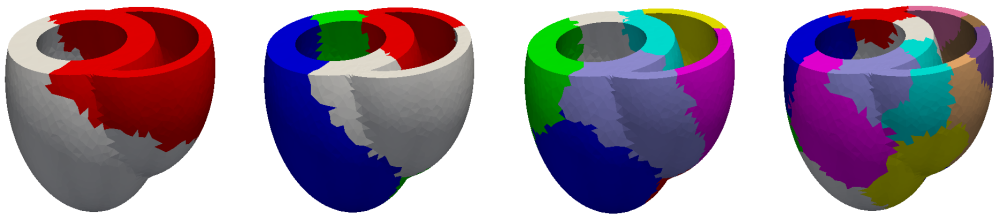


Figura 4.4: Distribuição do potencial transmembrânico, em quatro passos de tempos t, na simulação do modelo monodomínio para a malha biventricular.

A malha foi particionada para 2 e 4 processos e o particionamento dos elementos obtido pelo METIS é exibido na Figura 4.5 para até 16 processos. Os tempos de execução obtidos, bem como os e a eficiência são mostrados na Tabela 4.7.



(a) (b) (c) (d)

Figura 4.5: Particionamento da malha biventricular para (a) 2 processos (b) 4 processos (c) 8 processos e (d) 16 processos (cada cor denota a região do domínio destinada a cada processo diferente).

A simulação apresenta um ganho de *speedup* razoável, o que é suficiente para demonstrar como o paralelismo torna mais viável a simulação de problemas mais realísticos. A execução do problema sequencial tomou aproximadamente 6 horas e 52 minutos, enquanto que a execução para 4 processos tomou aproximadamente 2 horas e 32 minutos, o que representa um ganho de desempenho considerável.

Tabela 4.7: Tempo, *speedup* e eficiência problema monodomínio

Processos	Tempo (em segundos)	*Speedup*	Eficiência
1	24741.0	1.000	1.000
2	15956.0	1.550	0.775
4	9148.90	2.704	0.676

5 Conclusão

Este trabalho fez uso da computação paralela e ferramentas pertinentes para paralelizar a simulação do MEF para a atividade elétrica cardíaca, objetivando-se o ganho de desempenho na simulação de problemas de grande porte com intensa demanda computacional.

A princípio, foram feitas as implementações paralelas das operações de multiplicação matriz-vetor e da solução de um sistema linear, intencionando-se verificar a instalação da biblioteca e a eficiência das estruturas e operações da biblioteca PETSc, bem como familiarizar-se com a mesma. Após isto, fez-se a paralelização dos componentes responsáveis pela simulação do MEF para a atividade elétrica cardíaca do simulador Cardiax. Inicialmente paralelizou-se a simulação do problema de Poisson em três dimensões e, em seguida, paralelizou-se a simulação do modelo monodomínio.

Para verificar a validade das implementações paralelas da multiplicação matriz-vetor e da solução do sistema linear, foram feitos testes com diferentes dimensões de matrizes e, calculados os tempos de execução, *speedup* e eficiência obtidos. Os resultados se mostraram satisfatórios, uma vez que observou-se que os códigos são eficientes e escaláveis, validando a eficácia da biblioteca PETSc.

A implementação paralela do problema de Poisson foi submetida a testes com malhas de diferentes dimensões que descrevem problemas simples para a distribuição do potencial elétrico em um domínio cúbico. Verificou-se o ganho de desempenho computacional e, para o teste com a maior malha, foi possível demonstrar a viabilidade da implementação paralela, visto que, dada a dimensão da malha, o problema não pôde ser executado sequencialmente por falta de memória.

Por fim, para verificar a paralelização do modelo monodomínio, verificou-se o ganho de desempenho na simulação da distribuição do potencial transmembrânico em uma malha biventricular de geometria mais detalhada. Com a execução do problema para até 4 processos foi possível verificar a efetividade da implementação paralela, tornando a execução do problema consideravelmente menos custosa, o que é de grande valia na aplicação do modelo de um ponto de vista prático.

5.1 Trabalhos futuros

As simulações conduzidas no trabalho utilizaram apenas o precondicionador de Jacobi para a resolução dos sistemas lineares. Como uma possibilidade de exploração em trabalhos futuros, pretende-se testar o uso de diferentes precondicionadores e avaliar as suas implicações em desempenho.

Na atual implementação paralela do simulador, os nós da malha de entrada seguem a numeração fornecida no arquivo que foi fornecido pelo gerador de malha, apesar dos elementos serem particionados nos diferentes processos. Com isso, na etapa da montagem da matriz global do sistema de equações, pode ocorrer bastante comunicação. Pretende-se futuramente alterar esse detalhe da implementação e renumerar os nós de maneira sequencial em cada processo a fim de reduzir a comunicação na etapa de montagem final da matriz global do MEF.

Bibliografia

Amorim, R.; dos Santos, R. Solving the cardiac bidomain equations using graphics processing units. **Journal of Computational Science**, v.4, n.5, p. 370–376, 2013.

Balay, S.; Gropp, W. D.; McInnes, L. C. ; Smith, B. F. **Efficient management of parallelism in object-oriented numerical software libraries**. In: Modern software tools for scientific computing, p. 163–202. Springer, 1997.

Balaji, P.; Bland, W.; Gropp, W.; Latham, R.; Lu, H.; Pena, A. J.; Raffenetti, K.; Seo, S.; Thakur, R. ; Zhang, J. Mpich user's guide. **Argonne National Laboratory**, 2014.

Balay, S.; Abhyankar, S.; Adams, M.; Brown, J.; Brune, P.; Buschelman, K.; Dalcin, L.; Eijkhout, V.; Gropp, W.; Karpeyev, D.; Kaushik, D.; Knepley, M.; McInnes, L. C.; Rupp, K.; Smith, B.; Zampini, S. ; Zhang, H. **PETSc users manual. Technical Report ANL-95/11 - Revision 3.7**. Argonne National Laboratory, 2016.

Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; Maler, E. ; Yergeau, F. Extensible markup language (xml). **World Wide Web Journal**, v.2, n.4, p. 27–66, 1997.

Elias, R.; Braganholo, V.; Clarke, J.; SANTOS, I.; MATTOSO, M. ; COUTINHO, A. Using xml with large parallel datasets: Is there any hope? **Parallel Computational Fluid Dynamics: Recent Advances and Future Directions**, p. 358, 2010.

George, K. **METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.1.0**. Department of Computer Science Engineering University of Minnesota, 2013.

Karypis, G.; Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. **SIAM Journal on scientific Computing**, v.20, n.1, p. 359–392, 1998.

Folk, M.; Heber, G.; Koziol, Q.; Pourmal, E. ; Robinson, D. **An overview of the hdf5 technology suite and its applications**. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, p. 36–47. ACM, 2011.

de Computação Científica, L. N. **Cluster altix-xe**. `http://www.lncc.br/altix-xe/#2`, 2017. [Online; accessado em 27-Novembro-2017].

LeVeque, R. J. Finite difference methods for differential equations. **Draft version for use in AMath**, v.585, n.6, 1998.

Mena, A.; Ferrero, J. M. ; Matas, J. F. R. Gpu accelerated solver for nonlinear reaction–diffusion systems. application to the electrophysiology problem. **Computer Physics Communications**, v.196, p. 280–289, 2015.

Nash, M. P.; Panfilov, A. V. Electromechanical model of excitable tissue to study reentrant cardiac arrhythmias. **Progress in biophysics and molecular biology**, v.85, n.2, p. 501–522, 2004.

Pacheco, P. **An introduction to parallel programming**. Elsevier, 2011.

Plonsey, R. Bioelectric sources arising in excitable fibers (alza lecture). **Annals of biomedical engineering**, v.16, n.6, p. 519–546, 1988.

Rincon, M. A.; IShih, L. **Introducao ao metodo de elementos finitos–computaçao e análise em equaçoes diferenciais parciais**, 2013.

Rocha, B. M. **Modelagem da atividade eletromecânica do coração e os efeitos da deformação na repolarização**. 2014. Tese de Doutorado - Laboratório Nacional de Computação Científica.

Saad, Y. **Iterative methods for sparse linear systems**. 2. ed., Siam, 2003.

Sainte-Marie, J.; Chapelle, D.; Cimrman, R. ; Sorine, M. Modeling and estimation of the cardiac electromechanical activity. **Computers & structures**, v.84, n.28, p. 1743–1759, 2006.

Sanderson, C. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. 2010.

Sundnes, J.; Lines, G. T.; Cai, X.; Nielsen, B. F.; Mardal, K.-A. ; Tveito, A. **Computing the electrical activity in the heart**, volume 1. Springer Science & Business Media, 2007.

Vigmond, E.; Dos Santos, R. W.; Prassl, A.; Deo, M. ; Plank, G. Solvers for the cardiac bidomain equations. **Progress in biophysics and molecular biology**, v.96, n.1, p. 3–18, 2008.

Zhu, J.; Taylor, Z. ; Zienkiewicz, O. **The finite element method: its basis and fundamentals**. 6. ed., Elsevier, 2005.

Dos Santos, R. W.; Plank, G.; Bauer, S. ; Vigmond, E. **Preconditioning techniques for the bidomain equations**. In: Domain Decomposition Methods in Science and Engineering, p. 571–580. Springer, 2005.

Organization, W. H. **World health organization - the atlas of heart disease and stroke**. `http://www.who.int/cardiovascular_diseases/resources/atlas/en/`, 2017. [Online; accessado em 21-Novembro-2017].