UNIVERSIDADE FEDERAL DE SANTA CATARINA

CURSO DE GRADUAÇÃO EM ENGENHARIA MECÂNICA

Henrique Pacheco Garcia

# APLICAÇÃO DE MÉTODOS DE PARALELISMO NA BIBLIOTECA DE DESENVOLVIMENTO (EFVLib)

Florianópolis

2015

Henrique Pacheco Garcia

# APLICAÇÃO DE MÉTODOS DE PARALELISMO NA BIBLIOTECA DE DESENVOLVIMENTO (EFVLib)

Monografia submetida ao Programa de Bolsistas da Petrobras como conclusão da bolsa de iniciação científica.

Orientador: Clovis Raimundo Maliska

Florianópolis

2015

# Agradecimentos

Eu gostaria de expressar minha gratidão ao professor Clóvis Maliska e ao Programa Petrobras de Formação de Recursos Humanos pela disponibilidade de uma bolsa de iniciação científica e um lugar para poder desenvolver o atual trabalho.

Reitero minha gratidão à Tatiane Schveitzer e ao Axel Dihlmann por toda ajuda e suporte durante todo o período de bolsa, assim como fizeram todos os demais integrantes da equipe do Sinmec.

# Sumário

# Lista de ilustrações

# Lista de tabelas

# Lista de símbolos

| | |
|---|---|
| $(\xi, \eta, \zeta)$ | Coordenadas locais |
| $(x, y, z)$ | Coordenadas globais |
| $\Gamma$ | Coeficiente difusivo |
| $\mathcal{N}$ | Função de forma |
| $\varphi$ | Função geral definida no domínio de interesse |
| $\rho$ | Massa específica |
| $J$ | Matriz Jacobiana |
| $Q$ | Termo fonte |
| $S$ | Speedup |
| $E$ | Eficiência paralela |
| $T_p$ | Tempo de execução |

# Resumo

O emprego da simulação numérica na área de pesquisas de engenharia de petróleo e gás se tornou indispensável. Com a aplicação de novas técnicas e métodos, as simulações estão cada vez mais detalhadas e precisas. Mesmo com computadores potentes o tempo de computação necessário para a realização de todos os cálculos de um algoritmo complexo pode ser ainda muito alto. Para que as simulações atinjam um patamar mais alto de excelência, esse tempo precisa diminuir. O paralelismo consiste em processadores diferentes trabalharem em conjunto para resolver um problema, ao invés de apenas um fazer o trabalho todo. Fazendo uso desse método, a malha que compreende o espaço do reservatório de petróleo pode ser dividida de modo que cada processador tenha que realizar os cálculos necessários apenas de uma região do reservatório. Como os diferentes processadores podem trabalhar de forma simultânea, quanto mais processadores, menor será o tempo de resolução dos sistemas lineares. O importante no processo é balancear o trabalho de cada processador para que os tempos gastos sejam parecidos no momento da troca de informações. No Laboratório de Simulação Numérica em Mecânica dos Fluidos e Transferência de Calor (Sinmec) da UFSC foi desenvolvida a EFVLib, uma biblioteca que aplica o método de volumes finitos baseado em elementos na simulação numérica de reservatórios de petróleo em malhas não estruturadas híbridas. Aplicar a decomposição de domínio da computação paralela nesta biblioteca foi o objetivo do presente trabalho.

**Palavras-chave**: paralelismo, EFVLib, reservatórios de petróleo.

# 1 Introdução

## 1.1 Preliminares

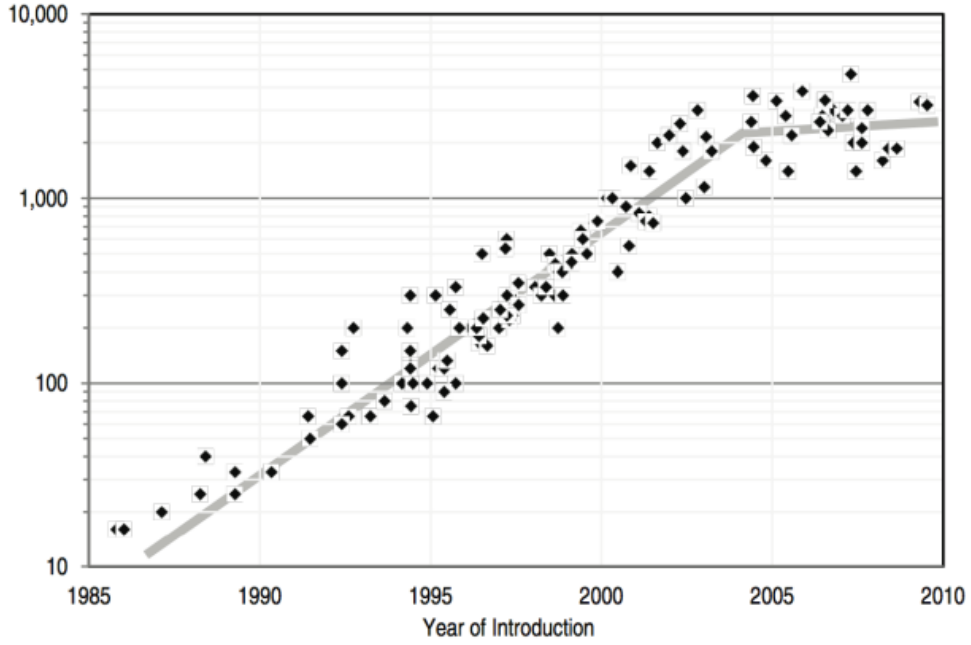
Com o avanço da tecnologia e do conhecimento, representações mais geometricamente exatas de reservatórios de petróleo podem ser alcançadas por meio de métodos numéricos. Juntamente com essa melhor representação está a possibilidade de se usar malhas cada vez mais refinadas. Mas até mesmo para computadores de alta performance atualmente existe um limite para quão complexa e refinada a malha pode ser, seja esse limite relacionado ao tempo de computação ou a falta de memória para armazenar as informações. Esforços tem sido empregados para melhorar a resolução dos sistemas lineares resultantes da simulação numérica, mas resultados mais eficientes podem ser encontrados fazendo mudanças no hardware, mais especificamente no tipo de arquitetura usada no computador.

A chave para resolução dos sistemas lineares encontra-se na unidade de processamento, ou seja, o processador. Vários parâmetros determinam a performance de um processador, mas a mais aclamada é sua frequência. A Figura 1 ilustra a mudança na frequência dos processadores ao longo dos anos. É possível perceber que está acontecendo uma saturação na frequência. Embora nem sempre maior frequência signifique melhor performance, essa continua sendo a regra geral. Além da saturação, essa melhora no processamento exige sistemas cada vez mais robustos de refrigeração e o preço é cada vez maior. A estratégia adotada atualmente é fazer uso de diversos processadores mais simples trabalhando em conjunto, ao invés de apenas um de alto desempenho. Deste modo, cada processador trabalha apenas com uma parte da carga computacional.

Uma coisa que merece ser notada é que apenas adicionar mais processadores não vai melhorar o tempo de simulação; é muito provável que até piore, já que os processadores usados são mais simples. É preciso também uma mudança no código numérico, deixando claro quais operações cada processador vai executar e garantir que ocorra uma divisão igualitária das tarefas computacionais, para evitar que um processador fique sobrecarregado e ele dite a performance do sistema.

As técnicas de computação paralela foram empregadas na EFVLib, uma biblioteca computacional desenvolvida no SINMEC, Laboratório de Simulação Numérica em Mecânica dos Fluidos e Transferência de Calor da UFSC. Nessa biblioteca estão compiladas as ferramentas para simulação 2 e 3D em malhas não estruturada híbridas, fazendo uso do EbFVM.

Figura 1 – Frequência em MHz de processadores ao longo dos anos




Fonte: SHANKLAND, 2012

## 1.2 Objetivos

Os objetivos do presente trabalho são:

- Auxiliar o mestrando Ederson Grein na implementação de uma metodologia eficiente de decomposição do domínio;
- Fazer adaptações na EFVLib para suportar computação paralela;
- Realizar testes para verificar a validade das mudanças realizadas.

# 2 Computação Paralela

## 2.1 Introdução

Atualmente, um grande obstáculo à maior excelência de algoritmos para simulação de reservatórios de petróleo é o tempo levado para a realização dos cálculos numéricos. Uma solução cada vez mais popular para este problema é a utilização de técnicas de computação paralela. O conceito de computação paralela pode ser definido como o uso de mais de um processador para realização de determinada tarefa computacional (PADUA, 2011). Embora a Lei de Moore ainda seja válida, seu fim pode estar próximo devido à enorme quantidade de energia que seria necessária para o funcionamento de um processador de elevado desempenho, além de um sistema de refrigeração sofisticado para dissipação do calor gerado. Na verdade, é mais prático fazer uso de diversos processadores mais simples para atingir a mesma performance (GEBALI, 2011).

O principal objetivo da computação paralela é diminuir o tempo que o computador leva para executar os cálculos necessários para determinada tarefa (TROBEC et al., 2009). Os computadores atuais podem armazenar uma quantidade enorme de dados, mas processar e realizar operações com esses dados é um trabalho árduo para um único processador. Com a computação paralela é possível dividir a tarefa entre diversos núcleos computacionais, cada um sendo responsável apenas por uma parte do problema, diminuindo a carga computacional nos processadores (DOROH, 2012). A seção seguinte mostra as diferenças entre a arquitetura de um processo serial e a de um paralelo.

## 2.2 Arquiteturas Serial e Paralela

De acordo com GEBALI (2011), algoritmos podem ser classificados baseando-se nas dependências das tarefas:

- Algoritmos seriais: São considerados algoritmos seriais aqueles em que as tarefas não podem ser executadas simultaneamente; a dependência entre os dados exige que elas sejam executadas uma após a outra;

- Algoritmos paralelos: Num algoritmo paralelo não existe dependência de dados entre suas tarefas, logo, é possível dividi-las entre diversos processadores para serem executadas simultaneamente. O processador com a maior carga computacional dita o desempenho, por isso é importante uma divisão mais igualitária possível das tarefas;

- Algoritmos serial-paralelos: Características no código permitem que as tarefas de um algoritmo serial-paralelo sejam agrupadas em estágios, onde as tarefas de cada estágio podem ser executadas de forma paralela, mas os estágios precisam ser executados de forma serial;

- Algoritmos não serial-paralelos: Em um algoritmo não serial-paralelo o fluxo das tarefas ou não segue um padrão, ou tem um padrão fixo, não podendo ser encaixado em nenhuma das classificações anteriores.

Apenas o algoritmo não é o bastante para que alguma operação seja executada em serial ou paralelo. É preciso também que elementos de hardware estejam interligados de maneira a garantir a funcionalidade desejada. A esta interligação é dado o nome de arquitetura computacional (DOROH, 2012). Para os computadores sequenciais a arquitetura comum é a Máquina de Acesso Aleatório (Random Access Machine), enquanto que para os computadores paralelos as três mais usadas são: Memória Compartilhada (Shared Memory), Memória Distribuída (Distributed Memory) e Memória Híbrida (Hybrid Memory).

Nas subseções a seguir os tipos de arquiteturas citadas acima serão brevemente descritas, com base nos trabalhos de DOROH (2012), DONGARRA (2003) e TROBEC et al. (2012).

### 2.2.1 Máquina de Acesso Aleatório

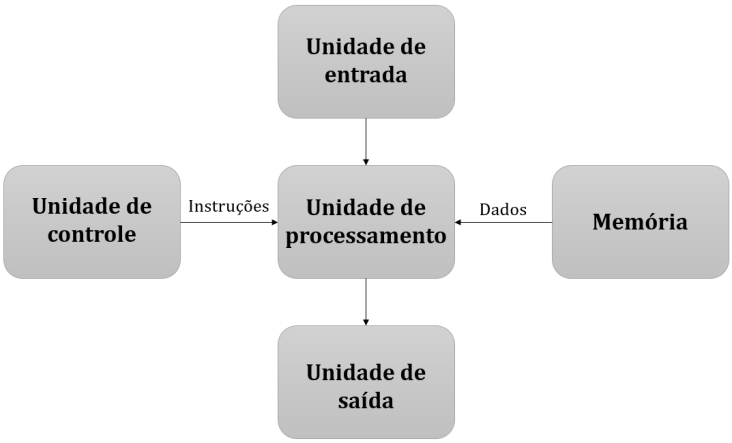
Na arquitetura serial o modelo mais usado é o da máquina de acesso aleatório. Nele, apenas uma sequência de instruções é executada por vez, sendo usado apenas um processador. A Figura 2 ilustra esse modelo. A unidade de processamento recebe dados da memória e instruções com quais operações e a ordem que serão executadas da unidade de controle. Como as operações são realizadas pela única unidade de processamento presente, o tempo computacional pode ser muito grande.

### 2.2.2 Memória Compartilhada

A arquitetura de memória compartilhada foi desenvolvida como uma derivação da máquina de acesso aleatório, tendo a mesma configuração, sendo apenas adicionados mais processadores. Todos os processadores compartilham uma memória global, garantindo um eficiente acesso e troca de dados. As diretivas OpenMP são as mais usadas nesta arquitetura para operações de sincronização de dados entre os processadores e outras operações necessárias. A Figura 3 ilustra esse tipo de arquitetura.

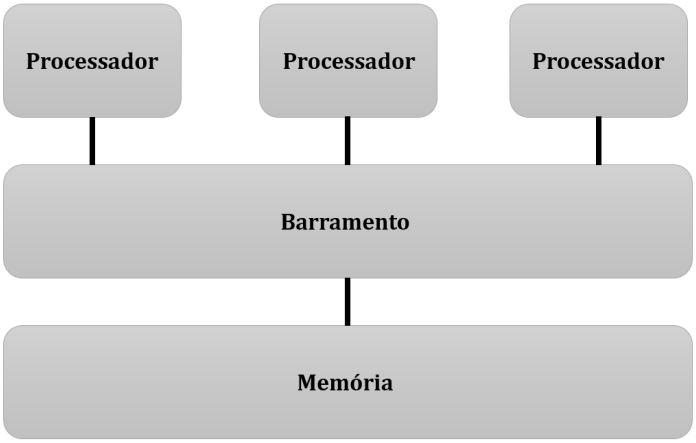
Também chamado de máquina de acesso aleatório paralela, este tipo de arquitetura tem suas vantagens e desvantagens. Como foi dito anteriormente, todos os processadores

Figura 2 – Arquitetura de um computador serial




Fonte: Do Autor - Adaptado de TROBEC et al. (2012)

Figura 3 – Arquitetura de memória compartilhada




Fonte: Do Autor - Adaptado de DONGARRA (2003)

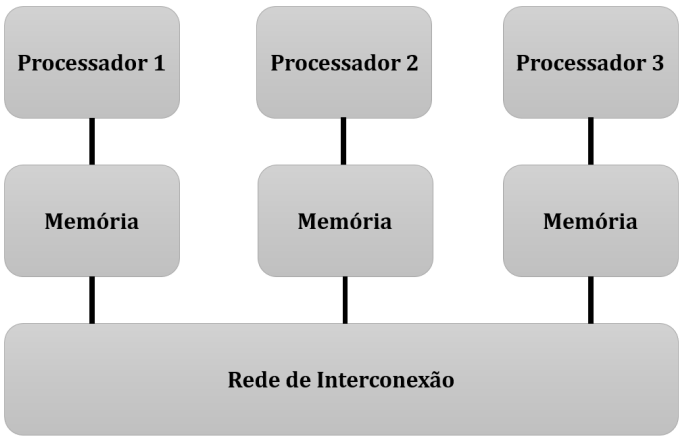
compartilham a mesma memória global, desta forma programar operações de leitura de dados é tão simples como se o código fosse serial. Mas esse compartilhamento de memória pode ser um problema para operações de escrita, já que dois ou mais processadores podem tentar escrever em um mesmo local do endereço de memória, ocorrendo a chamada colisão de memória. Por este motivo operações de escrita são mais complexas, sendo necessário bloquear o acesso ao local onde estas operações estão sendo realizadas. Outra desvantagem é que para um certo tamanho de memória global, existe um limite de quantos processadores podem fazer seu compartilhamento. Quanto mais processadores são usados, mais difícil e mais caro é construir essa arquitetura. Além disso, um computador com uma única

memória grande pode se tornar muito caro.

### 2.2.3 Memória Distribuída

A arquitetura de memória distribuída diferencia-se da memória compartilhada por cada processador possuir sua própria memória. As memórias dos diferentes processadores não possuem acesso direto entre si, mas a comunicação pode ser feita por meio da rede de interconexão. Essa comunicação é feita por meio de MPI (Message Passing Interface), onde um processador envia uma mensagem, enquanto um outro a recebe. Algumas operações de MPI podem envolver todos os processadores, como um processador mandar dados para todos os outros processadores, ou todos outros processadores mandarem seus dados para um processador. A Figura 4 ilustra a arquitetura de memória distribuída.

Figura 4 – Arquitetura de memória distribuída

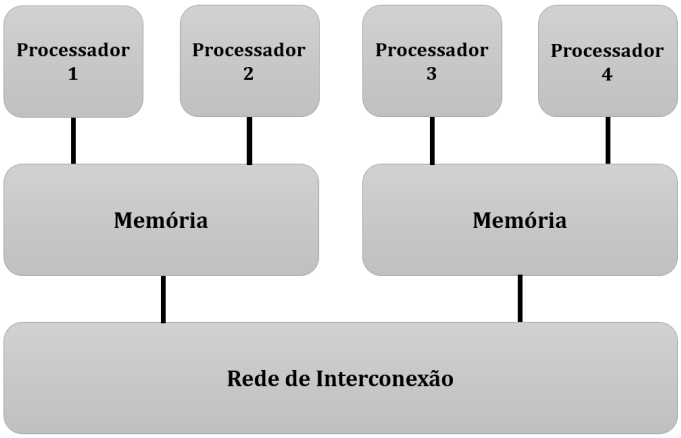



Fonte: Do Autor - Adaptado de DONGARRA (2003)

### 2.2.4 Memória Híbrida

A arquitetura de memória híbrida conta com características de tanto a memória compartilhada quanto da distribuída. O processador de cada nó computacional da memória distribuída é substituído por um conjunto de processadores, ou seja, várias arquiteturas de memória compartilhada são interligadas pela rede de interconexão. Assim, é possível tirar proveito da eficiente comunicação entre os processadores de um nó computacional, e também da possibilidade de usar mais processadores. A esquematização dessa arquitetura é exibida na Figura 5.

Figura 5 – Arquitetura de memória híbrida




Fonte: Do Autor

## 2.3 Medidas de Eficiência

Como foi visto anteriormente, o objetivo central da computação paralela é diminuir o tempo de execução dos cálculos computacionais. Existem várias maneiras de se quantificar essa melhora na performance. A mais comum e simples é a chamada *speedup*, definido como sendo a razão entre o tempo necessário para rodar o programa de maneira serial e o tempo para rodar em paralelo, como pode ser visto na Equação 2.1, onde $N$ representa a quantidade de processadores (GEBALI, 2011).

$$S_N = \frac{T_p(1)}{T_p(N)}. \tag{2.1}$$

Em um caso ideal, 100% do código estaria paralelizado e o tempo de comunicação entre os processadores seria desprezível. Neste caso, um programa rodando com $N$ processadores teria um *speedup* igual a $N$, sendo chamado de *speedup* linear. Como na maioria dos casos algumas porções do código são executadas serialmente, o *speedup* possui geralmente um valor menor que $N$, recebendo o nome de *speedup* sublinear. Raramente pode acontecer do *speedup* ser maior que o linear, sendo chamado de *speedup* super linear (DOROH, 2012).

Uma outra medida de eficiência, chamada de eficiência paralela, é definida como a razão entre o *speedup* e o número de processadores (PADUA, 2008):

$$E_N = \frac{S_N}{N}. \tag{2.2}$$

Num caso em que o *speedup* é linear, a eficiência paralela tem valor 1; caso ele seja

sublinear, seu valor será menor que 1; sendo super linear, seu valor será maior que 1.

Em situações normais, existem limites teóricos para o valor do *speedup*. Pequenas seções executadas em serial diminuem a eficiência e, por consequência, o *speedup* assume valores sublineares. A Lei de Amdahl leva em conta a fração paralelizada $f$ do código e a não paralelizada $1 - f$ durante o cálculo do *speedup* (DOROH, 2012). O tempo de execução de um código com essas variáveis com $N$ processadores pode ser expresso por:

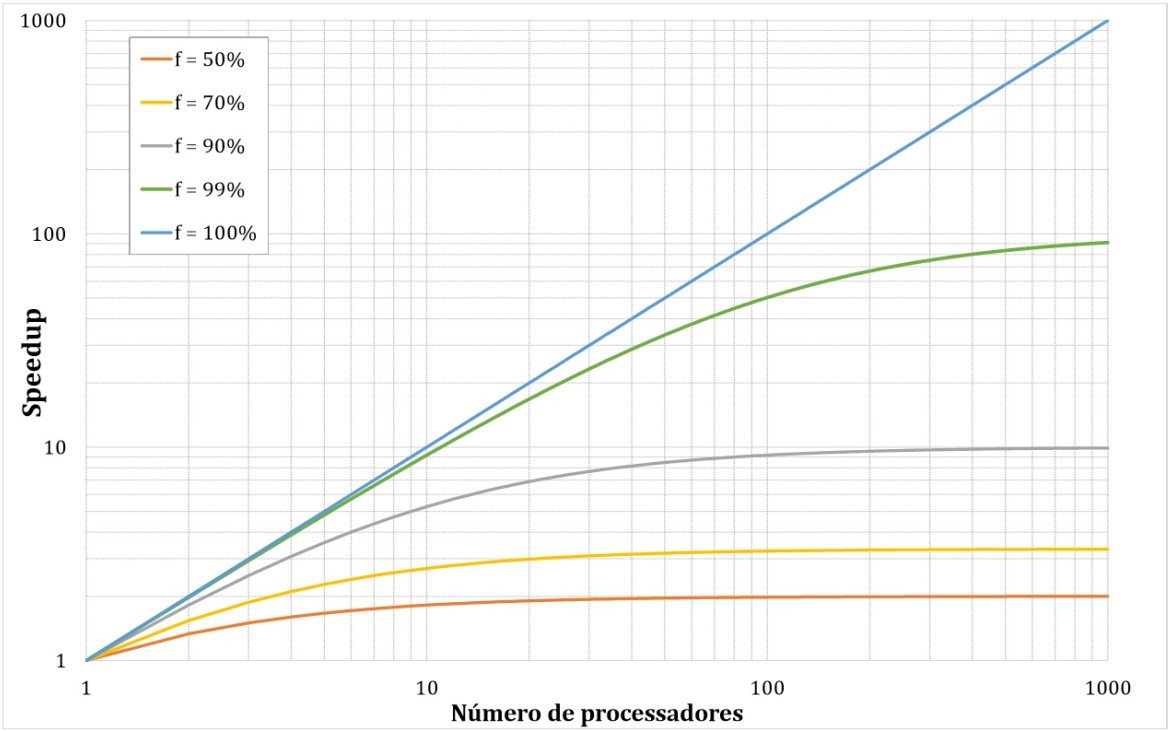
$$T_p(N) = f\frac{T_p(1)}{N} + (1 - f)T_p(1). \tag{2.3}$$

Portanto, como o *speedup* é definido como a razão entre o tempo de execução em serial e o em paralelo:

$$S_N = \frac{T_p(1)}{T_p(N)} = \frac{1}{f/N + (1 - f)} \tag{2.4}$$

O gráfico na Figura 6 mostra a variação do *speedup* em relação ao número de processadores para diversos valores da fração $f$. É possível perceber que quanto mais próximo de 100% está $f$, mais a curva assume um caráter linear; assim como ocorre a saturação do *speedup* após ultrapassar uma determinada quantidade de processadores.

Figura 6 – Limite teórico da Lei de Amdahl




Fonte: Do Autor

Um outro limite teórico definido foi a Lei de Gustafson, que difere da Lei de Amdahl por ser menos pessimista (GEBALI, 2011). Nela, o tempo usado como referência é o tempo de execução em serial.

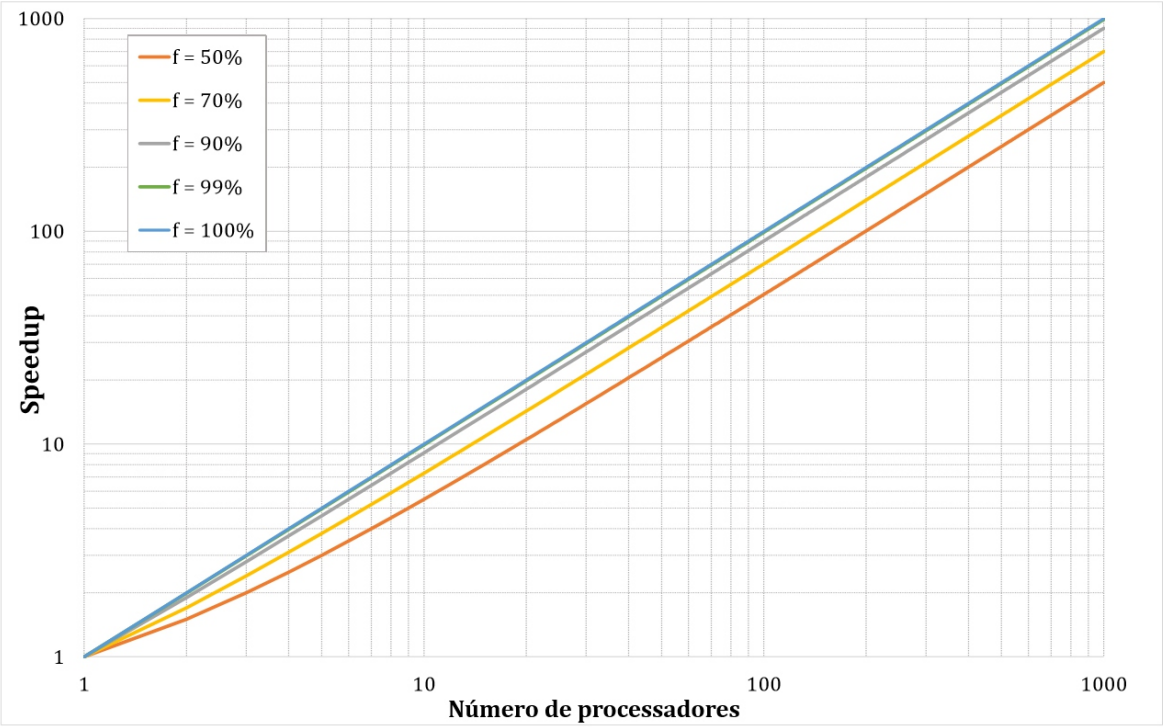
$$T_p(1) = (1 - f)T_p(N) + fNT_p(N), \tag{2.5}$$

Logo, o *speedup* é dado por:

$$S_N = 1 + f(N - 1). \tag{2.6}$$

A Figura 7 ilustra o gráfico da Lei de Gustafson, sendo possível notar o caráter menos pessimista devido a não existência da saturação e a maior proximidade das curvas ao que pode ser considerado o "regime linear".

Figura 7 – Limite teórico da Lei de Gustafson



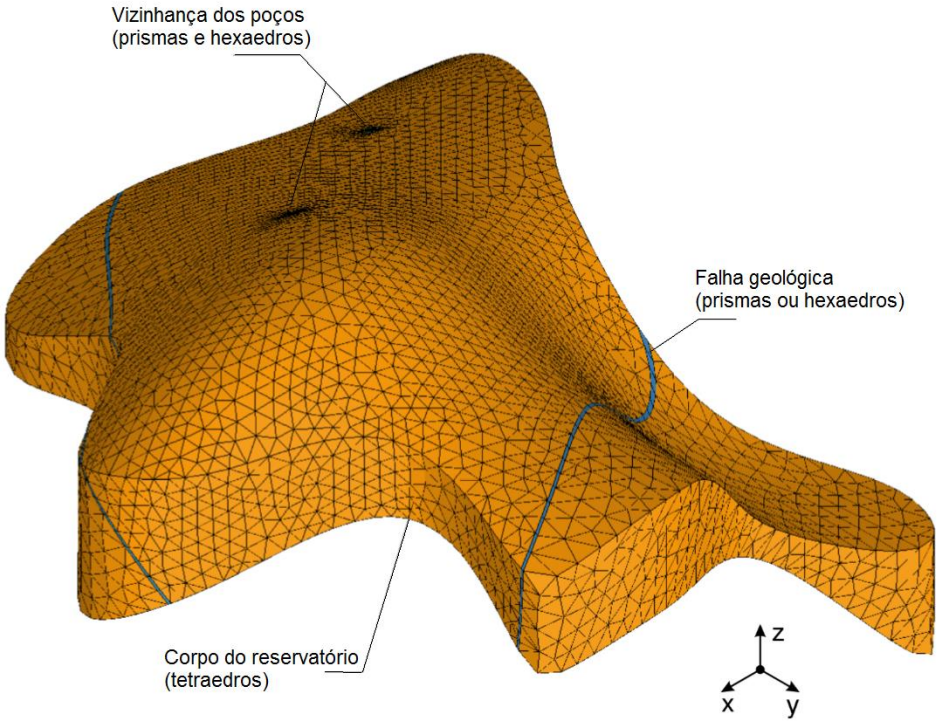

Fonte: Do Autor

# 3 Método dos Volumes Finitos Baseado em Elementos

## 3.1 Introdução

O uso do método dos volumes finitos permite a resolução das equações diferenciais quando aplicadas as leis da conservação de uma propriedade em um volume de controle. O método dos volumes finitos baseado em elementos consiste em aplicar a abordagem dos volumes de controle em uma malha de elementos finitos. Com isso, é possível representar regiões geométricas complexas por meio de malhas não estruturadas híbridas, garantindo uma maior flexibilidade método dos volumes finitos. A Figura 8 mostra a discretização de um reservatório fazendo uso de malha não-estruturada hibrida, mostrando sua versatilidade em representar uma região complexa.

Figura 8 – Discretização de um reservatório usando malha não-estruturada híbrida




Fonte: MALISKA et al., 2008

## 3.2 Entidades

Na EFVLib podem ser usadas tanto malhas estruturadas quanto não-estruturadas, sendo que as não-estruturadas são mais usadas por ser possível realizar uma melhor discretização com elas. Essas malhas são preenchidas por entidades geométricas chamadas de elementos (MALISKA et al., 2008). Em uma malha bidimensional são usados elementos formados por quadrados e triângulos, enquanto que em tridimensionais eles são tetraedros, hexaedros, prismas e pirâmides. Mais de um tipo de elemento pode ser usado ao mesmo tempo, sendo a malha então chamada de híbrida.

O contorno dos elementos é chamado de faceta, sendo linhas em malhas bidimensionais e superfícies em tridimensionais. Embora a representação geométrica das malhas seja realizada por elementos, as equações diferenciais são aproximadas levando em conta volumes de controle. Em uma malha bidimensional os vértices dos elementos são chamados de nós e o volume de controle é construído ao redor deles. Cada elemento que compartilha esse nó contribui para a formação de uma parte do volume de controle, sendo essa parte chamada de sub elemento. O contorno dos sub elementos é chamado de faces e são formados ligando o centróide do elemento ao ponto médio de suas facetas. O contorno do volume de controle é chamado de superfície de controle e é formado por um conjunto de faces. No EbFVM o campo das variáveis é associado aos vértices dos elementos (MALISKA et al., 2008). A Figura 9 ilustra essa representação. O procedimento para malhas tridimensionais e análogo e pode ser encontrado em (MALISKA, 2004).

## 3.3 Esquema Numérico

Uma dificuldade de se trabalhar com malhas não-estruturadas, diferentemente das malhas cartesianas, é que o sistema de coordenadas globais não coincide com as coordenadas locais de cada elemento, devido a distorções geométricas dos elementos (HURTADO, 2011). Uma maneira de se tratar disso é através do uso de chamado mapeamento, que oferece equações de transformação entre as coordenadas, sendo vantajoso por ser mais simples trabalhar com a discretização no nível local, onde o elemento tem uma representação regular.

O método utilizado para a transformação de coordenadas foi mediante funções de forma. Em um elemento, existe uma função de forma para cada vértice. No vértice $i$, a função de forma é $N_i$ valendo 1 para o vértice $i$ e 0 para todos outros. Desta forma, cada ponto $p = (x, y, z)$ nas coordenadas globais pode ser representado por

$$p = \sum_{i=1}^{n_v} \mathcal{N}_i(\xi, \eta, \zeta)\ p_i, \tag{3.1}$$

onde $n_v$ representa a quantidade de vértices, $p_i$ os vértices do elemento, e $(\xi, \eta, \zeta)$ as

Figura 9 – Entidades principais da malha


Fonte: MALISKA et al., 2008

coordenadas locais de $p$. As funções de forma possuem algumas propriedades: são continuas, deriváveis e devem obedecer a seguinte relação de unidade para cada ponto do sistema de coordenadas locais (ZIENKIEWICZ, 2000):

$$\sum_{i=1}^{n_v} \mathcal{N}_i(\xi, \eta, \zeta) = 1 \tag{3.2}$$

O EbFVM permite que uma função $\varphi$ definida nos vértices dos elementos seja aproximada baseando-se nas mesmas funções de forma definidas no mapeamento dos elementos (HURTADO, 2011) conforme pode ser visto a seguir:

$$\varphi(x, y, z) = \sum_{i=1}^{n_v} \mathcal{N}_i(\xi, \eta, \zeta)\ \varphi_i. \tag{3.3}$$

Como as funções de forma são diferenciáveis, aplicando o gradiente em $\varphi$ no sistema global de coordenadas chega-se em

$$\nabla\varphi = \nabla\left(\sum_{i=1}^{n_v}\mathcal{N}_i(\xi,\eta,\zeta)\ \varphi_i\right) = \sum_{i=1}^{n_v}\begin{pmatrix}\partial_x\mathcal{N}_i\\ \partial_y\mathcal{N}_i\\ \partial_z\mathcal{N}_i\end{pmatrix}\varphi_i = \begin{pmatrix}\partial_x\mathcal{N}_1 & \partial_x\mathcal{N}_2 & \cdots & \partial_x\mathcal{N}_{n_v}\\ \partial_y\mathcal{N}_1 & \partial_y\mathcal{N}_2 & \cdots & \partial_y\mathcal{N}_{n_v}\\ \partial_z\mathcal{N}_1 & \partial_z\mathcal{N}_2 & \cdots & \partial_z\mathcal{N}_{n_v}\end{pmatrix}\Phi_e, \quad (3.4)$$

onde

$$\Phi_e = \begin{pmatrix}\varphi_1 & \varphi_2 & \ldots & \varphi_{n_v}\end{pmatrix}^T \quad (3.5)$$

é o vetor com os valores de $\varphi$ nos vértices do elemento.

A Equação 3.4 como está definida não é conveniente, já que as funções de forma são dadas em termos das coordenadas locais, logo uma transformação de coordenadas se faz necessária para corrigir as derivadas. Aplicando a regra da cadeia,

$$\begin{pmatrix}\partial_\xi\mathcal{N}_i\\ \partial_\eta\mathcal{N}_i\\ \partial_\zeta\mathcal{N}_i\end{pmatrix} = \begin{pmatrix}\partial_\xi x & \partial_\xi y & \partial_\xi z\\ \partial_\eta x & \partial_\eta y & \partial_\eta z\\ \partial_\zeta x & \partial_\zeta y & \partial_\zeta z\end{pmatrix}\begin{pmatrix}\partial_x\mathcal{N}_i\\ \partial_y\mathcal{N}_i\\ \partial_z\mathcal{N}_i\end{pmatrix}. \quad (3.6)$$

A matriz

$$J = \begin{pmatrix}\partial_\xi x & \partial_\xi y & \partial_\xi z\\ \partial_\eta x & \partial_\eta y & \partial_\eta z\\ \partial_\zeta x & \partial_\zeta y & \partial_\zeta z\end{pmatrix} \quad (3.7)$$

é a matriz Jacobiana. Definindo

$$D \equiv \begin{pmatrix}\partial_\xi\mathcal{N}_1 & \partial_\xi\mathcal{N}_2 & \cdots & \partial_\xi\mathcal{N}_{n_v}\\ \partial_\eta\mathcal{N}_1 & \partial_\eta\mathcal{N}_2 & \cdots & \partial_\eta\mathcal{N}_{n_v}\\ \partial_\zeta\mathcal{N}_1 & \partial_\zeta\mathcal{N}_2 & \cdots & \partial_\zeta\mathcal{N}_{n_v}\end{pmatrix}, \quad (3.8)$$

a Equação 3.6 pode ser reescrita como

$$D.J^{-1} = \begin{pmatrix}\partial_x\mathcal{N}_i\\ \partial_y\mathcal{N}_i\\ \partial_z\mathcal{N}_i\end{pmatrix}. \quad (3.9)$$

Substituindo na Equação 3.4,

$$\nabla\,\varphi = J^{-1}D\ \Phi_e. \quad (3.10)$$

A matriz $J^{-1}D$ pode ser interpretada com um operador gradiente discreto.

## 3.4 Discretização da Equação de Difusão

De acordo com MALISKA (2004), a equação de difusão de uma propriedade genérica $\varphi$ pode ser escrita como

$$\frac{\partial}{\partial t}(\rho\varphi) = \nabla\cdot(\Gamma\nabla\,\varphi) + Q, \tag{3.11}$$

onde $\rho$ é a massa específica, $\Gamma$ é o coeficiente difusivo e $Q$ o termo fonte associado a propriedade $\varphi$. Como os casos a serem trabalhados serão em regime permanente e sem geração, a Equação 3.11 pode ser reduzida para

$$\nabla\cdot(\Gamma\nabla\,\varphi) = 0. \tag{3.12}$$

A discretização em volumes finitos é feita integrando a equação acima em cada volume de controle. Sendo $\psi$ um volume de controle, integrando a Equação 3.12 em $\psi$ e aplicando o teorema da divergência

$$\int_{\Psi}\Gamma\nabla\,\varphi\dot{\forall} = \int_{\Psi}(\Gamma\nabla\,\varphi)\cdot\hat{\mathbf{n}}\dot{A} \approx \sum_{f}(\Gamma\nabla\,\varphi)_f\cdot\Delta\mathbf{A}_f. \tag{3.13}$$

Usando o operador gradiente discreto definido na Equação 3.10,

$$\int_{\Psi}\Gamma\nabla\,\varphi\dot{\forall} \approx \left(\Gamma J^{-1}D\;\Phi_e\right)\cdot\Delta\mathbf{A}_f. \tag{3.14}$$

# 4 Metodologia

## 4.1 Introdução

Visando atingir o objetivo de melhorar a performance com a programação paralela, foi adotado uma metodologia baseada na decomposição do domínio, já que operações relacionadas a malha tomam grande parte do tempo de simulação. Nessa metodologia é feita a divisão da malha do reservatório e cada processador trabalha com apenas uma parte de dessa divisão. Devido a dependência que alguns vértices tem com outros, uma comunicação entre os processadores é imprescindível para o processo. Para isso, são usadas as rotinas MPI (Message Passing Interface).

## 4.2 Representação por Grafos

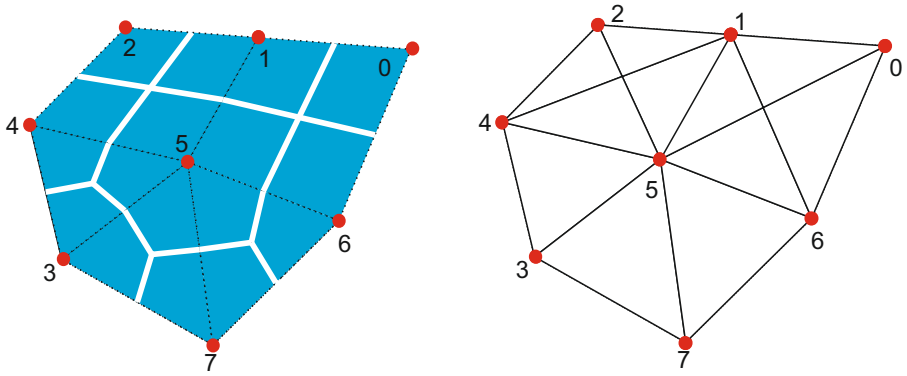
Uma maneira simples de encarar a decomposição do domínio seria fazer uma divisão dos elementos da malha e os distribuir entre os processadores disponíveis. Mas como foi visto anteriormente, na EFVLib o campo das variáveis está armazenado nos vértices dos elementos, logo, operações desnecessárias podem ser realizadas já que um vértice pode estar em mais de um domínio computacional. Portanto, foi adotada uma divisão baseada nos vértices, garantindo que cada vértice pertença a apenas um domínio computacional.

Para facilitar a divisão dos vértices e a comunicação entre os processadores que possuam vértices pertencentes ao mesmo elemento foi empregada uma representação da malha por grafos (DONGARRA, 2003). O cálculo do fluxo de uma propriedade pela face de um elemento requer dados armazenados nos vértices, logo, é possível imaginar uma conexão entre um vértice e os outros vértices de todos os elementos a que ele pertença. Essa conexão pode ser representada por grafos, em que os vértices são os nós do grafo e vértices pertencentes ao mesmo elemento são conectados por uma aresta. A Figura 10 ilustra uma malha e sua representação por grafos.

Durante a divisão da malha arestas são cortadas, e, consequentemente, é necessária uma troca de dados entre os domínios computacionais a que os nós do grafo foram alocados. Portanto, minimizar o número de arestas cortadas é essencial para uma divisão de qualidade.

A biblioteca Metis (METIS) foi utilizada para realizar a divisão dos grafos. Essa biblioteca disponibiliza dois métodos para a divisão: bisseção recursiva chamada `Metis_PartGraphRecursive` e um esquema k-way, o `Metis_PartGraphKWay`. O segundo foi usado como padrão nas simulações por possuir vantagens, como requerir menos tempo

Figura 10 – Representação por grafos de uma malha




Fonte: Do Autor

(KARYPIS & KUMAR, 1998). Mais informações sobre os métodos podem ser encontradas em (KARYPIS, 1998) e (KARYPIS & KUMAR, 1998), respectivamente.

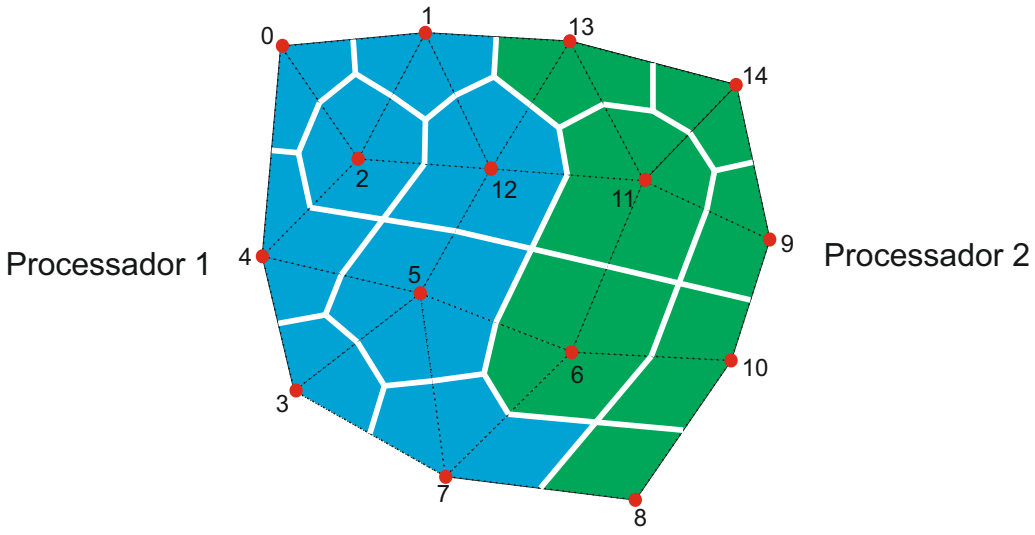
## 4.3 Nós Fantasma (Ghost Nodes)

Como foi visto na Seção 3.3, o cálculo das funções de forma necessita de todos os vértices de determinado elemento. Isso, juntamente com o fato que os valores dos campos definidos ao longo da malha estão armazenados nos vértices, gera a necessidade de se entender o subdomínio de cada processador para conter os vértices restantes do elemento. Os vértices vindos de outro subdomínio são chamados de nós fantasmas, sendo seu papel apenas armazenar valores, sem que qualquer computação seja realizada neles. Quando o valor de um campo mudar em um determinado nó, esse valor deve ser atualizado em todos os subdomínios que possuam esse nó como nó fantasma.

Seja a Figura 11 a representação de uma malha que passou pelo processo de divisão. A região azul foi alocada a uma unidade computacional, enquanto a verde de alocada a outra. Caso apenas os vértices de cada subdomínio fossem alocados as suas respectivas unidades computacionais, a simulação não seria bem-sucedida, já que os nós 1, 5, 7 e 12 do subdomínio 1 requerem valores armazenados nos nós 6, 8, 11 e 13, sendo o oposto também verdade. A Figura 12 mostra cada subdomínio após a adição dos nós fantasma.

Para melhorar a organização e facilitar as operações computacionais, os nós dos subdomínios possuem índices locais. Primeiro é feita a numeração dos nós locais e os nós fantasmas são alocados aos últimos índices. A Figura 13 mostra como ficam os subdomínios da Figura 12 com os índices locais.

Figura 11 – Divisão da malha em dois subdomínios




Fonte: Do Autor

## 4.4 Sistema de Equações Lineares

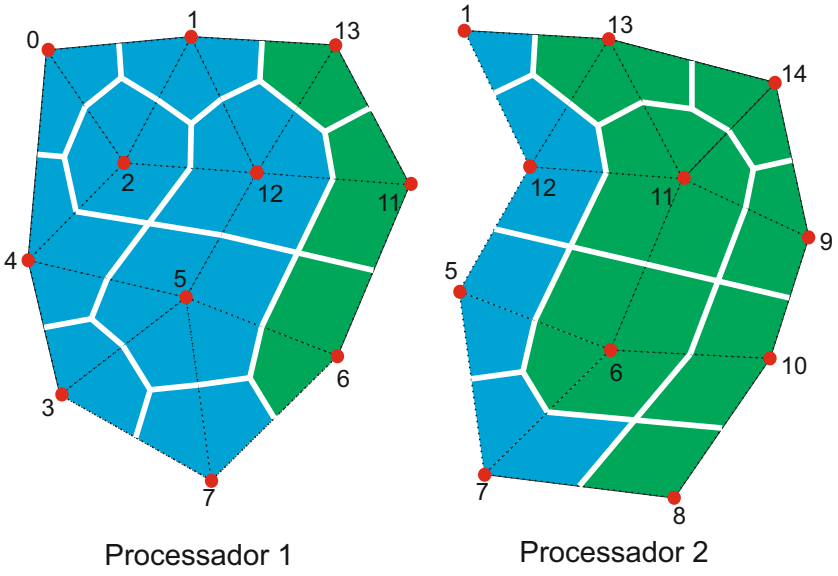
A solução da equação da difusão, ou de qualquer outra equação que pode ser implementada no atual modelo de simulação, é gerada através da resolução de um sistema de equações lineares. Essa é umas das operações que mais consume tempo computacional. Para sua resolução foi usada a biblioteca externa Portable, Extensible Toolkit for Scientific Computation (PETSc) (BALAY et al., 2013). Considerando um sistema de equações lineares como sendo $Ax = b$, a PETSc reserva uma linha da matriz do sistema para cada processador. Informações sobre como a PETSc resolve os sistemas lineares podem ser encontradas em BALAY et al. (2013).

## 4.5 Código Numérico

O código desenvolvido pode ser usado por qualquer usuário da EFVLib que queira rodar seu código em paralelo. Seu uso não exige que o usuário possua um extenso conhecimento em programação paralela, apenas o necessário para entender a chamada de cada função, graças a interface de uso simples que foi desenvolvida. Ao ser criada, foi utilizado o sistema Test Drive Development (TDD), onde após cada alteração no código foram criados testes para verificar a validade dessas alterações.

Além das linhas de código escritas pelo mestrando Ederson Grein e dos testes realizados pelo bolsista Henrique Pacheco, algumas bibliotecas externas foram usadas durante o desenvolvimento do código: Metis 5.1.0, PETSc 3.4.4 e Boost 1.5.5 (Boost, 2015).

Figura 12 – Subdomínios separados com os nós fantasma



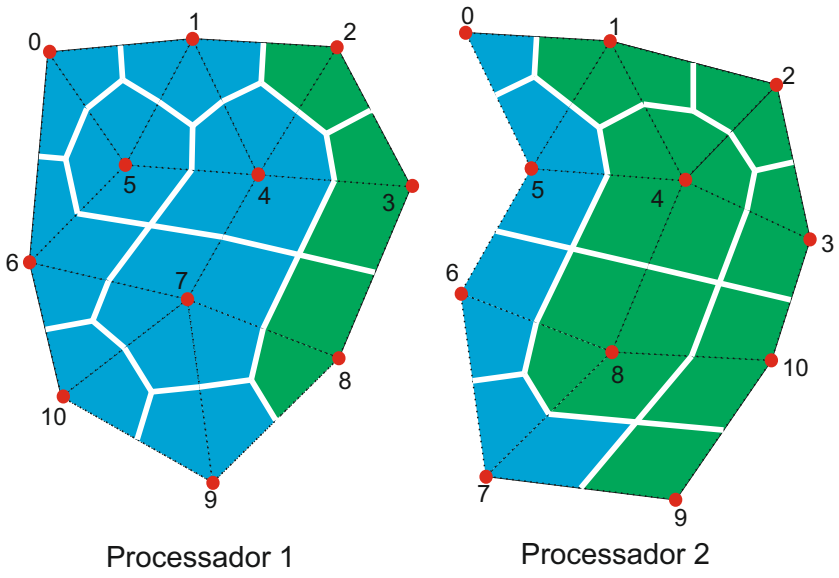

Fonte: Do Autor

A Metis, como foi explicado na Subseção 4.2, realiza a divisão do domínio; a PETSc é usada para resolver os sistemas lineares; e Boost fornece uma interface simplificada para as funções MPI.

O pacote de ferramentas para computação paralela na EFVLib possui quatro classes principais: `VertexPooler`, `GridDivider`, `FieldDivider` e `FieldOnVerticesSynchronizer`. A classe `VertexPooler` é responsável por fazer o *pool* dos vértices nos subdomínios usando as funções da Metis. Como padrão, o `partitionMethod` é o método k-way, mas a bisseção também pode ser usada. A principal função da classe é a `divide`, que possui dois parâmetros de entrada: `GridData`, uma estrutura computacional que armazena os dados essenciais da malha (MALISKA et al., 2011); e `nParts` representando o número de partições da malha. A classe `VertexPooler` é abstrata, logo é preciso criar uma classe derivada para implementar a função `computeWeightArray`. Essa função é responsável por definir pesos para divisão dos vértices da malha. Duas classes derivadas foram criadas: `UnweightedVertexPooler` (divisão sem peso) e `InverseDistanceWeightedVertexPooler` (peso baseado no inverso da distância).

A classe `GridDivider` faz uso de um `VertexPooler` para dividir a malha e criar o `GridData` local de cada subdomínio a partir do `GridData` global. A função `divide` é chamada para dividir a malha. Caso o processador que fazer a chamada da função seja o processador mestre (rank 0), são passados como parâmetros o `GridData` global e um `VertexPooler`. No caso de ser qualquer outro processador, nenhum parâmetro é passado, ele simplesmente recebe do processador mestre o `GridData` local do seu subdomínio.

Figura 13 – Subdomínios com índices locais




Fonte: Do Autor

A classe `FieldOnVerticesSynchronizer` é responsável por atualizar os valores nos nós fantasma. A função `synchronize` é usada para esse propósito, sendo chamada por todos os processadores passando como parâmetro o campo a ser atualizado. Essa classe possui a estrutura `SynchronizerVerticesVector`, que é usada nas rotinas MPI para uma comunicação eficiente entre os processadores.

A classe `FieldDivider` possui as funções `scatter` e `gather`, usadas para "espalhar" um campo global para locais e "juntar" campos locais em um global, respectivamente. A função `scatter` é usada no início da simulação, para o processador mestre distribuir os dados para os subdomínios locais. A chamada pelo processador mestre passa como parâmetros de entrada o campo global e o local, enquanto que a chamados dos demais processadores só usa o campo local. A função `gather` é usada para coletar dados dos processadores para o processador mestre e sua chamada segue o mesmo esquema da função `scatter`.

# 5 Resultados

Como o objetivo do presente trabalho é desenvolver um pacote de ferramentas para programação paralela na EFVLib, não é preciso rodar várias simulações para validar a biblioteca. Neste capítulo serão apresentados os resultados da simulação de dois casos, ambos problemas de difusão pura em regime permanente sem geração de calor. O primeiro caso possui uma geometria simples em 2D com vários elementos, enquanto que o segundo é uma geometria 3D com mais elementos. Ambos casos foram rodados no cluster do SINMEC, o Laboratório de Simulação Numérica em Mecânica dos Fluidos e Transferência de Calor Computacional da Universidade Federal de Santa Catarina. O cluster tem 64 nós computacionais, cada um com 8 Gb de RAM e 8 núcleos com 2,00 GHz de frequência de clock.

## 5.1 Caso 1

O primeiro caso é um problema de difusão de calor. A malha empregada foi bidimensional, hibrida e não estruturada. Existem 288025 nós e 574048 triângulos. As condições de contorno empregadas nas fronteiras foram as seguintes: na fronteira superior foi definida que a temperatura é conhecida e varia conforme a equação

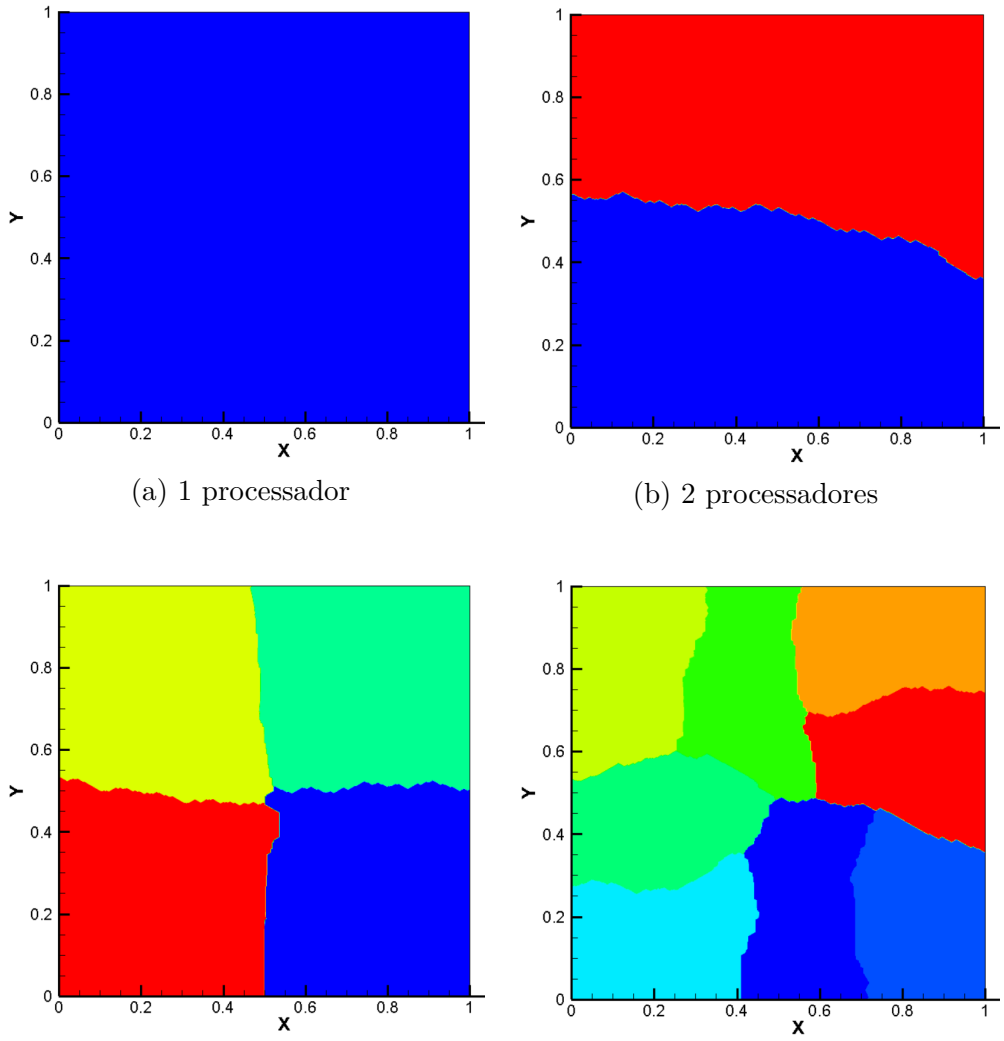
$$\frac{\sin(\pi * x) * \sinh(\pi * y)}{\sinh(\pi)}. \tag{5.1}$$

Nas demais fronteiras foi empregada condição de contorno de Dirichlet nula. O domínio é suposto como homogêneo e isotrópico, além de se encontrar em regime permanente.

O sistema de equações lineares foi resolvido com auxílio da PETSc. Para isso o caso foi executado várias vezes usando 1, 3, 4, 8, 16, 32, 64 e 128 processadores. A Figura 14 ilustra a divisão do domínio de acordo com a quantidade de processadores utilizada. Cada cor representa um domínio computacional e entre duas cores diferentes se encontram os nós fantasma.

Com a mudança no número de processadores usados, operações realizadas no decorrer do código, como a leitura da malha, sua divisão e posterior construção, levam tempos diferentes para serem executadas. A Tabela 1 mostra o tempo que cada operação levou para ser realizada com a quantidade de processadores mostrada. A leitura da malha é sempre realizada apenas pelo processador mestre, logo seu tempo de execução idealmente seria o mesmo para qualquer quantidade de processadores, mas devido a variações na carga do processador, seu tempo teve uma pequena variação. A divisão da malha consiste na chamada da função `divide`. Como para apenas 1 processador não é

Figura 14 – Divisão da mallha usando diferentes números de processadores




(a) 1 processador

(b) 2 processadores

(c) 4 processadores

(d) 8 processadores

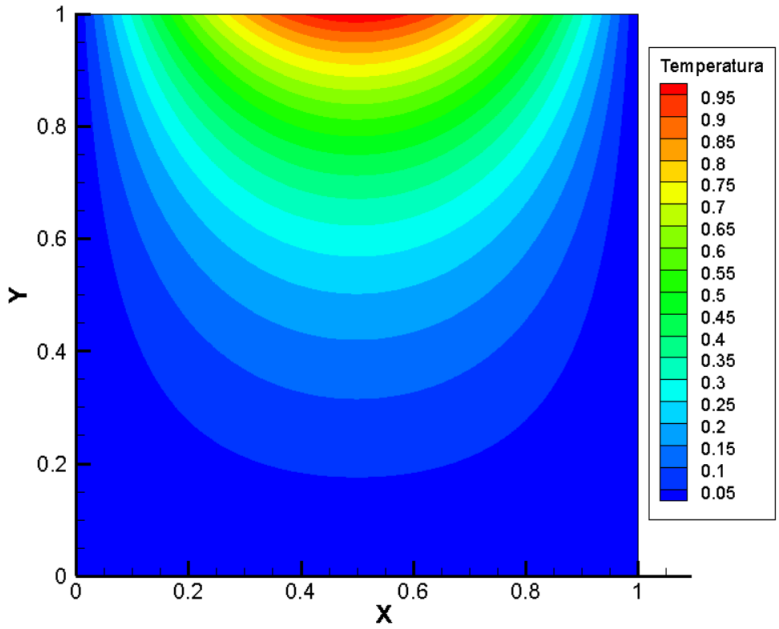
Fonte: Do Autor

Tabela 1 – Tempo de cada operação para diferentes números de processadores

| **Operação** | **#1** | **#2** | **#4** | **#8** | **#16** | **#32** | **#64** | **#128** |
|---|---|---|---|---|---|---|---|---|
| Leitura da malha | 1,77 | 1,78 | 1,76 | 1,79 | 1,77 | 1,78 | 1,78 | 1,76 |
| Divisão da malha | 1,4e-5 | 1,55 | 1,66 | 1,75 | 1,78 | 1,90 | 2,00 | 2,23 |
| Construção da malha | 3,28 | 1,71 | 0,93 | 0,53 | 0,32 | 0,09 | 0,13 | 0,02 |
| Tempo total | 211,49 | 111,42 | 93,74 | 81,15 | 37,21 | 16,56 | 13,06 | 11,11 |

realizada qualquer divisão, seu tempo é praticamente zero. Quanto maior a quantidade de processadores, maior o tempo levado para divisão da malha, já que mais subdomínios devem ser gerados. A construção da malha refere-se a criar a malha dos subdomínios fazendo uso da `localGridData` gerada pela função `divide`. Conforme o número de processadores usados aumenta, o tamanho do `localGridData` diminui e, por consequência, o tempo levado também diminui. No geral, é possível notar que o tempo total para execução

Figura 15 – Distribuição da temperatura no caso 1




Fonte: Do Autor

da simulação foi diminuindo conforme mais processadores foram utilizados. Quando 16 processadores foram usados houve uma queda mais acentuada no tempo. Isso se deve ao fato que até 8 processadores foi usado apenas 1 nó computacional, enquanto que para 16 processadores foram usados 2 nós, logo a carga computacional em cada processador foi menor, já que cada processador trabalhou com um subdomínio menor. A Figura 16 e a Figura 17 ilustram os tempos mostrados na Tabela 1 em gráficos.

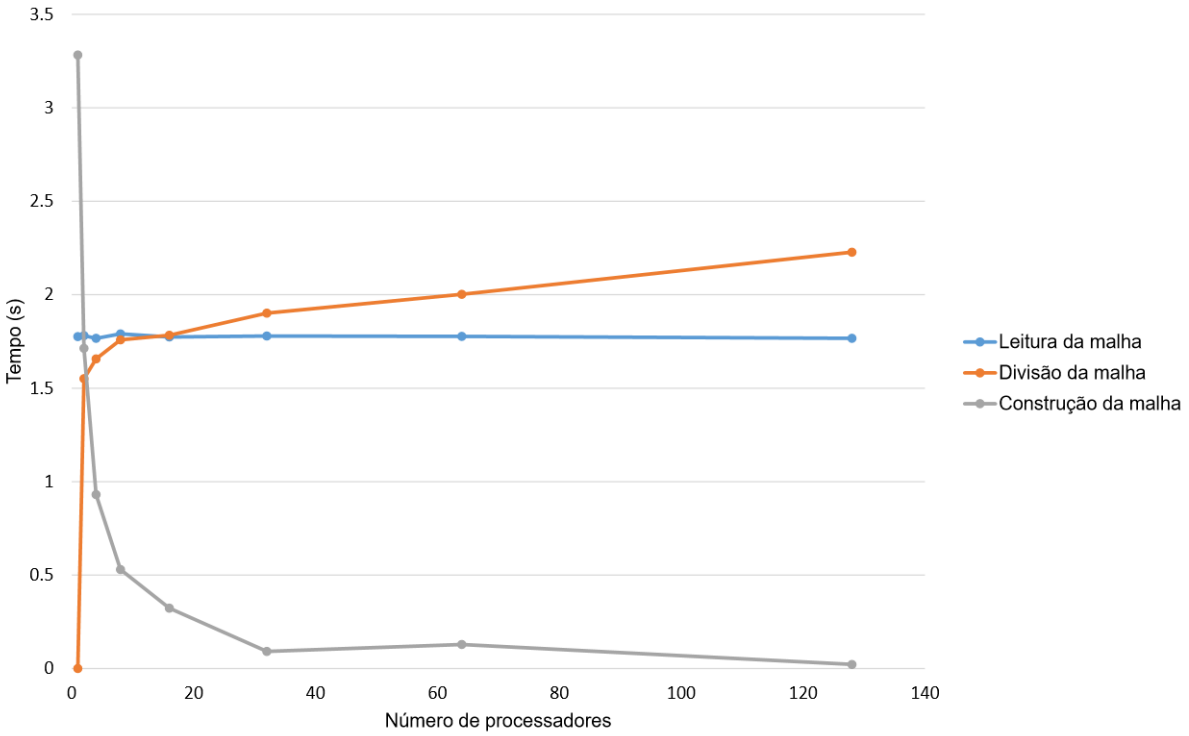
Na Figura 18 está representado o *speedup* calculado para o Caso 1. É possível perceber que seu comportamento não foi linear, como já era esperado. Com 2 processadores o *speedup* ainda se comporta linearmente, mas com 4 já ocorre um desvio que aumenta até 128 processadores. Com 32, 64 e 128 processadores ocorre uma diferença menor na mudança do *speedup*, demonstrando o início da região de saturação.

## 5.2 Caso 2

O segundo caso simulado também foi um problema de difusão de calor, mas com a diferença da malha ser tridimensional: ela conta com 1743958 nós e 7617709 de tetraedros. As condições de contorno são as mesmas do caso 1 e as suposições feitas também se aplicam. A Figura 19 mostra uma ilustração da malha.

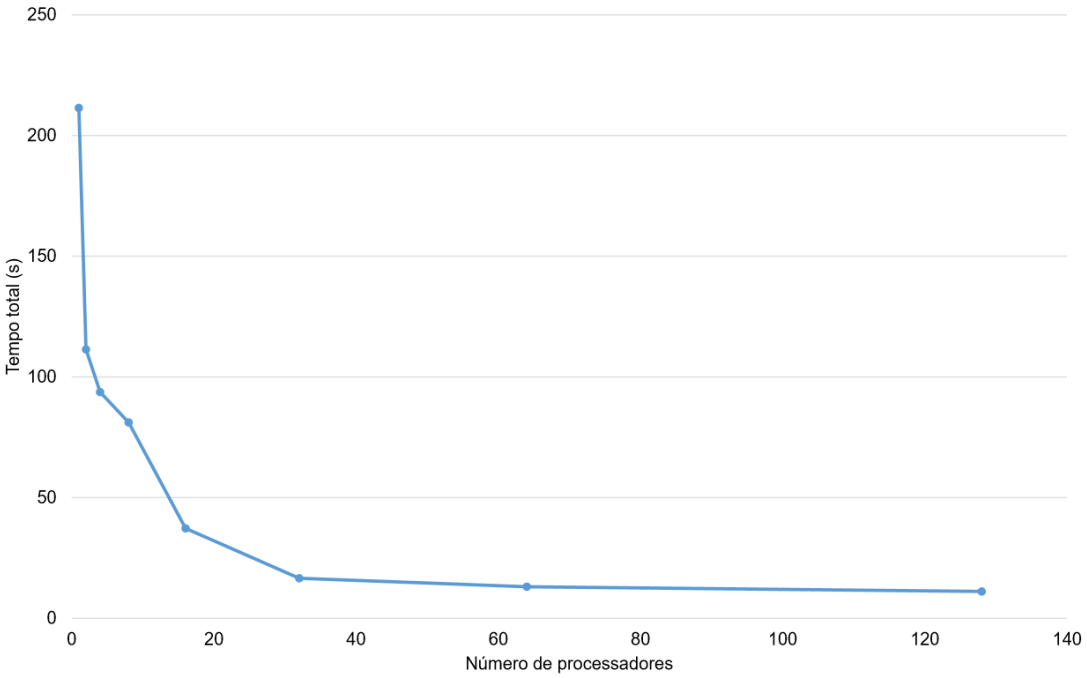
A Tabela 2 mostra o tempo que cada operação levou para ser realizada com a quantidade de processadores mostrada. Não foi possível gerar valores para 1, 2, 4 e 8 pro-

Figura 16 – Tempo computacional das operações na malha



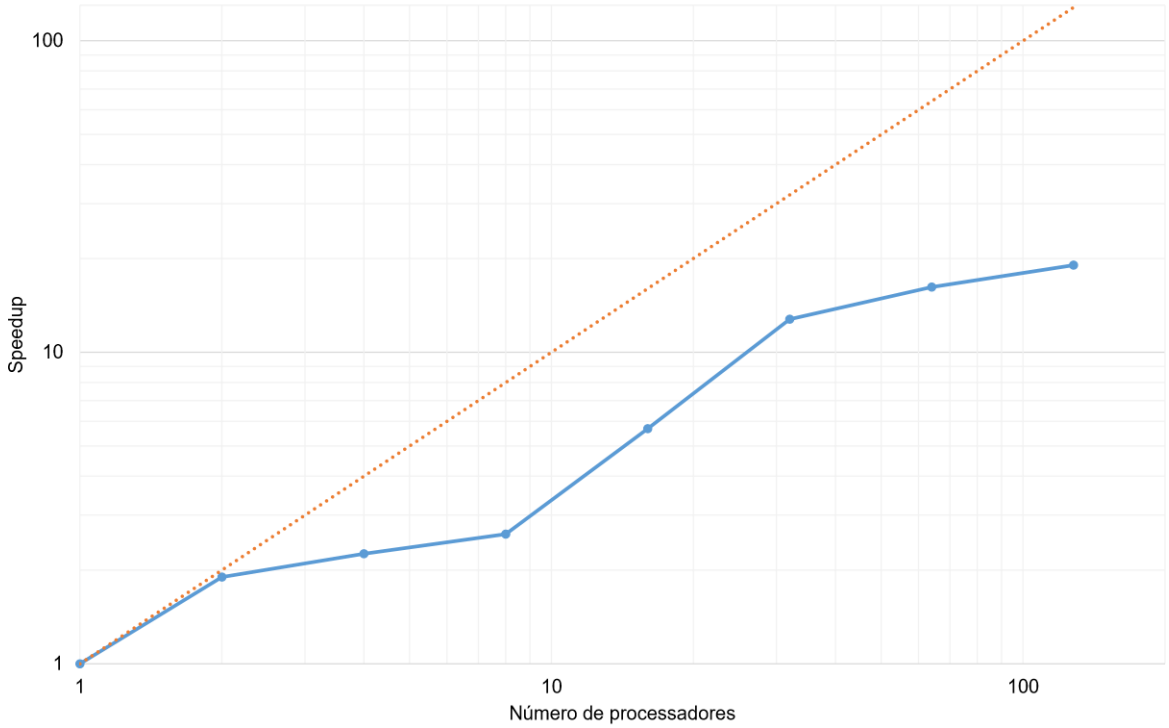

Fonte: Do Autor

Figura 17 – Tempo total de computação em função do número de processadores



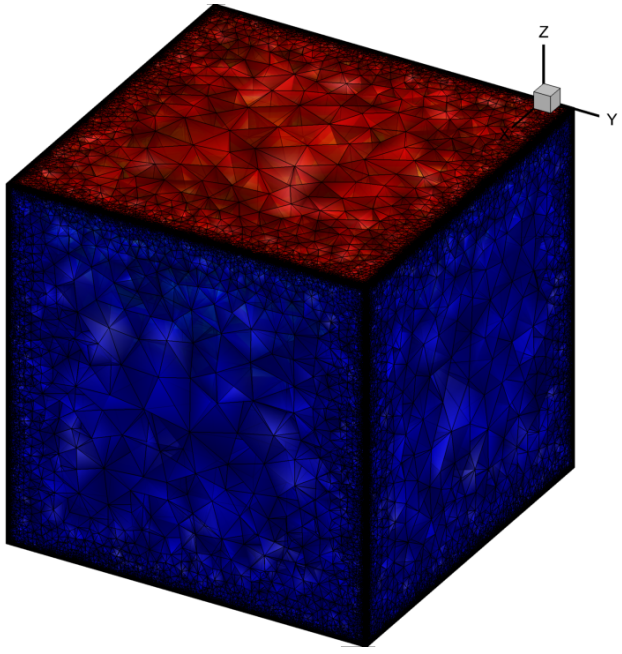

Fonte: Do Autor

Figura 18 – Distribuição da temperatura no caso 1




Fonte: Do Autor

Figura 19 – Ilustração da malha do caso 2



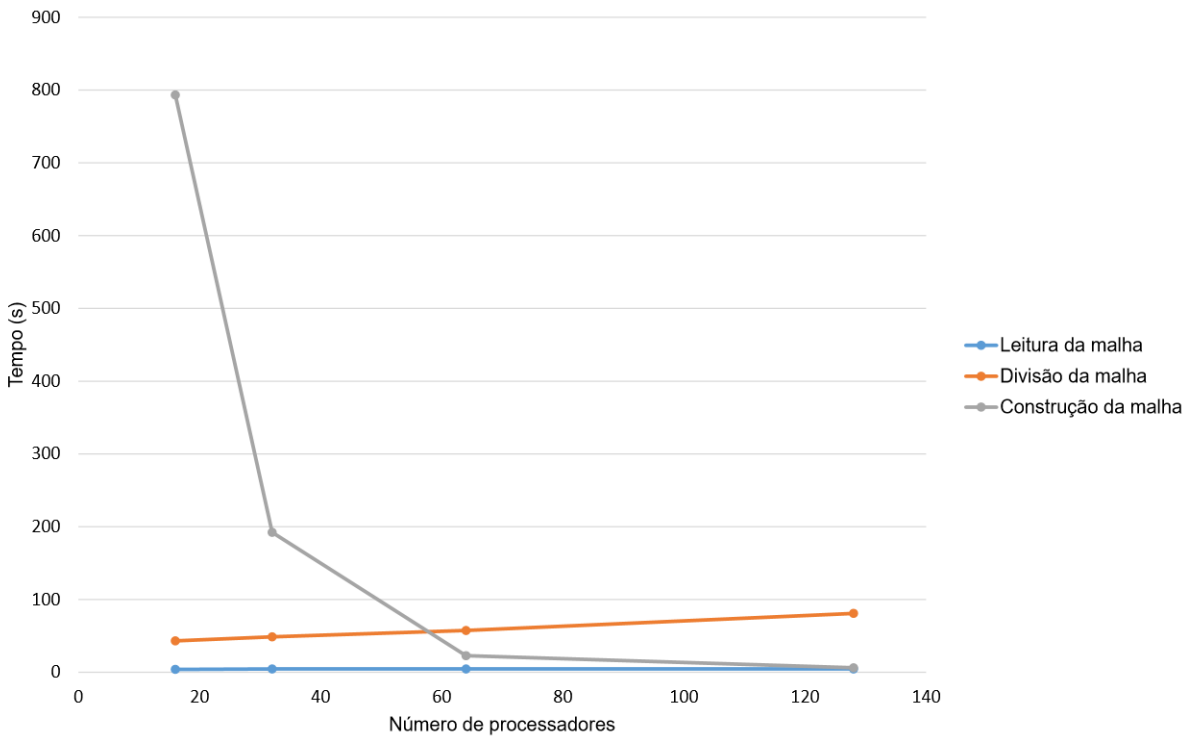

Fonte: Do Autor

cessadores, ou seja, usar apenas 1 nó computacional. A malha utilizada é tão refinida que apenas 1 nó não fornece a quantidade de RAM necessária para tal simulação. Quando se usam 16 ou mais processadores mais de 1 nó computacional é utilizado, logo foi possível gerar os resultados. Outra coisa que vale ser notada é a enorme diferença no tempo total da simulação. Utilizando 128 processadores o tempo foi quase 31 vezes menor que usar 16 processadores. A contrução da malha também foi grandemente reduzida com o aumento na quantidade de processadores, mostrando o grande impacto da programação paralela na performance de simulações de grande porte. Como não existem valores para 1 processador, não foi possível calcular o *speedup* da simulação.

Tabela 2 – Tempo de cada operação para diferentes números de processadores - caso 2

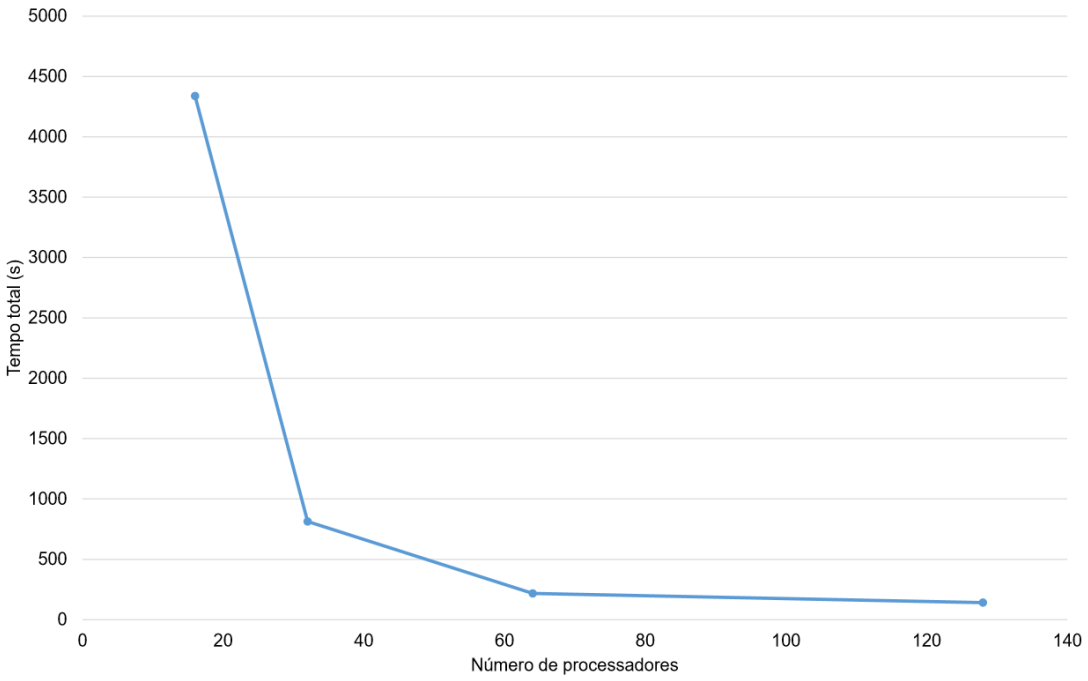
| **Operação** | **#16** | **#32** | **#64** | **#128** |
|---|---|---|---|---|
| Leitura da malha | 3,94 | 4,58 | 4,63 | 4,59 |
| Divisão da malha | 43,22 | 48,66 | 57,40 | 80,93 |
| Construção da malha | 793,32 | 192,39 | 22,73 | 6,29 |
| Tempo total | 4338,63 | 813,15 | 217,31 | 141,17 |

Figura 20 – Tempo computacional das operações na malha




Fonte: Do Autor

Figura 21 – Tempo total de computação em função do número de processadores




Fonte: Do Autor

# 6 Conclusão

## 6.1 Sumário

O objetivo do presente trabalho foi auxiliar o mestrando Ederson Grein na aplicação de métodos computação paralela na biblioteca de desenvolvimento EFVLib. A computação paralela consiste basicamente no uso de diversos processadores trabalhando ao mesmo tempo. Com a divisão do problema, cada processador trabalha apenas com uma parte do problema, reduzindo assim o tempo total de simulação. Idealmente o tempo de se usar $N$ processadores seria $N$ vezes menor que usar apenas 1. Mas devido a regiões no código que são executadas em serial, isso não acontece.

Para o uso da computação paralela é preciso uma combinação do hardware e do código computacional. O tipo de arquitetura do computador influência como a memória está disponível para os processadores e tem influências também na quantidade máxima de processadores que podem ser usados. O algoritmo usado deve especificar quais operações cada processador vai executar. Rotinas MPI foram usadas para garantir uma comunicação eficiente entre os processadores.

O kit de ferramentas para computação paralela foi desenvolvido para uso na biblioteca EFVLib. Nessa biblioteca é possível fazer uso de malhas 2 e 3D não estruturadas híbridas, que empregam o EbFVM. Foi empregada uma metodologia de decomposição do domínio, onde a malha foi dividida em subdomínios e cada um desses subdomínios foi alocado a um processador. Uma divisão igualitária se faz necessária para evitar que um processador fique sobrecarregado e ele dite a performance do sistema. A representação por meio de grafos é útil para que a divisão a ser feita corte a menor quantidade de arestas possível, assim diminuindo a posterior quantidade de comunicação entre os processadores. Os nós que precisam ser adicionados de outros subdomínios para completar os elementos partidos são chamados de nós fantasma e nenhuma computação é realizada neles, eles apenas armazenam valores. Os sistemas lineares resultados são resolvidos pela PETSc.

## 6.2 Conclusões

Os dois casos que foram rodados usando o novo kit de ferramentas para computação paralela foram problemas de difusão pura de calor em malhas não estruturadas híbridas. No primeiro caso foi usada uma malha bidimensional com elementos triangulares. Foi possível notar um aumento no tempo de divisão da malha, já que mais subdomínios precisam ser gerados para mais processadores e uma diminuição no tempo de construção, já que os

subdomínios ficaram menores. O tempo total de simulação também diminuiu, mostrando que o trabalho feito atingiu seu objetivo. O *speedup* não se comportou de maneira linear e foi possível perceber o início da região de saturação. No segundo caso foi usada uma malha tridimensional e a evolução do tempo total de simulação com mais processadores foi maior. Como a malha usada era muito refinada, apenas 1 nó computacional do cluster não teve memória suficiente para carregar ela, logo apenas simulações com mais de 16 processadores geraram resultados.

# 7 Referências

**METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices**. Department of Computer Science & Engineering, University of Minnesota.

BALAY, S., BROWN, J., BUSCHELMAN, K., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., ZHANG, H. **PETSc (Portable, Extensible Toolkit for Scientific Computation).** http://www.mcs.anl.gov/petsc, 2013.

**BOOST C++ LIBRARIES**. boost.org, January, 2015.

DONGARRA, J. J. **Sourcebook of Parallel Computing**. Morgan Kaufmann Publishers, 2003.

DOROH, M. G. **Development and Application of a Parallel Compositional Reservoir Simulator**. Master's Thesis, The University of Texas at Austin, 2012.

GEBALI, F. **Algorithms and Parallel Computing**. Wiley, 2011.

HURTADO, F. S. V. **Formulação tridimensional de volumes finitos para simulação de reservatórios de petróleo com malhas não- estruturadas híbridas**. Tese de doutorado, Departamento de Engenharia Mecânica, Universidade Federal de Santa Catarina, Florianópolis, Brasil, 2011.

KARYPIS, G., KUMAR, V. **Multilevel Algorithms for Multi-Constraint Graph Partitioning**. Technical report, University of Minnesota, Department of Computer Science / Army HPC Research Center, 1998.

KARYPIS, G., KUMAR, V. **Multilevel k-way Partitioning Scheme for Irregular Graphs**. Journal of Parallel and Distributed Computing, v. 48, pp. 96 – 129, 1998.

MALISKA, C. R. **Transferência de Calor e Mecânica dos Fluidos Computacional**. LTC Editora, Rio de Janeiro, RJ, 2004.

MALISKA, C. R., DA SILVA, A. F. C., HURTADO, F. S. V., DONATTI, C. N., AMBRUS, J. **Relatório Técnico Número 2 Parte 2, Rede Temática de Gerenciamento e Simulação de Reservatórios (SIGER)**. Technical report, Universidade Federal de Santa Catarina, 2008.

MALISKA, C. R., SILVA, A. F. C., HURTADO, F. S. V., DONATTI, C. N., PESCADOR JR., A. V. B., RIBEIRO, G. G. **Manual de classes da biblioteca EFVLib**. Relatório técnico SINMEC/SIGER I-06, Parte 1, Departamento de Engenharia Mecânica, Universidade Federal de Santa Catarina, 2011.

PADUA, D., editor. **Encyclopedia of Parallel Computing**. Springer, 2011.

SHANKLAND, S., October, 2012. Keeping moore's law ticking.

TROBEC, R., VAJTERSIC, M., ZINTERHOF, P. **Parallel Computing : Numerics, Applications, and Trends**. Springer, 2009.

ZIENKIEWICZ, O. C., TAYLOR, R. L. **The Finite Element Method**, fifth, vol. 1. Butterworth-Heinemann, 2000.