Edmilson Charles Nogueira

ESTUDO DA PARALELIZAÇÃO PARA SOLUÇÃO DO SISTEMA LINEAR DO MÉTODO DOS ELEMENTOS FINITOS GENERALIZADOS

Dissertação apresentada ao Curso de Mestrado em Modelagem Matemática e Computacional do Centro Federal de Educação Tecnológica de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Modelagem Matemática e Computacional.

Orientador: Prof. Dr. Yukio Shigaki

BELO HORIZONTE

2011

AGRADECIMENTO

Em primeiro lugar agradeço a Deus por ter me dado saúde e força para transpor todos os desafios que surgiram nesse período.

Ao Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG) e à FAPEMIG pelo apoio financeiro.

Ao professor Yukio pela orientação, pela paciência, pelos ensinamentos e pela amizade construída nesse período. Agradeço por ter sido nesse período muito mais que um orientador, me ajudando muitas vezes até mesmo em assuntos não relacionados com o mestrado.

Ao professor Felício (Departamento de Engenharia de Estruturas – UFMG) pela paciência, pelo apoio dado nesse período e também por ter fornecido os dados necessários para conclusão desse projeto.

A minha esposa e ao meu filho por terem sido meu "Porto Seguro", por terem sido minha motivação quando tudo parecia dar errado.

Aos meus pais e irmãos por sempre terem acreditado na minha capacidade.

Aos meus colegas de jornada por compartilhamos juntos conhecimentos e as dificuldades encontradas.

RESUMO

O Método dos Elementos Finitos Generalizados (MEFG) pode ser entendido como uma abordagem não convencional do Método dos Elementos Finitos, compartilhando características com os métodos sem malha. A matriz gerada por esse método é do tipo semidefinida positiva, que possui infinitas soluções, mantendo mesmo assim a unicidade da solução de Galerkin. Em trabalho publicado por Strouboulis *et al*. foi proposto um procedimento que provoca uma perturbação na matriz que se torna, então, definida positiva. As etapas desse procedimento exigem a solução de sistemas para cálculo das aproximações do resultado e do cálculo do erro do procedimento, sendo nesse trabalho utilizado o Método do Gradiente Conjugado Pré-condicionado (MGCP), devido a seu desempenho em matrizes esparsas como as geradas por esse tipo de problema. Como o procedimento apresenta um custo computacional muito elevado, torna-se interessante pesquisar a utilização da computação paralela para solucionar esse tipo de problema. A escolha, porém, do pré-condicionador para esse tipo de sistema paralelizado ainda não é clara. Este trabalho pretende explorar essas questões, estudando os métodos de solução de sistemas lineares, os pré-condicionadores específicos, e sua implementação em um *cluster Beowulf* para solução de sistemas gerados em problemas com o MEFG.

Palavras-chave: Programação Paralela; Solução de Sistemas Lineares; Método dos Elementos Finitos Generalizados.

ABSTRACT

The Generalized Finite Element Method (GFEM) can be understood as an unconventional approach of the Finite Element Method, sharing characteristics with the meshless methods. The matrix generated by this method is of type positive semidefinite, which has infinite solutions, still keeping the uniqueness of the solution of Galerkin. In a study published by Strouboulis *et al*. proposed a procedure that causes a disturbance in the matrix becomes then positive definite. The steps of this procedure requires the solution of systems approaches to calculate the result and the calculation of error procedure, and this work used the Preconditioned Conjugate Gradient Method (PCGM), due to his performance as the sparse matrices generated by this kind of problem. As the procedure has a very high computational cost, it is interesting to research the use of parallel computing to solve this problem. The choice, however, the pre-conditioner for this type of system is not yet clear parallelized. This paper aims to explore these questions by studying the methods of solution of linear systems, the specific pre-conditioners, and its implementation on a Beowulf cluster systems to solve problems with the generated GFEM.

Keywords: Parallel Programming; Solution of Linear Systems; Generalized Finite Element Method.

SUMÁRIO

1 INTRODUÇÃO 12

1.1 Objetivo Geral 14

1.2 Objetivos Específicos 14

1.3 Justificativa 15

1.4 Estrutura do Trabalho 15

2 MÉTODO DOS ELEMENTOS FINITOS GENERALIZADO (MEFG) 16

2.1 Síntese do MEFG 16

2.2 Procedimento Babuska 17

3 MÉTODOS PARA SOLUÇÃO DE SISTEMAS LINEARES 20

3.1 Métodos Diretos 20

3.1.1 Desvantagens dos Métodos Diretos 24

3.2 Métodos Iterativos 24

3.2.1 Método do Gradiente Conjugado (MGC) 26

3.2.2 Método do Gradiente Conjugado Pré-condicionado (MGCP) 28

3.3 Pré-condicionadores 29

3.3.1 Pré-condicionador de Jacobi 30

3.3.2 Pré-condicionador da Sobre-Relaxação Simétrica (SSOR) 30

3.3.3 Pré-condicionador de Fatoração Incompleta 31

4 PROCESSAMENTO PARALELO 33

4.1 Memória Compartilhada 33

4.2 Memória Distribuída 34

4.3 Memória Híbrida 34

4.4 Modelos de Implementação Paralela 35

4.4.1 OpenMP 35

4.4.1 Message Passing Interface (MPI) 36

4.4 Cluster Beowulf 38

4.5 Análise de Desempenho 38

5 CÓDIGO DESENVOLVIDO 40

5.1 Esquemas de Armazenamento 41

5.2 Implementação Utilizando o OpenMP 44

5.3 Implementação Utilizando o MPI 48

5.4 Implementação Utilizando a Abordagem Híbrida 56

6 RESULTADOS NUMÉRICOS 57

6.1 Ambiente de Teste 57

6.2 Modelo Analisado 58

6.3 Resultados Utilizando Somente o OpenMP 59

6.4 Resultados Utilizando Somente o MPI 61

6.5 Resultados Utilizando a Abordagem Híbrida (OpenMP +MPI) 64

7 CONCLUSÕES 66

8 TRABALHOS FUTUROS 68

REFERÊNCIAS BIBLIOGRÁFICAS 70

APÊNDICE A – DEFINIÇÕES 73

APÊNDICE B – IMPLEMENTAÇÃO DE UM CLUSTER BEOWULF 74

APÊNDICE C – TUTORIAL OPENMP 82

APÊNDICE D – TUTORIAL MPI 86

LISTA DE ALGORITMOS

Algoritmo 1 – Procedimento Babuska 19
Algoritmo 2 – Substituição Regressiva 21
Algoritmo 3 – Substituição Sucessiva 21
Algoritmo 4 – Eliminação de Gauss sem Pivotamento 22
Algoritmo 5 – Gradiente Conjugado 28
Algoritmo 6 – Gradiente Conjugado Pré-condicionado 29
Algoritmo 7 – Produto Interno sequencial 35
Algoritmo 8 – Produto Interno utilizando o OpenMP 36
Algoritmo 9 – Produto Interno utilizando o MPI 37
Algoritmo 10 – Produto Interno utilizando a abordagem híbrida (MPI + OpenMP) 37
Algoritmo 11 – Armazenamento CRS 42
Algoritmo 12 – Multiplicação Matriz-Vetor no Formato CRS 43
Algoritmo 13 – Substituição Sucessiva no formato CRS 43
Algoritmo 14 – Substituição Regressiva no formato CRS 44
Algoritmo 15 – Pré-condicionador da SSOR no formato CRS 44
Algoritmo 16 – Multiplicação Matriz-Vetor utilizando a abordagem OpenMP 45
Algoritmo 17 – Pré-condicionador de Jacobi em sua versão paralela utilizando OpenMP 45
Algoritmo 18 – Pré-condicionador da SSOR em sua versão paralela utilizando OpenMP 46
Algoritmo 19 – Gradiente Conjugado utilizando a abordagem OpenMP 47
Algoritmo 20 – Rotina de Pré-condicionamento do MGCP 47
Algoritmo 21 – Procedimento Babuska utilizando o OpenMP 48
Algoritmo 22 – Balanceamento de carga utilizado 50
Algoritmo 23 – Multiplicação matriz-vetor utilizando a abordagem MPI 50
Algoritmo 24 – Pré-condicionador de Jacobi utilizando a abordagem MPI 51
Algoritmo 25 – Pré-condicionador SSOR utilizando a abordagem MPI 52
Algoritmo 26 – Pré-condicionador SSOR utilizando a abordagem MPI: Segmento 1 52
Algoritmo 27 – Pré-condicionador SSOR utilizando a abordagem MPI: Segmento 2 52
Algoritmo 28 – Pré-condicionador SSOR utilizando a abordagem MPI: Segmento 3 53
Algoritmo 29 – Substituição sucessiva utilizando MPI 53
Algoritmo 30 – Substituição sucessiva utilizando MPI: Segmento 1 54
Algoritmo 31 – Substituição sucessiva utilizando MPI: Segmento 2 54
Algoritmo 32 – Substituição sucessiva utilizando MPI: Segmento 3 54
Algoritmo 33 – Gradiente Conjugado utilizando MPI 55
Algoritmo 34 – Procedimento Babuska utilizando MPI 55
Algoritmo 35 - Multiplicação matriz-vetor utilizando a abordagem Híbrida 56

LISTA DE FIGURAS

Figura 1 – Placa modelada 58
Figura 2 – Speed-up dos casos testados utilizando OpenMP somente 60
Figura 3 – Eficiência paralela dos métodos testados utilizando o OpenMP somente 60
Figura 4 – Tempo de execução dos métodos estudados utilizando o OpenMP somente 60
Figura 5 – Speed-up dos métodos implementados utilizando a abordagem MPI somente..... 63
Figura 6 – Eficiência paralela dos métodos testados utilizando o MPI somente 63
Figura 7 – Tempo de execução dos métodos estudados utilizando o MPI somente..... 63
Figura 8 – Speed-up dos métodos implementados utilizando a abordagem Híbrida..... 65
Figura 9 – Eficiência paralela dos métodos testados utilizando a abordagem Híbrida 65
Figura 10 – Tempo de execução dos métodos estudados utilizando a abordagem Híbrida 65

LISTA DE TABELAS

TABELA 1 – Variáveis utilizadas nesse trabalho (implementação do MPI) 48
TABELA 2 – Características das máquinas que compõem o cluster implementado 57
TABELA 3 – Características da placa modelada 58
TABELA 4 – Tempo de execução (em segundos) utilizando OpenMP 59
TABELA 5 – Tempo de execução (em segundos) utilizando o armazenamento CRS e a abordagem MPI 61
TABELA 6 – Tempo de execução (em segundos) do Pré-condicionador de Jacobi utilizando armazenamento denso com o MPI 61
TABELA 7 – Tempo de execução(em segundos) utilizando a abordagem híbrida 64

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CCS	*Compressed Column Storage*
CPU	*Central Processing Unit*
CRS	*Compressed Row Storage*
Flops	*Floating Point Operations per Second*
IP	*Internet Protocol*
MEF	Método dos Elementos Finitos
MEFG	Método dos Elementos Finitos Generalizados
MGC	Método do Gradiente Conjugado
MGCP	Método do Gradiente Conjugado Pré-condicionado
MIMD	*Multiple Instruction, Multiple Data*
MPI	*Message Passing Interface*
OpenMP	*Open Multi Processing*
PU	Partição de Unidade
PUFEM	*Partition of Unit Finite Element Method*
PVC	Problema de Valor de Contorno
SMP	*Symmetric Multiprocessor*
SSOR	*Symmetric Successive Over-relaxation*

LISTA DE SÍMBOLOS

$\alpha, \ldots, \omega$	escalares
κ	número de condicionamento de uma matriz
λ_n	maior autovalor de uma matriz
λ_1	menor autovalor de uma matriz
φ_i(x)	Funções da Partição da Unidade
$\Psi(x)$	Funções de enriquecimento
ω	Fator de sobre-relaxação
$A, \ldots, Z$	Matrizes
$\|A\|_2$	Norma l_2 da matriz A
A^{-1}	Matriz inversa
A^T	Matriz transposta
C	Matriz Pré-condicionadora
D	Matriz Diagonal
L	Matriz Triangular Inferior
U	Matriz Triangular Superior
I	Matriz Identidade
$a, \ldots, z$	Vetores
$u^h(x)$	Aproximação da solução
$u^{FEM}(x)$	Aproximação padrão do MEF
$u^{enr}(x)$	Enriquecimento da PU
$N_i(x)$	Função de forma padrão do MEF
δij	Delta de Kronecker
T_1	Tempo de processamento em uma máquina
T_n	Tempo de processamento em n máquinas
T_c	Tempo de comunicação

$diag()$	diagonal de uma matriz
$< x, y >$	Produto interno de x e y

1 INTRODUÇÃO

Devido ao elevado poder de processamento dos computadores atuais, bem como a quantidade de núcleos de processamento disponíveis por máquinas, e do custo de aquisição bem inferior ao antes praticado, a programação paralela vem se tornando uma alternativa atrativa para o desenvolvimento de aplicativos mais eficientes e voltados, sobretudo para a resolução de problemas de elevada escala.

Como o aumento do poder computacional, problemas cada vez maiores e mais complexos vêm sendo estudados e resolvidos. Alguns problemas, no entanto possuem dimensão e complexidade tão elevadas que com a utilização da computação tradicional (sequencial), o tempo necessário para obter a solução do problema pode ser tão grande que o torna impraticável.

Dentre as arquiteturas paralelas disponíveis, o *cluster Beowulf* é uma das mais utilizadas (STERLING, 2001), primeiramente por seu baixo custo e também por sua facilidade de implementação e manutenção, podendo ser composto tanto por máquinas mono processadas quanto máquinas multiprocessadas. Muitos dos sistemas utilizados no meio acadêmico e industrial são *clusters* e, no final da década de 90, havia 28 *clusters* na lista dos *TOP500* (lista dos 500 computadores mais rápidos do mundo), e isso sete anos após o primeiro ter sido inserido.

Desde o surgimento da computação paralela, muitos paradigmas de programação foram desenvolvidos e utilizados, sendo que alguns foram definidos como padrão pela comunidade de computação paralela. No mesmo ano do surgimento do primeiro *cluster Beowulf* (1994), o *Message Passing Interface* (MPI) foi definido como padrão de desenvolvimento para máquinas de memória distribuída (multicomputadores). Em 1997 foi criado o *Open Multi Processing* (*OpenMP*), que é uma interface baseada em *threads* (fluxos de processamento) utilizada para prover paralelismo incremental em máquinas com memória compartilhada (multiprocessadores) que tem sido bastante utilizada sobretudo por sua simplicidade (CHAPMAN *et al.*, 2008). Também é possível a utilização das duas interfaces em conjunto e assim desenvolver softwares para as máquinas híbridas (memória compartilhada distribuída), comum em *clusters* (agrupamento de computadores) compostos por máquinas multiprocessadas.

Nesse trabalho todo o código desenvolvido é utilizado na resolução dos sistemas lineares gerados pelo Método dos Elementos Finitos Generalizados (MEFG), que pode ser

entendido como uma abordagem não tradicional do Método dos Elementos Finitos (MEF), compartilhando diversas características com os métodos sem malha como o Método de Galerkin Livre de Malha (BELYTSCHKO *et al.*, 1994) e o Método das Nuvens hp (DUARTE; ODEN, 1996).

Devido a sua versatilidade o MEFG vem sendo empregado em problemas caracterizados por descontinuidades, singularidades, deformações localizadas e geometrias complexas, simplificando a solução, por exemplo, de problemas que envolvem propagação de trinca (BELYTSCHKO *et al.*, 2009). Algumas aplicações do MEFG podem ser encontradas em Kim *et al.* (2009), Khoei *et al.* (2007), Sukumar *et al.* (2000).

Uma característica importante do MEFG é que ele gera uma matriz semidefinida positiva, cuja solução é obtida conforme algoritmo iterativo sugerido em STROUBOULIS *et al.* (2000) sendo neste texto tratado como "Procedimento Babuska". Nesse procedimento, uma pequena perturbação é aplicada à diagonal principal da matriz de modo que ela se torne definida positiva, utilizado então em suas iterações, um método para solução de sistemas até que a solução do problema seja obtida, sendo nesse trabalho utilizado o Método do Gradiente Conjugado (MGC).

Vários motivos conduzem à utilização do MGC, dentre eles a boa convergência do método em sistemas com matrizes esparsas (comuns em problemas de elementos finitos), e também devido ao fato do método ser facilmente paralelizado.

Os métodos diretos também podem ser utilizados, porém o desempenho desses métodos é influenciado negativamente pelo aumento da dimensão do problema e, em problemas de dimensão muito grande sua computação é muito cara e muitas vezes impraticável. Esses métodos são marcados também por seu pobre paralelismo (sobretudo pela dependência de dados comum nesses processos), principalmente em sistemas com memória distribuída.

Como o número de condicionamento de uma matriz aumenta com a elevação do número de variáveis e, os métodos do subespaço de Krylov (como o MGC) são dependentes do número de condicionamento, a técnica de pré-condicionamento é extremamente necessária e, frequentemente, o Método do Gradiente Conjugado Pré-condicionado (MGCP) é utilizado ao invés do MGC. Porém com isso surge um novo problema: a escolha do pré-condicionador, pois além das características dos pré-condicionadores, é necessário também analisar o seu desempenho no ambiente paralelo. Os pré-condicionadores gerados pelo processo de fatoração incompleta, por exemplo, são extremamente eficientes, porém apresentam um fraco

paralelismo (BARRETT *et al.*,1994) devido à dependência de dados comum nesses processos e são, portanto, evitados neste trabalho.

1.1 Objetivo Geral

O objetivo deste trabalho é o estudo e desenvolvimento de uma opção paralelizada de um programa computacional e de seus pré-condicionadores, capaz de resolver sistemas lineares de ordem elevada gerados a partir do modelo utilizando o Método dos Elementos Finitos Generalizados.

1.2 Objetivos Específicos

Para atingir o objetivo proposto, pretende-se:

- Apresentar as técnicas de solução de sistemas lineares, concretamente o Método do Gradiente Conjugado Pré-condicionado (MGCP) e alguns dos seus Pré-condicionadores.

  Para a solução de sistemas lineares existem, basicamente, métodos diretos e iterativos. Deseja-se analisar a eficácia do MGCP quando aplicados à solução de sistemas de elevadas dimensões gerados pelo MEFG através de algoritmos paralelizados.

- Estudar e implementar computacionalmente no ambiente paralelo o método iterativo, denominado “Procedimento Babuska”, utilizando-o para obter a solução de sistemas lineares do MEFG.

  Utilizando a linguagem Fortran 95, deseja-se desenvolver um código paralelizado utilizando as abordagens MPI, OpenMP e Híbrida (MPI + OpenMP), utilizando tal código para resolver sistemas gerados pelo MEFG.

- Obter parâmetros de desempenho para o sistema implementado.

  Para verificar a eficiência do código implementado, deseja-se determinar alguns parâmetros de desempenho em arquiteturas paralelas.

1.3 Justificativa

Embora a computação paralela seja cada vez mais aplicada, o mercado ainda é carente de softwares paralelizados. A necessidade de resolução de sistemas cada vez maiores e mais complexos reforça a necessidade de estudo sobre esse novo paradigma.

1.4 Estrutura do Trabalho

No capítulo 2, apresenta-se uma breve síntese do MEFG, destacando-se a estrutura matricial peculiar do método e exibindo o procedimento iterativo proposto para a obtenção da solução do sistema gerado.

No capítulo 3 são apresentados alguns métodos utilizados para resolução de sistemas lineares. Um breve estudo sobre os métodos diretos é realizado apresentando os principais problemas que eles apresentam quando a matriz do sistema é esparsa, de elevada dimensão e também suas deficiências no processamento paralelo. É feito então um estudo sobre o MGC e de sua versão pré-condicionada e apresentados os pré-condicionadores mais utilizados.

No capítulo 4 são exibidas algumas características da computação paralela, principalmente os dois modelos de programação utilizados (MPI e OpenMP).

No capítulo 5 são exibidas e comentadas as implementações desenvolvida nesse trabalho utilizando as abordagens MPI, OpenMP e Híbrida, sendo os algoritmos apresentados desenvolvidos para matrizes armazenadas no formato CRS (*Compressed Row Storage*)

No capítulo 6 são exibidos os resultados utilizando as abordagens MPI, OpenMP e Híbrida, sendo esses resultados comentados conforme os parâmetros utilizados.

Finalmente (Capítulo 7) as principais conclusões referentes ao modelo analisado são apresentadas e também propostas algumas melhorias futuras.

2 MÉTODO DOS ELEMENTOS FINITOS GENERALIZADO (MEFG)

2.1 Síntese do MEFG

O Método dos Elementos Finitos Generalizado (MEFG), segundo Duarte *et al.* (2000) foi proposto independentemente por Babuska e colegas, com o nome *Partition of Unity Finite Element Method* , PUFEM, e por Duarte e Oden com o nome *hp-Clouds.*

Assim como o Método das Nuvens *hp*, o MEFG constrói seu espaço de aproximação por produto da Partição de Unidade (PU), por funções de enriquecimento que apresentam boas propriedades de aproximação (MENDONÇA; FANCELLO, 2002). A principal característica dos Métodos da Partição da Unidade é sua habilidade de utilizar conhecimento *a priori* sobre a solução de um problema na forma das suas funções de enriquecimento (KIM *et al*, 2009).

O conceito da Partição da Unidade (ODEN; REDDY, 1976) é a chave da construção das funções de forma do MEFG. A Partição de Unidade (BELYTSCHKO *et al.*, 2009) em um domínio Ω, é um conjunto de funções cujo somatório corresponde a 1, ou seja

$$\sum_{i=1}^{N} \varphi_i(x) = 1, \forall x \in \Omega \tag{01}$$

onde *N* corresponde ao conjunto de pontos nodais.

A partir da função (Equação 01) obtém-se (Equação 02), uma propriedade base para a construção das funções de forma de vários métodos.

$$\sum_{i=1}^{N} \varphi_i(x)\Psi(x) = \Psi(x) \tag{02}$$

A principal distinção entre o MEFG e os métodos sem malha que utilizam a Partição da Unidade, é que a PU do MEFG pode ser construída utilizando as mesmas funções de forma do MEF convencional, já que as funções langrangeanas também podem ser classificadas como PU, pois elas satisfazem a Equação 01. Portanto a implementação do MEFG é a mesma do MEF tradicional, diferenciando-se apenas na definição das funções de forma. Isso facilita a estratégia de integração em comparação com os métodos sem malha (Duarte *et al*, 2000).

Matematicamente a aproximação do MEFG pode ser dada com

$$u^h(x) = u^{FEM}(x) + u^{enr}(x) \tag{03}$$

onde u^h é a aproximação da solução, u^{FEM} é a aproximação padrão do MEF (Equação 04) e u^{enr} é o enriquecimento da PU (Equação 05).

$$u^{FEM}(x) = \sum_{\forall i} N_i(x)u_i \quad (04)$$

$$u^{enr}(x) = \sum_{\forall i} \varphi_i(x)\Psi(x)q_i \quad (05)$$

onde N_i corresponde às funções de forma padrão do MEF, u_i são os graus de liberdade nodal padrão, φ_i são funções da Partição da Unidade, $\Psi(x)$ é a função de enriquecimento que apresenta boa aproximação para o problema estudado, e q_i são parâmetros introduzidos (graus de liberdade adicionais). Como na abordagem do MEFG geralmente utiliza as funções lagrangeanas como PU, a Equação 05 pode ser reescrita como

$$u^{enr}(x) = \sum_{\forall i} N_i(x)\Psi(x)q_i \quad (06)$$

A escolha da PU afeta o sistema linear, visto que dependendo da escolha das funções da PU, as funções de forma do MEFG podem ser linearmente dependente ou linearmente independente o que faz do sistema singular ou não. O número de condicionamento da matriz também depende da escolha das funções da PU. Portanto a escolha das funções da PU também afeta na escolha do solver, visto que a escolha do solver depende do sistema linear a ser resolvido (Babuska *et al.*, 2004).

Um dos problemas do MEFG é o surgimento de uma matriz de rigidez singular mesmo após a aplicação das devidas condições de contorno. Isso ocorre quando as Partições da Unidade e as funções de enriquecimentos são polinomiais gerando um problema de dependência linear (MENDONÇA; FANCELLO, 2002). O sistema de equações resultante para a solução do problema numérico admite, portanto, infinitas soluções para os parâmetros associados às funções enriquecedoras; ainda assim a unicidade da solução de Galerkin é preservada.

Para resolver tal sistema emprega-se, neste trabalho, a estratégia iterativa sugerida em (STROUBOULIS *et al.*, 2000), denominada nesse trabalho como “Procedimento Babuska”.

2.2 Procedimento Babuska

Seja o seguinte sistema de equações, obtido da aproximação de Galerkin de um problema de valor de contorno (PVC) para a formulação do MEFG

$$Ax = b \quad (07)$$

onde $\boldsymbol{A}$ é uma matriz semidefinida positiva. Devido à dependência linear das equações do sistema linear formado, a matriz $\boldsymbol{A^{-1}}$ não pode ser definida.

Duarte *et al.* (2000) sugere que a matriz seja normalizada, ou seja, seja transformada em uma matriz onde todas as entradas da diagonal principal são iguais a 1, visando a redução de erros durante o processo iterativo. Da seguinte forma

$$T(i,j) = \frac{\delta ij}{K(i,j)} \tag{08}$$

$$A = TKT \tag{09}$$

$$b = Tf \tag{10}$$

onde

$$\delta ij = \begin{cases} 1, se\ i = j \\ 0, se\ i \neq j \end{cases} \tag{11}$$

é conhecida como Delta de Kronecker.

A estratégia proposta em (STROUBOULIS *et al.*, 2000) consiste em produzir uma pequena perturbação (ε) na matriz ***K*** de maneira que ela se torne definida positiva, dada por

$$K_e = K + \varepsilon I \tag{12}$$

onde $\varepsilon > 0$ e I é a matriz identidade.

Em (STROUBOULIS *et al.*, 2000) adota-se a perturbação no valor de 10^{-10} o que faz, segundo os autores, com que em uma única iteração e trabalhando-se em precisão dupla, já se tenha a convergência do método de solução.

Gera-se então uma primeira aproximação (iteração 0) para a solução do sistema

$$u^0 = {K_e}^{-1} f \tag{13}$$

Como u^0 é aproximado, haverá um resíduo definido por

$$r^0 = f - Ku^0 \tag{14}$$

O erro produzido no procedimento nessa etapa é dado por

$$e^0 = {K_e}^{-1} r^0 \tag{15}$$

A solução aproximada é então atualizada através de

$$u^1 = u^0 + e^0 \tag{16}$$

Cujo resíduo é

$$r^1 = f - Ku^1 = f - K(u^0 + e^0) \tag{17}$$

Que pode ser reformulado para

$$r^1 = r^0 - Ke^0 \tag{18}$$

As atualizações de r^i, e^i, e u^i devem ser repetidas até que uma determinada medida de erro seja pequena o suficiente e, quando isso ocorrer, obter o valor de x (Equação 19).

$$u = Tx \tag{19}$$

As iterações são realizadas até a razão $\left|\frac{eKe}{uKu}\right|$ seja suficientemente pequena, o que na prática ocorre com uma ou duas iterações (Babuska *et al*, 2004).O algoritmo completo encontra-se no Algoritmo 1.

$K_\varepsilon = K + \varepsilon I$
$u^{(0)} = {K_\varepsilon}^{-1} f$
$r^{(0)} = f - Ku^{(0)}$
$e^{(0)} = {K_\varepsilon}^{-1} r^{(0)}$
$i = 1$
enquanto (tolerância não atingida) faça
    $r^{(i)} = r^{(i-1)} - Ke^{(i-1)}$
    $e^{(i)} = {K_\varepsilon}^{-1} r^{(i)}$
    $u^{(i)} = u^{(i-1)} + e^{(i)}$
    $i = i + 1$
fim enquanto

Algoritmo 1 – Procedimento Babuska

3 MÉTODOS PARA SOLUÇÃO DE SISTEMAS LINEARES

Os algoritmos utilizados na resolução de sistemas lineares estão agrupados em duas categorias: diretos e iterativos. Segundo Rao (2004), um algoritmo que, na ausência de erros de arredondamento e de outros erros, fornece uma solução exata em um número finito de operações algébricas é chamado de **método direto**. Quando um algoritmo gera uma sequência de aproximações que, quando todas as operações aritméticas são executadas corretamente, tende a uma solução exata é chamado de **método iterativo** (LAX, 1997).

Os métodos iterativos são de extrema importância para a álgebra linear, visto que os métodos diretos possuem complexidade próxima a $O(n^3)$, sendo extremamente caro resolvê-los com valores muito grandes de n (TREFETHEN; BAU, 1997), onde n corresponde à dimensão do problema. Dizer que um método possui complexidade $O(n^3)$ significa que, por exemplo, o número de operações de ponto flutuante executadas pelo método é da ordem de n^3.

3.1 Métodos Diretos

O método da **Eliminação de Gauss** e suas variações (RAO, 2004) é um dos métodos mais utilizado na solução de sistemas lineares. Esse método é um método direto eficiente na solução de sistemas cuja matriz possua dimensão inferior a 10.000 linhas aproximadamente. Como os sistemas triangulares são mais fáceis de serem resolvidos, a Eliminação da Gauss consiste em transformar o sistema

$$Ax = b \tag{20}$$

em um sistema triangular equivalente (GOLUB; LOAN, 1996).

Sistema triangular é um sistema cuja matriz dos coeficientes é uma matriz triangular superior ou triangular inferior e, nesse caso, a solução do sistema pode ser obtida, respectivamente por meio da substituição regressiva ou substituição sucessiva, descritas nos Algoritmos 2 e 3, respectivamente. Tanto a substituição sucessiva, quanto a substituição regressiva possuem complexidade próxima a $O(n^2)$.

$$x(n) = \frac{b(n)}{a(n,n)}$$

$$faça\ i = n-1, 1, -1$$

$$val = \sum_{j=i+1}^{n} [a(i,j) * x(j)]$$

$$x(i) = \frac{b(i) - val}{a(i,i)}$$

$$fim\ faça$$

Algoritmo 2 – Substituição Regressiva

$$x(1) = \frac{b(1)}{a(1,1)}$$

$$faça\ i = 2, \quad n$$

$$val = \sum_{j=1}^{i-1} [a(i,j) * x(j)]$$

$$x(i) = \frac{b(i) - val}{a(i,i)}$$

$$fim\ faça$$

Algoritmo 3 – Substituição Sucessiva

Na Substituição Regressiva (Algoritmo 2) os elementos do vetor das incógnitas (vetor x) são obtidos em ordem decrescente, ou seja, do elemento x_n ao x_1. Na Substituição Sucessiva (Algoritmo 3), os elementos do vetor das incógnitas são obtidas em ordem crescente, ou seja, do elemento x_1 ao x_n. Em ambos os algoritmos observa-se que para o cálculo do próximo elemento é necessário o cálculo do elemento anterior, e essa dependência influencia o desempenho do método no ambiente paralelo.

Sendo $i = 1, 2, \ldots, n$, a proposta do método de Gauss é eliminar a variável $x(i)$ das $n - i$ equações do sistema $A^i x = b^i$ (AXELSSON; BARKER, 2001), ou seja, a primeira equação é utilizada para eliminar a incógnita $x(1)$ das $n - 1$ equações restantes, a segunda equação é utilizada para eliminar $x(2)$ das $n - 2$ equações restantes, e assim por diante (FENNER, 1996). No final do processo, a matriz A é uma matriz triangular superior e a solução pode ser obtida por meio da substituição regressiva (BATHE, 1996). O algoritmo da Eliminação de Gauss está descrito no Algoritmo 4.

```
faça i = 1,n − 1
    faça j = i + 1,n
        l(j,i) = a(j,i) / a(i,i)
        faça k = i + 1,n
            a(j,k) = a(j,k) − l(j,i) * a(i,k)
        fim faça
        b(j) = b(j) − l(j,i) * b(i)
    fim faça
fim faça
```

Algoritmo 4 – Eliminação de Gauss sem Pivotamento

Definindo ***U*** como sendo a matriz triangular superior retornada ao fim do Algoritmo 4 (matriz A modificada), e somando as entradas $l(j,i)$ a uma matriz identidade, obtém-se a **Fatoração LU**, dada por

$$A = LU \tag{21}$$

onde ***L*** é uma matriz triangular inferior e ***U*** é uma matriz triangular superior.

Substituindo a Equação 21 na Equação 20, tem-se

$$(LU)x = b \tag{22}$$

e a solução do sistema 22 pode ser obtida em duas etapas:

$$Ly = b \tag{23}$$

$$Ux = y \tag{24}$$

sendo as soluções do sistemas 23 e do sistema 24 obtidas por meio da substituição sucessiva e substituição regressiva, respectivamente.

Se a matriz A é simétrica, é fácil de perceber que

$$U = DL^T \tag{25}$$

onde D é um matriz diagonal composta pelas entradas da diagonal principal de U, e L e U são as matrizes triangular inferior e superior respectivamente, obtidas na fatoração LU (AXELSSON; BAKER, 2001).

Substituindo a Equação 25 na Equação 21, obtém-se a chamada **Fatoração $\boldsymbol{LDL^T}$**, ou seja,

$$A = LDL^T \tag{26}$$

Estando a matriz fatorada, a solução do sistema $(LDL^T)x = b$ é obtida de maneira análoga à obtida por meio da fatoração LU.

Se a matriz A for definida positiva, pode-se definir uma matriz triangular inferior G, tal que

$$G = \sqrt{D} * L \tag{27}$$

onde D é uma matriz diagonal e L uma matriz triangular inferior, ambas obtidas pela Equação 26. Substituindo a Equação 27 na Equação 26, obtém-se a **Fatoração de Cholesky**, definida como

$$A = GG^T \tag{28}$$

A solução do sistema $(GG^T)x = b$ é obtida da mesma forma que a solução obtida por meio da fatoração LU.

A Fatoração de Cholesky só é adequada para matrizes definidas positivas, enquanto que a Fatoração LDL^T pode ser utilizada para matrizes indefinidas (BATHE, 1996).

Tanto a eliminação de Gauss quanto suas variantes (formas fatoradas) possuem um custo computacional muito próximo. Porém, quando a matriz A é utilizada para resolver m sistemas diferentes que possuem a mesma matriz dos coeficientes (como no algoritmo descrito na seção 2.2), as versões fatoradas são mais eficientes (menor custo computacional), pois a fatoração é feita uma única vez e os m sistemas são resolvidos por meio m substituições regressivas e sucessivas (BURDEN; FAIRES, 2008).

3.1.1 Desvantagens dos Métodos Diretos

No método dos elementos finitos (MEF), as matrizes são geralmente esparsas (RAO, 2004) e a esparsidade cresce com o aumento do número das incógnitas (VOLAKIS *et al.*, 1998).

Quando uma matriz é fatorada, por exemplo, utilizando a Fatoração $A = LU$, as matrizes triangulares geradas (L e U) não possuem a mesma esparsidade da matriz A, sendo essas geralmente mais densas. Isso ocorre devido à inserção de elementos não nulos gerados durante a fatoração em posições que são nulas na matriz A. Esses elementos não nulos são conhecidos como preenchimento (do inglês, *fill-in*) (Dongarra *et al.*, 1998).

Os preenchimentos são indesejáveis, pois exigem armazenamento extra e aumentam o número de operações necessárias para realizar a fatoração (VOLAKIS *et al.*, 1998). Segundo Axelsson e Baker (2001), no MEF os preenchimentos são muito influenciados pela escolha da ordenação global da malha. Portanto uma técnica de ordenação, como por exemplo, a Ordenação de Cuthill-McKee (que reduz a banda da matriz) ou a Ordenação do Mínimo Grau (que reduz o número de preenchimentos), pode ser utilizada para minimizar o número de preenchimentos. Segundo Volakis *et al*. (1998) as estratégias de ordenação mais utilizadas são originadas na teoria dos grafos e é extremamente difícil encontrar uma técnica de ordenação ótima que garanta o menor número possível de preenchimentos. Além do mais, os métodos de ordenação requerem elevado tempo computacional (Axelsson; Baker, 2001).

Na solução direta, ao aumentar a dimensão de um sistema esparso, a armazenagem requerida cresce muito mais rápido que o número de elementos nulos (OÑATE, 2009). Portanto eles não são apropriados para sistemas de grandes dimensões.

No ambiente paralelo, os métodos diretos requerem uma grande quantidade de comunicação entre os processadores (WRIGGERS, 2008), devido à pobre eficiência paralela apresentada pelos processos de fatoração (BARRETT *et al.*, 1994).

3.2 Métodos Iterativos

Os sistemas lineares originários da discretização de problemas de elementos finitos geralmente são problemas de elevada ordem, sobretudo os problemas tridimensionais. Frequentemente a largura de banda dessas matrizes é tão grande que os métodos diretos não são eficientes e, mesmo nos casos lineares, os métodos iterativos são aconselhados.

Os métodos iterativos requerem menos memória e possuem menor custo computacional, em comparação com os métodos diretos em problemas de grandes dimensões. O método iterativo mais popular em problemas de mecânica dos sólidos é o Método do Gradiente Conjugado Pré-condicionado (WRIGGERS, 2008).

O MGC faz parte da família dos métodos do subespaço de Krylov. Nessa classe de métodos, a esparsidade da matriz A permite que o produto dessa matriz com qualquer vetor seja computado a um custo muito baixo (ELMAN *et al.*, 2006).

A velocidade de convergência dos métodos de subespaço de Krylov (como o MGC e o Método dos Mínimos Residuais - MINRES) depende das propriedades espectrais da matriz A, tal como a distribuição dos autovalores (DONGARRA *et al.*, 1998).

O número de condicionamento (κ) de uma matriz é definido como

$$\kappa(A) = \frac{\lambda_n}{\lambda_1} \tag{29}$$

onde λ_n e λ_1 correspondem, respectivamente, ao módulo do maior e do menor autovalor da matriz.

Quando o valor do número de condicionamento é elevado, a matriz está mal condicionada e a convergência pode ser muito lenta (BATHE, 1996). Um sistema é mal-condicionado quando qualquer pequena perturbação no sistema pode produzir grandes mudanças na solução exata (MEYER, 2000). Porém, a distinção entre pequena ou grande perturbação, segundo Trefethen e Bau (1997), depende da aplicação. Se for utilizada uma aritmética de n dígitos de ponto flutuante, o erro em $\boldsymbol{b}$ (vetor independente) pode ser tão grande quanto 10^{-n} e o erro relativo em $\boldsymbol{x}$ (vetor das incógnitas) pode ser tão grande quanto $10^{-n}\kappa(A)$ (LAX, 1997). Portanto, o número de condicionamento da matriz além de influenciar a taxa de convergência de um método pode influenciar também a exatidão do resultado.

Segundo Volakis *at al.* (1998), o número de condição de uma matriz geralmente aumenta com o número de variáveis. Axelsson e Barker (2001) apontam como fatores que aumentam o número de condição da matriz:

- Uma malha bastante refinada;
- Falta de suavidade nos dados de um problema de valor de contorno;
- Uma geometria de malha irregular com grande variação na dimensão dos elementos.

3.2.1 Método do Gradiente Conjugado (MGC)

O MGC é um método iterativo utilizado na resolução de sistemas lineares onde a matriz dos coeficientes é simétrica e definida positiva. Foi concebido inicialmente como um método direto, porém, nesse caso ele é menos eficiente que o método da Eliminação de Gauss com pivotamento. Ambos os métodos convergem com o mesmo número de passos, (aproximadamente *n* passos), mas o MGC possui um consumo computacional bem superior (BURDEN; FAIRES, 2008).

O Método do Gradiente Conjugado é utilizado para minimizar um funcional quadrático do tipo

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{30}$$

O gradiente de um funcional quadrático é definido como

$$g(x) = f'(x) \tag{31}$$

portanto o gradiente do funcional 30 é

$$g(x) = Ax - b \tag{32}$$

Igualando o gradiente do funcional a zero, obtém-se o sistema da equação 20. Portanto a solução desse sistema é um ponto critico do funcional 30. Sendo A uma matriz definida positiva, a solução de $Ax = b$ corresponde a um minimizador do funcional 30 (Shewchuk, 1994).

O Método do Gradiente Conjugado produz um minimizador para o funcional $f(x)$ após n iterações, onde n representa a dimensão do sistema. O minimizador é obtido por meio de iterações do tipo

$$x = x + tv \tag{33}$$

onde o vetor v é uma direção de busca, utilizado para melhorar a aproximação da iteração anterior e t é um escalar que fornece o comprimento do passo, dado por:

$$t = \frac{<v,r>}{<v,Av>} \tag{34}$$

sendo $<>$ a representação de produto escalar, ou seja $< v,r >= v^T * r$. O algoritmo completo está exposto no Algoritmo 5.

O primeiro passo do método é definir uma aproximação inicial, x^0, que frequentemente é atribuído como sendo um vetor nulo

$$x^0 = 0 \tag{35}$$

Calcula-se o resíduo para x^0

$$r^0 = -g(x) = b - Ax^0 \tag{36}$$

Como a menor diminuição no valor do gradiente ocorre na direção do resíduo, escolhe-se a primeira direção de busca como sendo

$$v^1 = r^0 \tag{37}$$

Calcula-se então o comprimento do passo (Equação 34) e atualiza-se a aproximação (Equação 33). Em seguida atualiza-se o resíduo

$$r^i = r^{i-1} - tAv^i \tag{38}$$

Calcula-se o melhoramento do passo

$$s = \frac{<r^i, r^i>}{<r^{i-1}, r^{i-1}>} \tag{39}$$

e também o vetor de direção de busca

$$v^{i+1} = r^i + sv^i \tag{40}$$

O método gera, portanto uma sequencia x^i com

$$f(x^0) \geq f(x^1) \geq f(x^2) \geq \cdots \geq f(x^n) \tag{41}$$

O conjunto de vetores $[v^1, v^2, \ldots, v^n]$ satisfazem a uma condição de A-ortogonalidade , ou seja, v^ie o vetor resultando do produto Av^j são ortogonais (Equação 42). Portanto v^ie v^jsão conjugados.

$$< v^i, Av^j > = 0, e \ < r^i, r^j > = 0 \ para \ i \neq j \tag{42}$$

$v = r = b - Ax$
$\alpha = < r, r >$
$enquanto\ (tolerância\ não\ atingida) faça$
    $u = Av$
    $\beta = < v, u >$
    $t = \frac{\alpha}{\beta}$
    $x = x + tv$
    $r = r - tu$
    $\gamma = < r, r >$
    $s = \frac{\gamma}{\alpha}$
    $v = r + sv$
    $\alpha = \gamma$
$fim\ enquanto$

Algoritmo 5 – Gradiente Conjugado

3.2.2 Método do Gradiente Conjugado Pré-condicionado (MGCP)

Quando o número de condicionamento da matriz é grande, o que ocorre com frequência em sistemas que surgem da discretização de problemas de valores de contorno elípticos, O MGC converge mais lentamente podendo até mesmo não convergir. O método trabalha bem com matrizes bem condicionadas ou com poucos autovalores distintos. Portanto a escolha de um bom pré-condicionador tem um forte efeito sobre a taxa de convergência (GOLUB; LOAN, 1997). Por isso frequentemente utiliza-se o Método do Gradiente Conjugado Pré-condicionado (MGCP) ao invés do MGC.

O número de iterações depende da estrutura (esparsidade) da matriz, do seu número de condicionamento, do pré-condicionador utilizado e da precisão requerida (BATHE, 1996).

A convergência de um sistema linear depende de algumas características da matriz dos coeficientes (matriz A) tais como autovalores, número de condicionamento e outras informações. Como nem todas as matrizes possuem as características necessárias para uma boa convergência, a técnica do pré-condicionamento é utilizada para melhorar as propriedades de uma matriz e acelerar a convergência do sistema.

O Algoritmo 6 apresenta o algoritmo do Método do Gradiente Conjugado Pré-condicionado. O algoritmo é bem mais caro computacionalmente que o MGC, visto que além de ser necessário encontrar uma matriz pré-condicionadora (matriz C) que atenda aos pré-requisitos necessários, o método realiza, ainda, uma solução interna de sistema ($v = C^{-1}r$ e

$w = C^{-1}r$). Embora a notação de matriz inversa seja conveniente, seu cálculo não é necessário.

$r = b - Ax$
$v = C^{-1}r$
$\alpha = < r, v >$
enquanto(*tolerância não atingida*)faça
    $u = Av$
    $\beta = < v, u >$
    $t = \frac{\alpha}{\beta}$
    $x = x + tv$
    $r = r - tu$
    $w = C^{-1}r$
    $\gamma = < r, w >$
    $s = \frac{\gamma}{\alpha}$
    $v = w + sv$
    $\alpha = \gamma$
fim enquanto

Algoritmo 6 – Gradiente Conjugado Pré-condicionado

3.3 Pré-condicionadores

Segundo Benzi (2002) um pré-condicionador é uma matriz que transforma um sistema em outro com propriedades mais favoráveis para a solução iterativa. A técnica de pré-condicionamento é utilizada para melhorar as propriedades espectrais da matriz A, e/ou, no caso da matriz ser Simétrica Definida Positiva (SDP), agrupar os autovalores próximo a 1.

Sendo C uma matriz não singular, o intuito do pré-condicionador é pré-multiplicar o sistema 20 por C^{-1} tal que o número de condicionamento de (C^{-1}A) seja menor que o de A, e que o sistema 43 seja mais fácil de resolver. A matriz C^{-1} é a matriz utilizada para pré-condicionar o sistema.

$$C^{-1}Ax = C^{-1}b \tag{43}$$

Além do mais, a matriz C deve ser próxima de A, porém mais fácil de inverter (ALLAIRE, 2007). Se os autovalores de $C^{-1}A$ forem próximos a 1 e $|C^{-1}A^{-1}|_2$ (norma l2) for pequena, então a convergência pode ocorrer mais rapidamente (TREFETHEN; BAU, 1997).

Nos últimos anos muitos estudos têm sido realizados à procura de técnicas de pré-condicionamento eficazes. A próxima seção apresenta alguns dos pré-condicionadores mais utilizados.

3.3.1 Pré-condicionador de Jacobi

Também chamado de Pré-condicionador Diagonal, é o pré-condicionador mais simples. Ele é composto pelos elementos da diagonal principal da matriz ***A*** (Equação 44).

Pode ser utilizado se houver uma variação nos valores dos coeficientes da diagonal principal (DEMMEL, 1996), pois se esses valores forem iguais, o pré-condicionador não terá efeito sobre a convergência. Por outro lado, apresenta resultados significativos quando as magnitudes das entradas da diagonal variam consideravelmente (GOCKENBACK, 2006).

$$c(i,j) = \begin{cases} a(i,j), & se\ i = j \\ 0, & caso\ contrário \end{cases} \tag{44}$$

3.3.2 Pré-condicionador da Sobre-Relaxação Simétrica (SSOR)

Dada a fatoração:

$$A = D - L - U \tag{45}$$

onde D corresponde aos elementos da diagonal de A, e L e U correspondem respectivamente às partes inferior e superior de A, tal que

$$d(i,j) = \begin{cases} a(i,j), & se\ i = j \\ 0, & caso\ contrário \end{cases} \tag{46}$$

$$l(i,j) = \begin{cases} -a(i,j), & se\ i > j \\ 0, & caso\ contrário \end{cases} \tag{47}$$

$$u(i,j) = \begin{cases} -a(i,j), & se\ i < j \\ 0, & caso\ contrário \end{cases} \tag{48}$$

Esse simples, porém, efetivo pré-condicionador é dado por

$$C = \frac{1}{\omega(2-\omega)}(D - \omega L)D^{-1}(D - \omega U) \tag{49}$$

representado matricialmente por

$$c(i,j) = \begin{cases} \frac{a(i,j)}{\omega(2-\omega)}, & se\ i = j \\ \frac{\omega a(i,j)}{a(i,i)}, & se\ i < j \\ \frac{a(i,j)}{(2-\omega)}, & se\ i > j \end{cases} \quad (50)$$

Onde $0 < \omega < 2$, caso contrário a matriz C não será definida positiva.

A escolha de ω não é crítica, porém a escolha de ω ótimo pode melhorar ainda mais a taxa de convergência. É recomendado que ω = [1.2, 1.6] (KNABNER; ANGERMANN, 2003).

Segundo Braess (2007) multiplicar o pré-condicionador por um fator positivo não altera o número de iterações, portanto o fator ω(2-ω) pode ser ignorado na Equação 49.

3.3.3 Pré-condicionador de Fatoração Incompleta

Os processos de fatoração geralmente destroem a estrutura esparsa de uma matriz (inserção de preenchimentos). Uma fatoração incompleta consiste em manter artificialmente a estrutura esparsa da matriz triangular forçando a zero as novas entradas não nulas cujas entradas na mesma posição na matriz $\boldsymbol{A}$ são nulas (descarte de preenchimentos).

Quando se descartam alguns ou todos os preenchimentos, são obtidos pré-condicionadores simples mas poderosos. Os preenchimentos podem ser descartados baseados em diferentes critérios tais como posição, valor ou ambos. Quando todos os preenchimentos são descartados, a fatoração é chamada de nível zero, ocupando exatamente o mesmo espaço de armazenagem da matriz não fatorada. A fatoração incompleta de nível zero é fácil de ser implementada e sua computação é de baixo custo (Benzi, 2002).

Durante a fatoração incompleta pode ocorrer tentativa de divisão por um pivô igual a zero ou mesmo resultar em matrizes indefinidas (pivô negativo), mesmo a fatoração completa sendo possível (BARRETT *et al.*, 1994).

Uma das estratégias utilizadas para transpor esses obstáculos é a utilização da estratégia de mudança dinâmica da diagonal local, utilizada na esperança de que apenas poucos pivôs sejam instáveis (não positivos). Isso pode produzir bons pré-condicionadores, porém infelizmente essa técnica frequentemente produz pré-condicionadores pobres. Para evitar um pivô negativo, pode-se substituí-lo por um valor positivo *p* qualquer, porém isso não é uma tarefa trivial, pois se *p* for muito grande o pré-condicionador será inexato, se for muito pequeno o sistema se tornará instável (Benzi, 2002).

Uma estratégia melhor propõe uma mudança global na diagonal, dada por

$$\hat{A} = A + \alpha * diag(A) \tag{51}$$

Se a fatoração incompleta de $\hat{\boldsymbol{A}}$ falhar, incrementa-se α e repete-se a fatoração até obter sucesso. Existe um valor ótimo para α que garante a fatoração incompleta, que é um valor que torna a matriz $\hat{\boldsymbol{A}}$ diagonalmente dominante. A desvantagem dessa abordagem é a dificuldade de se obter um α ótimo, pois como ele é obtido por meio de tentativa e erro, seu custo é elevado (Benzi, 2002).

Esses pré-condicionadores possuem um custo computacional elevado, mas que pode ser amenizado se o método iterativo realiza muitas iterações ou se o mesmo pré-condicionador puder ser utilizado em vários sistemas lineares (BARRETT *et al.*,1994).

Esse método de pré-condicionamento frequentemente é mais rápido que o SSOR, porém não parece haver uma regra geral que indique qual dos dois é mais efetivo (Braess, 2007).

4 PROCESSAMENTO PARALELO

As máquinas mais utilizadas no processamento paralelo estão incluídas na arquitetura MIMD (*Multiple Instruction, Multiple Data*), conforme a taxonomia de Flynn (EL-REWINI; ABD-EL-BARR, 2005). Nessa arquitetura as máquinas são compostas por múltiplos processadores e múltiplos módulos de memória conectados pela mesma rede de comunicação. Divide-se em três categorias: Memória Compartilhada, Memória Distribuída e Memória Híbrida.

4.1 Memória Compartilhada

Sistemas de memória compartilhada são aqueles em que um endereço de memória é compartilhado por todo o sistema e a comunicação entre os processadores ocorre via variável de dados compartilhada (KSHEMKALYANI; SINGHAL, 2008).

Quando o acesso à memória compartilhada é feita de forma balanceada, esses sistemas são chamados de SMP (*Symmetric Multiprocessor*). Nesse sistema, cada processador tem igual possibilidade de ler/escrever na memória, e com igual velocidade. É necessário que se faça um ‘controle de acesso’, ou seja, determinar qual ou quando um processo pode acessar um recurso. Esse processo pode ser feito por meio da sincronização e da proteção.

A sincronização assegura a funcionalidade do sistema, pois restringe o limite de tempo de acesso de processos compartilhantes a recursos compartilhados.

A proteção é uma característica do sistema que impede processos de realizarem acesso arbitrário a recursos pertencentes a outros processos.

A programação em sistemas de memória compartilhada pode ser feita por meio do uso de *threads* (fluxos de processamento) ou por meio da API (*Applications Programming Interface*) *OpenMP* (*Open Multi Processing*). O *OpenMP* é um modelo de programação baseado em *threads* (GROPP *et al.*, 1999), que simplifica a vida do programador, pois, a paralelização é feita incluindo as diretivas do *OpenMP* nos trechos em que ser pretende paralelizar, inclusive em código já existente.

4.2 Memória Distribuída

Sistemas de memória distribuída também são chamados de sistemas de passagem de mensagem. A passagem de mensagens é um paradigma extremamente utilizado em máquinas paralelas (SNIR *et al.*, 1999). Os agrupamentos de computadores como, por exemplo, os *clusters* utilizam memória distribuída. Nessa arquitetura cada CPU possui sua própria memória e a comunicação entre elas é feita por meio de passagem de mensagem utilizando a rede de comunicação.

A programação em sistemas com memória distribuída é realizada por meio de passagem de mensagens podendo ser feita com o uso de várias APIs, sendo que a mais utilizada é a MPI (*Message Passing Interface*). O MPI controla todo o fluxo de mensagens (envio, recebimento, reenvio caso necessário, etc.), porém o programador é o grande responsável por definir o que será enviado e quando será enviado.

4.3 Memória Híbrida

Sistemas híbridos, também chamados de sistemas de memória compartilhada distribuída e possuem características dos dois grupos anteriores (EL-REWINI; ABD-EL-BARR, 2005).

Nessa arquitetura um agrupamento de computadores é composto por máquinas multiprocessadas, possuindo, portanto, duas formas de comunicação: uma feita internamente entre os processadores por meio do barramento de memória e outra feita entre os nós por meio de passagem de mensagens utilizando a rede de comunicação.

Os sistemas de memória compartilhada distribuída utilizam o MPI para a comunicação internó e o *OpenMP* para a comunicação intranó (KARNIADAKIS;KIRBY II, 2003). Essa integração foi facilitada com a versão MPI-2.

4.4 Modelos de Implementação Paralela

4.4.1 *OpenMP*

O *OpenMP* é um modelo de programação para máquinas multiprocessadas com memória compartilhada ou memória híbrida (CHANDRA *et al.*, 2001) desenvolvido pela SGI em colaboração com outros vendedores de máquinas paralelas e apoiado por mais de 15 vendedores de softwares e desenvolvedores.

O *OpenMP* não é uma nova linguagem de programação, mas sim um conjunto de diretivas de compilação que podem ser utilizados em conjunto com as linguagens Fortran, C ou C++. Essas diretivas são inseridas como comentários no Fortran ou como *pragmas* no C/C++ e entendidas somente por compiladores *OpenMP*, sendo ignoradas pelos demais compiladores.

O *OpenMP* é utilizado frequentemente para paralelizar incrementalmente um código sequencial já existente (CHAPMAN *et al.*, 2008). No Algoritmo 7 encontra-se o código sequencial do produto escalar. O Algoritmo 8 apresenta o código do produto escalar utilizando OpenMP. Percebe-se que a única diferença entre os dois códigos é a presença das diretivas OpenMP no segundo código.

```
subroutine produtoInterno(v1, v2, val, tam)
implicit none
        double precision  :: val, v1(:), v2(:)
        integer :: tam, i
        val = v1(1)*v2(1)
        do i = 2,tam
                val = val + v1(i)*v2(i)
        end do
end subroutine produtoInterno
```

Algoritmo 7 – Produto Interno sequencial

```
subroutine produtoInterno(v1,v2,val,tam,p)
implicit none
        double precision :: val, v1(:), v2(:)
        integer :: tam, i, p
        val = v1(1)*v2(1)
        call OMP_SET_NUM_THREADS(p)
        !$OMP PARALLEL DO DEFAULT(none) &
        !$OMP SHARED(v1,v2,tam) PRIVATE(i) &
        !$OMP REDUCTION(+:val)
                do i = 2,tam
                        val = val + v1(i)*v2(i)
                end do
        !$OMP END PARALLEL DO
end subroutine produtoInterno
```

Algoritmo 8 – Produto Interno utilizando o OpenMP

O Apêndice C contém a descrição das diretivas OpenMP utilizadas nesse trabalho.

4.4.1 *Message Passing Interface* (MPI)

As especificações do MPI foram criadas pelo MPI *Forum*, um grupo formado por vendedores de máquinas paralelas, cientistas da computação e usuários. Assim como o *OpenMP*, o MPI não é uma linguagem de programação e sim um conjunto de especificações que podem ser inseridas em códigos escritos em Fortran, C e C++ e amplamente utilizado no desenvolvimento de aplicativos para máquinas com memória distribuída.

O maior objetivo do MPI é fornecer um grau de portabilidade através de máquinas diferentes (SNIR *et al.*,1999), ou seja, o mesmo código pode ser executado de forma eficiente em uma variedade de máquinas, além de sua habilidade de rodar transparentemente em máquinas heterogêneas.

O mecanismo básico de comunicação do MPI é a transmissão de dados entre um par de processos, chamada de comunicação *point-to-point* (SNIR *et al.*, 1999). Cada processo é identificado por um valor inteiro denominado *rank* que é utilizado, sobretudo para identificar o remetente e o destinatário da mensagem, bem como a porção de código que será executada por cada processador. O processo principal (*master*) é identificado pelo *rank* "0". O Algoritmo 9 apresenta o código do produto interno utilizando o MPI e o Algoritmo 10 apresenta o código do produto interno utilizando a abordagem híbrida. Percebe-se que a única diferença entre eles é a presença das diretivas do OpenMP.

```
subroutine produtoInterno(v1, v2, val, tam_local, rank, processos)
        double precision :: v1(:), v2(:)
        double precision :: val, valor
        integer :: tam_local, i, erro, rank, processos
        ! Cálculo local
        valor = v1(1)*v2(1)
        do i = 2, tam_local
                valor = valor + v1(i)*v2(i)
        end do
        ! Cálculo Global
        if(processos > 1) then
                call MPI_REDUCE(valor, val, 1, MPI_DOUBLE_PRECISION, MPI_SUM, &
0, MPI_COMM_WORLD, erro)
        else
                val = valor
        end if
end subroutine produtoInterno
```

Algoritmo 9 – Produto Interno utilizando o MPI

```
subroutine produtoInterno(v1, v2, val, tam_local, rank, processos, p)
        double precision :: v1(:), v2(:), val, valor
        integer :: tam_local, i, erro, rank, processos
        call OMP_SET_NUM_THREADS(p)
        !Cálculo local
        valor = v1(1)*v2(1)
        !$OMP PARALLEL DO DEFAULT(none) &
        !$OMP SHARED(v1,v2,tam_local) PRIVATE(i) &
        !$OMP REDUCTION(+:valor)
                do i = 2, tam_local
                        valor = valor + v1(i)*v2(i)
                end do
        !$OMP END PARALLEL DO
        !Cálculo global
        if(processos > 1) then
                call MPI_REDUCE(valor, val, 1, MPI_DOUBLE_PRECISION, MPI_SUM, &
0, MPI_COMM_WORLD, erro)
        else
                val = valor
        end if
end subroutine produtoInterno
```

Algoritmo 10 – Produto Interno utilizando a abordagem híbrida (MPI + OpenMP)

O Apêndice D contém a descrição dos comandos MPI utilizados nesse trabalho.

4.4 *Cluster Beowulf*

Cluster é um conjunto de computadores individuais conectados por meio de uma rede de comunicação que se apresentam a seus usuários como um sistema único (VRENIOS, 2002) e que se comportam como um supercomputador.

Diversas arquiteturas de computadores podem ser utilizadas para a paralelização de um programa de computador. Um dos mais conhecidos é o chamado *Beowulf* (STERLING, 2001).

O *Beowulf* é um agrupamento de computadores cujo poder de processamento é bem próximo ao de um supercomputador, porém a um custo muito inferior. O *Beowulf* tem sido muito utilizado no meio acadêmico e científico devido principalmente ao seu alto desempenho e fácil configuração.

O *Beowulf* é um *cluster* centralizado, ou seja, todo o sistema é controlado por uma máquina central que é responsável, sobretudo pela distribuição da carga a ser processada nas demais máquinas do *cluster*. Todo o processo é feito por meio de passagem de mensagens, tornando a rede de comunicação um fator importante para o sucesso do sistema.

O Apêndice B apresenta um tutorial para a implementação de um *cluster Beowulf*.

4.5 Análise de Desempenho

Uma questão importante na programação paralela é o desempenho de um sistema paralelo. O cálculo do desempenho pode ser feito por meio de métricas diferentes, sendo que as mais utilizadas são:

a) *Performance de pico teórico* – corresponde ao número máximo de operações de ponto flutuante por segundo que o sistema pode alcançar (STERLING, 2001). É dada pela fórmula

$$P = n * c * f * r \tag{52}$$

Onde n é o número de nós, c o número de CPUs, f o número de operações de ponto flutuante por período de *clock* e r a taxa de *clock* medida em ciclos por segundo. A unidade de P é *flops (floating point operations per second)*.

b) *Performance de Aplicação* – corresponde ao número de operações realizadas pelo sistema durante a execução de um problema dividido pelo tempo gasto. Também é

dada em *flops*. Seu resultado é mais significativo que a *performance de pico* teórico, pois fornece informações do desempenho real da aplicação e não do desempenho que o sistema pode atingir.

c) *Speed-up* (escalabilidade) – fator pelo qual o tempo de uma solução pode ser melhorado quando comparado com o tempo de um único processador (SNIR *et al.*, 1999). Pode ser calculado por meio da equação

$$S(n) = \frac{T_1}{T_n} \tag{53}$$

onde T_1 e T_n correspondem, respectivamente, ao tempo gasto por um processador e ao tempo gasto por vários processadores para executar uma mesma tarefa. Uma variável importante e que pode ser considerada no cálculo do *speed-up* é o *overhead* de comunicação (T_c) que corresponde ao tempo que um processador necessita para se comunicar e trocar dados enquanto executa suas subtarefas (EL-REWINI; ABD-EL-BARR, 2005). Portanto

$$T_n = \frac{T_1}{n} + T_c \tag{54}$$

$$S(n) = \frac{n}{1+n*\frac{T_c}{T_1}} \tag{55}$$

d) *Eficiência paralela* – Corresponde à medida do *speed-up* atingida por processador (EL-REWINI; ABD-EL-BARR, 2005). Considerando-se o tempo de comunicação é dada por

$$\xi = \frac{s(n)}{n} = \frac{1}{1+n*\frac{T_c}{T_1}} \tag{56}$$

É esperado um valor próximo de 1.

5 CÓDIGO DESENVOLVIDO

Todo o código desenvolvido foi escrito em linguagem Fortran, utilizando a versão conhecida como “Fortran95”. Inicialmente foi desenvolvida a versão sequencial (alguns seguimentos código já apresentados em seções anteriores) e, posteriormente, foi modificada para suas versões paralelas em OpenMP, MPI e Híbrida.

Nesta etapa foram analisadas as operações realizadas por trechos dos algoritmos agrupando-as em tarefas.

O primeiro algoritmo analisado foi o “Procedimento Babuska”, exposto no Algoritmo 1. Agrupando as operações desse algoritmo em tarefas, verificou-se que o mesmo realiza quatro tarefas distintas. São elas:

- Resolução de sistema linear (linhas 2, 4 e 8);
- Multiplicação matriz x vetor (linha 7);
- Atualização vetorial (linha 3, 7 e 9) e;
- Atualização matricial (linha 1).

Em seguida foi analisado o código do MGC e do MGCP (Algoritmo 5 e Algoritmo 6, respectivamente), visto que um ou outro pode ser utilizado para solucionar os sistemas das linhas 2, 4 e 8 do Procedimento Babuska. Analisando os Algoritmos 5 e 6, detectou-se as seguintes tarefas:

- Atualização vetorial (linhas 1, 7, 8 e 11 do MGC);
- Cálculo do produto interno (linhas 2, 5 e 9 do MGC);
- Multiplicação matriz x vetor (linha 4);
- Pré-condicionamento (utilizado nas linhas 2 e 10 do MGCP).

Para o MGCP é necessário o cálculo da matriz pré-condicionadora, sendo essa outra tarefa a ser paralelizada.

Portanto concluiu-se que as tarefas a serem implementadas para a obtenção da solução do sistema são:

- Cálculo de produto interno;
- Atualização vetorial;
- Multiplicação matriz x vetor;

- Atualização matricial
- Pré-condicionamento
- Pré-condicionador

Dentre essas tarefas uma já foi apresentada em sua versão paralelizada: o cálculo do produto interno (Algoritmos 8, 9 e 10). Outras duas tarefas representam operações triviais e, portanto não serão implementadas como rotinas específicas: a atualização vetorial e a atualização matricial. Portanto, versões paralelizadas das tarefas 'multiplicação matriz x vetor' e 'pré-condicionador' serão abordadas nas seções seguintes.

Neste trabalho foram implementados os pré-condicionadores de Jacobi e o SSOR. A escolha do pré-condicionador influência na implementação da resolução dos sistemas exigido nas linhas 2 e 10 do Algoritmo 6. Se o pré-condicionador escolhido for o pré-condicionador de Jacobi, essas duas linhas são implementadas como multiplicação matriz-vetor, visto que o algoritmo proposto do pré-condicionador diagonal retorna uma matriz pré-condicionadora já invertida. Porém se o pré-condicionador utilizado for o pré-condicionador SSOR, as linhas 2 e 10 desse algoritmo são resolvidas por meio da substituição sucessiva e substituição regressiva, visto que o algoritmo proposto retorna a matriz em sua forma fatorada na versão LU.

Quando o pré-condicionador de Jacobi é utilizado, a matriz não é normalizada, pois uma matriz normalizada possui todas as entradas da diagonal principal igual a um e, quando não há variações nas entradas da diagonal principal, esse pré-condicionador não surte efeito.

Antes, porém, torna-se necessário um estudo sobre um esquema de armazenamento de matrizes, pois sua influência no tempo de solução do problema é bastante expressiva. Todos os algoritmos paralelos apresentados foram implementados utilizando o formato CRS.

5.1 Esquemas de Armazenamento

Geralmente os sistemas lineares obtidos no MEF são muito esparsos e essa esparsidade cresce como o aumento do número de variáveis, tornando necessário que estas matrizes sejam armazenadas de forma eficiente.

Existem vários esquemas de armazenagem de matrizes esparsas, sendo que o *Compressed Row Storage* (CRS) é um dos mais utilizado (VOLAKIS *et al.*, 1998). O formato CRS armazena os elementos não nulos da matriz linha a linha, juntamente com seu índice. Um esquema análogo ao CRS é o *Compressed Column Storage* (CCS), cujo armazenamento é

realizado coluna a coluna. Ambos utilizam endereçamento indireto em suas operações como, por exemplo, na multiplicação matriz-vetor, o que pode tornar seu uso proibitivo (BARRETT *et al.*, 1994). Esse endereçamento indireto é mais comum em matrizes simétricas quando somente a porção simétrica é armazenada (VOLAKIS *et al.*, 1998).

Para o esquema CRS é necessária uma estrutura de dados composta por quatro vetores. São eles:

VAL – vetor do tipo real onde os elementos não nulos da matriz são armazenados. Sua dimensão corresponde ao número de elementos não nulos da matriz.

COL – vetor do tipo inteiro que armazena o índice na linha do elemento armazenado, ou seja, em que coluna ele se encontra. Possui a mesma dimensão de VAL.

ROW – vetor do tipo inteiro que armazena o índice em VAL do primeiro elemento não nulo de cada linha. Sua dimensão é $n + 1$, onde n é a dimensão da matriz.

DIAG – vetor do tipo inteiro que armazena o índice em VAL do pivô ($entrada\ a(i,i)$) de cada linha. Esse vetor é opcional, e possui dimensão n.

O Algoritmo 11 foi utilizado para armazenar uma dada matriz no formato CRS. Essa rotina recebe como entrada uma matriz qualquer n x n e retorna os vetores val, col, row e $diag$.

```
l = 1   ! Contador para a variável row
k = 1   ! Contador para as variáveis val e col
m = 1  ! Contador para a variável diag
faça i = 1, n
    row(l) = k
    l = l + 1
    faça j = 1, n
        se(a(i,j) /= 0) então
            val(k) = a(i,j)
            col(k) = j
            se(i == j) então
                diag(m) = k
                m = m + 1
            fim se
            k = k + 1
        fim se
    fim faça
fim faça
row(n + 1) = k
```

Algoritmo 11 – Armazenamento CRS

O Algoritmo 12 apresenta uma multiplicação matriz-vetor com a matriz armazenada no formato CRS. Essa rotina recebe como entrada os vetores val, row, col e $vetor$ (vetor operando) e retorna o resultado no vetor res. Observa-se que na linha 3 o intervalo $[row(i), row(i+1)-1]$ contém os índices que em val correspondem aos elementos não-nulos da linha i da matriz. Percebe-se também que na linha 4 o índice do elemento de $vetor$ que opera com o elemento em $val(j)$ é definido por $col(j)$. Isso corresponde ao endereçamento indireto.

```
faça i = 1, n
    res(i) = 0
    faça j = row(i), row(i + 1) − 1
        res(i) = res(i) + val(j) ∗ vetor(col(j))
    fim faça
fim faça
```

Algoritmo 12 – Multiplicação Matriz-Vetor no Formato CRS

Os Algoritmos 13 e 14 apresentam respectivamente os códigos da Substituição Sucessiva e da Substituição Regressiva, respectivamente, com a matriz armazenada no formato CRS. O intervalo $[row(i),\ diag(i)-1]$, encontrado na linha 4 do Algoritmo 13, é utilizado para definir o índice dos elementos não nulos da linha (i) de uma matriz triangular inferior em val. De modo análogo o intervalo $[diag(i)+1,\ row(i+1)-1]$, encontrado na linha 4 do Algoritmo 14, é utilizado para definir o índice dos elementos não nulos da linha (i) de uma matriz triangular inferior em val.

```
x(1) = b(1) / val(diag(1))
faça i = 2, n
    soma = 0
    faça j = row(i), diag(i) − 1
        soma = soma + val(j) ∗ x(col(j))
    fim faça
    x(i) = (b(i) − soma) / val(diag(i))
fim faça
```

Algoritmo 13 – Substituição Sucessiva no formato CRS

$$x(n) = \frac{b(n)}{val(diag(n))}$$
$$faça\ i\ =\ n-1,\ 1,\ -1$$
$$soma = 0$$
$$faça\ j\ =\ diag(i)+1, row(i+1)-1$$
$$soma\ =\ soma\ +\ val(j) * x(col(j))$$
$$fim\ faça$$
$$x(i) = \frac{b(i)-soma}{val(diag(i))}$$
$$end\ do$$

Algoritmo 14 – Substituição Regressiva no formato CRS

O Algoritmo 15 apresenta o código para a obtenção do Pré-condicionador da SSOR. Nesse algoritmo, um vetor do tipo inteiro, $inc()$, é utilizado para obter o índice do elemento a ser alterado. O valor de $row(i)\ +\ inc(i)$ é utilizado para definir o índice do elemento que será acessado. O vetor inc funciona como um controle de passo.

$$v\ =\ w * (2-w)$$
$$valor\ =\ 2-w$$
$$faça\ i\ =\ 1, n$$
$$c(diag(i))\ =\ val(diag(i))/v$$
$$faça\ j\ =\ diag(i)+1, row(i+1)-1$$
$$c(j)\ =\ val(j)/valor$$
$$ind2\ =\ col(j)$$
$$ind\ =\ r(ind2) + inc((ind2))$$
$$c(ind) = val(ind) * w/val(diag(ind2))$$
$$inc(ind2)\ =\ inc(ind2)\ +\ 1$$
$$fim\ faça$$
$$fim\ faça$$

Algoritmo 15 – Pré-condicionador da SSOR no formato CRS

5.2 Implementação Utilizando o OpenMP

Dos três modelos de programação paralela estudados, o OpenMP é o que possui a implementação mais simples. Basta encontrar as porções do código que podem ser executadas em paralelo e inserir as diretivas de OpenMP correspondentes.

Um código paralelo para a multiplicação matriz-vetor está exposto no Algoritmo 16. Essa rotina recebe a matriz armazenada no formato CRS (os vetores val, col e row) e o vetor operando, e retorna o resultado no vetor res.

```
subroutine multiMVC(val, col, row, vetor, res, tam, p)
      double precision :: val(:), vetor(:), res(:)
      integer :: col(:), row(:), tam, i, j, p
      call OMP_SET_NUM_THREADS(p)
      !$OMP PARALLEL DO DEFAULT(NONE) &
      !$OMP SHARED(row, vetor, res, val, col, tam) PRIVATE(i,j)
            do i = 1, tam
                res(i) = 0
                do j = row(i),row(i+1)-1
                      res(i) = res(i) + val(j)*vetor(col(j))
                end do
            end do
      !$OMP END PARALLEL DO
end subroutine multiMVC
```

Algoritmo 16 – Multiplicação Matriz-Vetor utilizando a abordagem OpenMP

A outra tarefa implementada foi o pré-condicionador. Eles são apresentados nos Algoritmos 17 e 18.

O Algoritmo 17 é apresenta o código para o pré-condicionador de Jacobi. Percebe-se que o valor fornecido por $diag(i)$ na linha 4 fornece a posição em val do pivô $a(i,i)$. Esse código retorna a matriz pré-condicionadora já invertida.

O Algoritmo 18 exibe um código para o pré-condicionador da SSOR. Nesse código a matriz pré-condicionadora é retornada fatorada no formato LU.

```
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED(c,val,diag,tam) PRIVATE(i)
      do i = 1,tam
            c(i) = 1/val(diag(i))
      end do
!$OMP END PARALLEL DO
```

Algoritmo 17 – Pré-condicionador de Jacobi em sua versão paralela utilizando OpenMP

```
v1 = 2-w
v2 = w*v1
!$OMP PARALLEL DO DEFAULT(NONE)  &
!$OMP SHARED(row,diag,val,c,tam,inc,i,col,v1,v2,w)  PRIVATE(i, j, ind, ind2)
      do i = 1,tam
            c(diag(i)) = val(diag(i))/v2
            do j = diag(i)+1,row(i+1)-1
                  c(j) = val(j)/v1
                  ind2 = col(j)
                  ind = row(ind2)+inc((ind2))
                  c(ind) = val(j)*w/val(diag(ind2))
                  !$OMP CRITICAL
                        inc(ind2) = inc(ind2) + 1
                  !$OMP END CRITICAL
            end do
      end do
!$OMP END PARALLEL DO
```

Algoritmo 18 – Pré-condicionador da SSOR em sua versão paralela utilizando OpenMP

Com base nas implementações já citadas, o código do Gradiente Conjugado é fornecido no Algoritmo 19. A única Sub-rotina ainda não comentada é a Sub-rotina $residuo()$ que corresponde ao cálculo $b - Ax$.

Como a única diferença entre os algoritmos do MGC e do MGCP são as linhas do pré-condicionamento, essas podem ser calculadas utilizando o Algoritmo 20. Se a variável $tipo$ for igual 1, o pré-condicionador utilizado é o Pré-condicionador de Jacobi, se for igual a 2, é utilizado o Pré-condicionador SSOR. Se o primeiro argumento da Sub-rotina $substituicaoRegressiva()$ for igual a 1, será executado o código da substituição sucessiva, caso contrário, será executado o código da substituição regressiva.

Para finalizar as implementações utilizando o OpenMP, o Algoritmo 21 apresenta o código do Procedimento Babuska, que engloba todas as implementações aqui citadas. Nesse código há uma Sub-rotina não fornecida, $normaRelativa()$, que corresponde ao critério de parada do método. O Algoritmo 21 recebe a matriz de rigidez já perturbada, sendo essa rotina executada em um algoritmo a parte.

```
call residuo(val, col, row, b, x, r, tam, p)
v = r
call produtoInterno(r, r,alfa,tam,p)
 it = 1
do while (it <= tam)
        call multiMVC(1, val, col, row, v, u, tam, p)
        call produtoInterno(u, v,beta,tam,p)
        t = alfa/beta
        x = x + t*v
        r = r - t*u
        call produtoInterno(r, r, gama, tam, p)
        s = gama/alfa
        v = r + s*v
        call produtoInterno(r, r, norma_e, tam, p)
        norma_e = dsqrt(norma_e)
        if(norma_e < tol) then
              exit
        else
              alfa = gama
              it = it + 1
        end if
end do
```

Algoritmo 19 – Gradiente Conjugado utilizando a abordagem OpenMP

```
select case(tipo)
        case(1)
              do i = 1,tam
                    v(i) = c(i)*r(i)
              end do
        case(2)
              call substituicaoRegressiva(1, c, col, row, d, r, y, tam)
              call substituicaoRegressiva(2, c, col, row, d, y, v, tam)
end select
```

Algoritmo 20 – Rotina de Pré-condicionamento do MGCP

```
call mgc (ve, col, row, bt, u, tol, tam, p)
call residuo(vt, col, row, bt, u, r, tam, p)
call mgc(ve, col, row, r, e, tol, tam, p)
val = normaRelativa(vt, col, row, e, u, aux1, tam, p)
if(val > tol) then
      i = 1
      do while(i < 3)
        r = r - aux1
        call mgc(ve, col, row, r, e, tol, tam, p)
        u = u + e
        val = normaRelativa(vt, col, row, e, u, aux1, tam, p)
        if(val < tol) then
              exit
        else
              i = i + 1
        end if
end do
```

Algoritmo 21 – Procedimento Babuska utilizando o OpenMP

5.3 Implementação Utilizando o MPI

A paralelização usando o modelo de programação MPI é muito mais complexa do que a paralelização utilizando o OpenMP. Nesse etapa é necessário definir algumas estratégias para o funcionamento do *cluster*. É necessário definir, por exemplo, como os dados serão divididos entre as máquinas (conhecida como decomposição do domínio ou decomposição do problema) e se o processo *master* será responsável somente pela administração da execução ou se ele também realizará processamento. Nesse trabalho ficou definido, devido ao tamanho pequeno do cluster implementado, que o *master* também participará do processamento.

As variáveis utilizadas e suas funções são exibidas na Tabela 1.

TABELA 1 – Variáveis utilizadas nesse trabalho (implementação do MPI)

VARIÁVEIS	DESCRIÇÃO
tam_global	Variável inteira que armazena a dimensão da matriz utilizada
tam_local	Variável inteira que armazena a porção do sistema será utilizada na máquina corrente
processos	Variável inteira que armazena o número de processos que serão executados

Rank	Variável inteira que corresponde ao ID (identificação) de um nó. Por padrão à máquina *master* é atribuído o *rank* = 0
Eu	Variável inteira que corresponde a *rank* + 1.
Ant	Variável inteira que armazena o *rank* de um nó vizinho cujo *rank* seja inferior ao 'eu'
Prox	Variável inteira que armazena o *rank* de um nó vizinho cujo *rank* seja superior ao 'eu'
tamanhos()	Vetor do tipo inteiro que armazena o tamanho local de cada máquina. O valor da posição i+1 desse vetor corresponde ao 'tam_local' da máquina de *rank* = i. A dimensão desse vetor é 'processos'
limites()	Vetor do tipo inteiro que define a partir de qual índice de uma matriz ou vetor inicia-se o segmento de uma máquina. Esse valor é utilizado, sobretudo para encontrar a posição de um pivô em um seguimento matricial dado.

Antes de apresentar as implementações propostas, é necessário definir como os dados são divididos entre as máquinas. Optou-se por realizar a divisão da seguinte forma:

1. Divide-se tam_global pelo número processos;
2. Se tam_global for divisível por processos, o tam_local de cada máquina será definido como o valor da divisão, caso contrário o restante será repartido entre as demais máquinas, excluindo o *master*;
3. Define-se o limite de cada processo como sendo o valor correspondente à soma do tamanho do processo anterior com o limite do processo anterior. O código em Fortran desse procedimento encontra-se no Algoritmo 22. Esse código é executado pelo processo *master*.

```
subroutine balanceamento(tam_global, tamanhos, limites, processos)
        integer, dimension(:) :: tamanhos,limites
        integer tam_global, processos,  i, div, resto
        div = tam_global/processos
        resto = mod(tam_global, processos)
        tamanhos(1) = div
        limites(1) = 0
        do i = 1, processos - 1
                if(i <= resto) then
                        tamanhos(i + 1) = div + 1
                else
                        tamanhos(i + 1) = div
                end if
                limites(i+1) = limites(i)+tamanhos(i)
        end do
end subroutine balanceamento
```

Algoritmo 22 – Balanceamento de carga utilizado

Na multiplicação matriz-vetor (Algoritmo 23) o código é divido em duas etapas: cálculo local e calculo global. No cálculo local o segmento de código é semelhante ao código da multiplicação matriz-vetor sequencial. No cálculo global os valores locais são reunidos, somados e armazenados no *master*.

```
subroutine multiMV(val, col, row, vetor, res, tam_local, processos, rank)
implicit none
        double precision :: val(:), vetor(:), res(:)
        double precision, allocatable :: valor(:)
        integer :: col(:), row(:), processos, rank, tam_local, i, j, erro
        allocate(valor(tam_local))
        !Cálculo local
        valor = 0.0D0
        do i = 1, tam_local
                do j = row(i), row(i+1)-1
                        valor(i) = valor(i) + val(j)*vetor(col(j))
                end do
        end do
        !Cálculo global
        call MPI_REDUCE(valor, res, tam_local, MPI_DOUBLE_PRECISION, MPI_SUM, &
0, MPI_COMM_WORLD, erro)
        deallocate(valor)
end subroutine multiMV
```

Algoritmo 23 – Multiplicação matriz-vetor utilizando a abordagem MPI

O Algoritmo 24 exibe o segmento de código para o cálculo do pré-condicionador de Jacobi. Percebe-se que não há nenhuma rotina de comunicação MPI, sendo essa uma das vantagens desse pré-condicionador, ou seja, cada processo constrói seu segmento do pré-condicionador sem a necessidade de comunicação com os outros processos.

```
do i = 1,tam_local
      c(i) = 1/val(diag(i))
end do
```

Algoritmo 24 – Pré-condicionador de Jacobi utilizando a abordagem MPI

O Algoritmo 25 apresenta o código do pré-condicionador da SSOR que, por ser é um pouco mais complexo, foi dividido em três segmentos, exibidos nos Algoritmos 26, 27 e 28. A variável p é utilizada nesses quatro algoritmos define qual segmento do código uma determinada máquina executará. Quando a variável eu de um processo possui o mesmo valor de p indica que essa máquina está processando a subregião da matriz que contém os pivôs e, nesse caso, o código é semelhante ao código do processamento sequencial. Quando a variável eu é menor que p, indica que o processo está em uma subregião da matriz que se encontra em posição abaixo do pivô. Se eu for maior que p, indica que esse processo está processando dados que se encontram em posição superior ao pivô. Observa-se que nesse algoritmo cada processo constrói seu segmento de pré-condicionador sem a necessidade de comunicação com os outros processos.

```
v1 = 2-w
v2 = w*v1
do p = 1, processos
        if(p == eu) then
                !segmento 1
        else
                if(p > eu) then
                        !segmento 2
                else
                        !segmento 3
                end if
        end if
end do
```

Algoritmo 25 – Pré-condicionador SSOR utilizando a abordagem MPI

```
do i = 1,tam_local
        c(diag(i)) = val(diag(i))/v2
        do j = diag(i)+1,row(i+limites(eu)+1)-1
                c(j) = val(j)/v1
                ind2 = col(j)+limites(eu)
                ind = row(ind2)+inc((ind2))
                c(ind) = val(ind)*w/val(diag(i))
                inc(ind2) = inc(ind2) + 1
        end do
end do
```

Algoritmo 26 – Pré-condicionador SSOR utilizando a abordagem MPI: Segmento 1

```
do i = 1,tamanhos(p)
        do j = row(i+limites(p)),  row(i+1+limites(p))-1
                c(j) = val(j)*w/val(diag(col(j)))
        end do
end do
```

Algoritmo 27 – Pré-condicionador SSOR utilizando a abordagem MPI: Segmento 2

```
do i = 1+limites(p),limites(p)+tamanhos(p)
      do j = row(i), row(i+1)-1
            c(j) = val(j)/v1
      end do
end do
```

Algoritmo 28 – Pré-condicionador SSOR utilizando a abordagem MPI: Segmento 3

O algoritmo da Substituição Sucessiva também foi dividido em segmentos, expostos nos Algoritmos 29, 30, 31 e 32.

No Algoritmo 30 (segmento 1) é realizado o cálculo da primeira posição do vetor x local. O *master* define o valor de $x(1)$ simplesmente atribuindo a ele o valor da divisão de $b(1)$ pelo pivô correspondente. As demais máquinas só calculam sua primeira incógnita quando recebem o valor de val calculada e enviada por ser visinho imediato à esquerda e, enquanto esse valor não é recebido, a máquina fica bloqueada, aguardando.

O segmento de código exibido no Algoritmo 31(segmento 2) é executado por apenas uma máquina por vez e, quando uma máquina está processando esse segmento, seu vizinho imediato à direta está aguardando. Nesse segmento a máquina corrente calcula seu vetor solução local.

O segmento exposto no Algoritmo 32 (segmento 3) é executado pelos processos que já calcularam seu vetor solução local. Nesse segmento os processos auxiliam seus processos vizinhos à direita efetuando cálculos com seu segmento local, enviando-os em seguida.

A solução final é então obtida recolhendo-se as soluções locais e montando a solução global (última linha do Algoritmo 29).

```
!segmento 1
!segmento 2
!segmento 3
call MPI_GATHERV(aux, tam_local, MPI_DOUBLE_PRECISION, x, tamanhos, limites, &
MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, erro)
```

Algoritmo 29 – Substituição sucessiva utilizando MPI

```
if(rank == 0) then
      x(1) =b(1)/val(diag(1))
else
      call MPI_RECV(valor, 1, MPI_DOUBLE_PRECISION, esq, MPI_ANY_TAG, &
MPI_COMM_WORLD, status, erro)
      x(1) = (b(1) – valor)/val(diag(1))
end if
```

Algoritmo 30 – Substituição sucessiva utilizando MPI: Segmento 1

```
do i = 2, tam_local
      if(esq /= MPI_PROC_NULL) then
            call MPI_RECV(valor, 1, MPI_DOUBLE_PRECISION, esq, MPI_ANY_TAG, &
MPI_COMM_WORLD, status, erro)
      else
            valor = 0
      end if
      do j = row(i+limites(eu)),diag(i)-1
            valor = valor + val(j)*x(col(j))
      end do
      x(i) = (b(i) – valor)/val(diag(i))
end do
```

Algoritmo 31 – Substituição sucessiva utilizando MPI: Segmento 2

```
if(dir /= MPI_PROC_NULL) then
      do i = limites(prox)+1,tam_global
            if(esq /= MPI_PROC_NULL) then
                  call MPI_RECV(valor, 1, MPI_DOUBLE_PRECISION, esq, MPI_ANY_TAG, &
MPI_COMM_WORLD, status, erro)
            else
                  valor = 0
            end if
            do j = row(i),row(i+1)-1
                  valor = valor + val(j)*aux(col(j))
            end do
            if(processos > 1) then
                  call MPI_SEND(valor, 1, MPI_DOUBLE_PRECISION, dir, dir, &
MPI_COMM_WORLD, erro)
            end if
      end do
end if
```

Algoritmo 32 – Substituição sucessiva utilizando MPI: Segmento 3

Para finalizar as implementações utilizando o MPI, o Gradiente conjugado é então fornecido no Algoritmo 33 e o Procedimento Babuska apresentado no Algoritmo 34.

```
call residuo(val, col, row, b, x, r, tam_global, tamanhos, processos, rank, limites); v = r
call produtoInterno(r, r, alfa, tam_local,rank,processos); it = 1
do while (it <= tam_global)
      call multiMV(1, val, col, row, v, u, tam_global)
      call MPI_SCATTERV(u, tamanhos, limites, MPI_DOUBLE_PRECISION, aux_u, &
tamanhos(rank+1), MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, erro)
      call produtoInterno(aux_u, v, beta, tam_local,rank,processos)
      if(rank==0) t = alfa/beta
      call MPI_BCAST(t,1,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,erro)
      x = x + (t*v)
      r = r - (t*aux_u)
      call produtoInterno(r, r, gama, tam_local,rank,processos)
      if(rank==0) s = gama/alfa
      call MPI_BCAST(s,1,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,erro)
      v = r + (s*v)
      call produtoInterno(v, v, norma_e, tam_local,rank,processos)
      call MPI_BCAST(sqrt(norma_e),1,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,erro)
      if(norma_e < tol) then
            exit
      else
            alfa = gama; it = it + 1
      end if
end do
```

Algoritmo 33 – Gradiente Conjugado utilizando MPI

```
call mgc(ve, col, row, bt, u, tol, tam_global, tamanhos, processos, rank, limites)
call residuo(vt, col, row, bt, u, r, tam_global, tamanhos, processos, rank, limites)
call mgc(ve,col, row, r, e, tol, tam_global, tamanhos, processos, rank, limites)
call normaRelativa(vt, col, row, e, u, val, aux1, tam_global, tamanhos, limites, processos, rank)
if(val > tol) then
      i = 1
      do while(i < 3)
            r = r - aux1
            call mgc(ve,c,row,r,e,tol,tam_global,tamanhos,processos,rank,limites)
            u = u + e
            call normaRelativa(vt, c, row, e, u, val, aux1, tam_global, tamanhos, limites, &
processos,rank)
            if(val < tol) then
                  exit
            else
                  i = i + 1
            end if
      end do
endif
```

Algoritmo 34 – Procedimento Babuska utilizando MPI

5.4 Implementação Utilizando a Abordagem Híbrida

Uma vez obtido o código MPI, a abordagem híbrida é facilmente obtida por meio de inserção das diretivas OpenMP. O Algoritmo 35 apresenta o código da multiplicação matriz-vetor é exibido. Os demais algoritmos são obtidos de maneira semelhante.

```
subroutine multiMV(val, col, row, vetor, res, tam, processos, rank, p)
implicit none
      double precision :: val(:), vetor(:), res(:)
      integer :: col(:), row(:),processos,rank,  tam,i,j,erro,p
      double precision, allocatable:: valor(:)
      call OMP_SET_NUM_THREADS(p)
      allocate(valor(tam))
      valor = 0.0D0
      !$OMP PARALLEL DO DEFAULT(NONE)  &
      !$OMP SHARED(row,vetor,valor,val,col,tam)  PRIVATE(i,j)
            do i = 1, tam
                  do j = row(i),row(i+1)-1
                        valor(i) = valor(i) + val(j)*vetor(col(j))
                  end do
            end do
      !$OMP END PARALLEL DO
      call MPI_REDUCE(valor, res, tam, MPI_DOUBLE_PRECISION, MPI_SUM, 0, &
MPI_COMM_WORLD, erro)
      deallocate(valor)
end subroutine multiMVCP
```

Algoritmo 35 - Multiplicação matriz-vetor utilizando a abordagem Híbrida

6 RESULTADOS NUMÉRICOS

Para análise dos gráficos e tabelas dessa seção, considerar:

- Siglas utilizadas

  SN Matriz dos coeficientes sem normalização;

  CN Matriz dos coeficientes normalizada;

  SP Método sem pré-condicionamento (MGC);

  PJ MGCP com Pré-condicionador de Jacobi;

  PS MGCP com Pré-condicionador SSOR;

- *Número de nós* corresponde ao número de núcleos utilizados na abordagem OpenMP ou número de máquinas utilizadas na abordagem MPI. Na abordagem híbrida, os testes foram realizados utilizando os quatro núcleos disponíveis em cada máquina.
- O tempo de execução é dado em segundos.
- Para os testes utilizando o Pré-condicionador SSOR, o fator de sobre-relaxação (ω) foi dado *a priori*.

6.1 Ambiente de Teste

Todos os testes foram realizados no laboratório CCC (Centro de Computação Científica) do CEFET-MG, onde foi implementado um *cluster Beowulf* composto por três máquinas com as seguintes características:

TABELA 2 – Características das máquinas que compõem o cluster implementado

Número de máquinas	3
Processador	Q8400 2.66 GHz
Núcleos de processamento por máquina	4
Memória RAM	4 GB
Sistema Operacional	Ubuntu 10.04
Switch	modelo 4200 26 portas da 3Com

Todo o código utilizado foi escrito em Fortran (Fortran 95) e foram desenvolvidos códigos para sistemas utilizando armazenamento denso e código para sistemas utilizando armazenamento compacto (CRS). Foi utilizada a implementação do MPI 2 fornecida no pacote MPICH2.

6.2 Modelo Analisado

O sistema analisado foi gentilmente cedido pelo professor Felício Bruzzi Barros (Departamento de Engenharia de Estrutura da UFMG). Ele corresponde à análise por MEFG de uma placa engastada em uma das extremidades, submetida a uma carga uniformemente distribuída verticalmente para baixo, com valor de 10 KN /m (Figura 1) com as características descritas na Tabela 3.

TABELA 3 – Características da placa modelada

Módulo de elasticidade (GPa)	*200*
Coeficiente de Poisson	0,33
Dimensões (mm)	160x40x12

A malha foi gerada com 576 elementos (48 ao longo do comprimento e 12 ao longo da altura), formada por elementos quadrilaterais isoparamétricos com quatro nós (totalizando 637 nós), e enriquecido com monômios como esquema de enriquecimento do MEFG.

O enriquecimento gerou uma função de aproximação de grau 4 com 16 funções de forma para cada nó, gerando um sistema de 10192 equações, ocupando em disco um espaço de aproximadamente 2,3 Gb. Essa matriz possui 1.282.164 elementos não nulos.

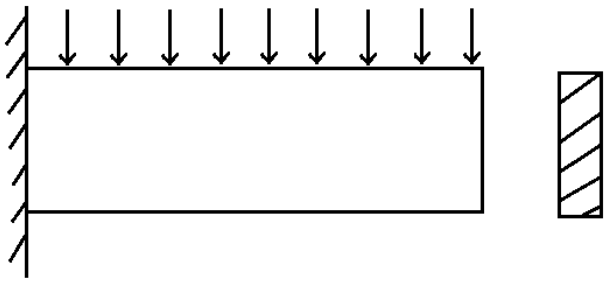

Figura 1 – Placa modelada

Os testes foram então realizados utilizando as três abordagens: utilizando somente o OpenMP, utilizando somente o MPI e utilizando OpenMP e MPI em conjunto. Foram analisados resultados obtidos utilizando o MGC (utilizando matriz normalizada e matriz sem normalização) e o MGCP (utilizando o Pré-condicionador de Jacobi sem normalização e o Pré-condicionador SSOR com a matriz normalizada). Todos os testes foram executados até que o critério de parada (tolerância atingida) fosse inferior a $\mathbf{10^{-10}}$.

6.3 Resultados Utilizando Somente o OpenMP

Todos os resultados nessa seção foram obtidos utilizando o esquema de armazenamento compacto. O tempo de execução do sistema (dado em segundos) utilizando os quatro métodos implementados está exposto na Tabela 4.

TABELA 4 – Tempo de execução (em segundos) utilizando OpenMP

Nº de nós	***SN, SP***	***CN ,SP***	***SN, PJ***	***CN,PS***
1	571,0670	351,9050	152,9710	127,1200
2	355,3190	268,3970	125,1570	117,3210
3	267,9910	209,2920	113,1260	114,8490
4	241,1560	185,9080	110,0070	111,8750

De acordo com a Tabela 4 percebe-se que normalizar a matriz nos métodos sem pré-condicionamento é extremamente necessário. Nessa Tabela percebe-se na abordagem utilizando o OpenMP que ocorre uma queda expressiva no tempo de execução quando se aumenta o número de núcleos de processamento em todos os casos testados. Observa-se também que o uso do pré-condicionador apresentou melhores resultados que nos métodos sem pré-condicionamento. Também é possível notar que a diferença entre o método utilizando o Pré-condicionador de Jacobi e o método utilizando o Pré-condicionador da SSOR é muito pequena, sendo que a diferença entre eles apresentou uma queda com o aumento dos núcleos de processamento. Informações adicionais são apresentadas na Figura 2, 3 e 4. Essas Figuras apresentam gráficos gerados a partir dos dados da Tabela 4 e apresentam informações complementares.

Analisando a Figura 2 percebe-se que o método apresentou o melhor *speed-up* quando foi utilizado o MGC sem normalização, enquanto que o pior foi obtido com o uso do Pré-condicionador da SSOR. A Figura 3 mostra que o melhor paralelismo foi obtido também pelo MGC sem normalização e o pior paralelismo foi apresentado pelo Pré-condicionador da SSOR. A Figura 4 mostra que a maior queda no tempo de execução ocorreu também no MGC sem normalização, embora esse método tenha apresentado o maior tempo de execução

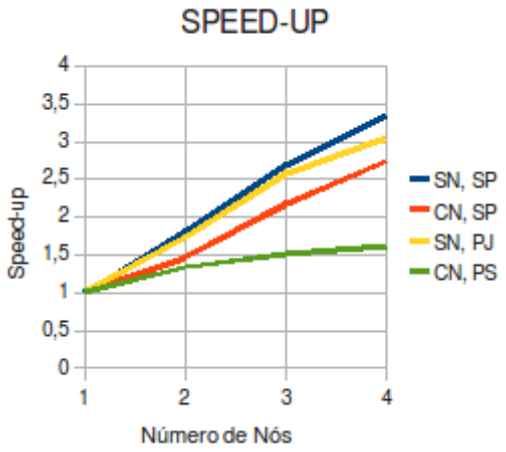

Figura 2 – Speed-up dos casos testados utilizando OpenMP somente

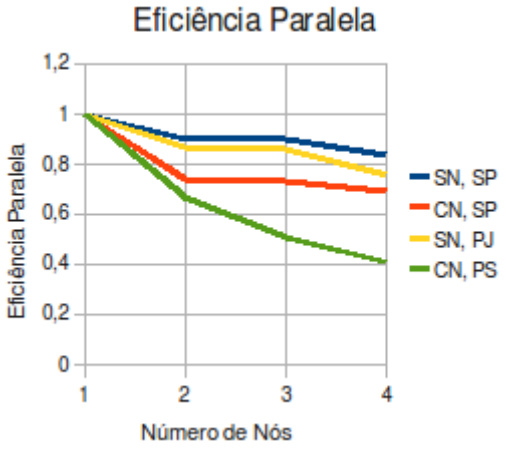

Figura 3 – Eficiência paralela dos métodos testados utilizando o OpenMP somente

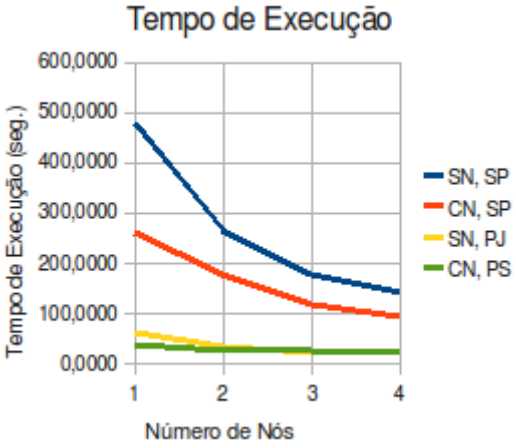

Figura 4 – Tempo de execução dos métodos estudados utilizando o OpenMP somente

6.4 Resultados Utilizando Somente o MPI

Os resultados exibidos nessa seção foram obtidos inicialmente utilizando o esquema de armazenamento compacto. Porém, como houve um aumento no tempo de execução com o aumento do número de processos, também são apresentados alguns resultados obtidos com a utilização do esquema de armazenagem densa para demonstrar que esse problema não é inerente à implementação. O tempo de execução de cada método implementado utilizando armazenamento compacto é exibido na Tabela 5.

TABELA 5 – Tempo de execução (em segundos) utilizando o armazenamento CRS e a abordagem MPI

Nº de nós	***SN, SP***	***CN ,SP***	***SN, PJ***	***CN,PS***
1	456,9690	335,6400	140,9600	134,7800
2	829,1840	822,2250	162,6370	161,8500
3	1151,2970	1117,5240	199,7900	210,2300

Observa-se que ocorre um aumento no tempo de execução decorrente do aumento do número de nós ao invés de uma diminuição. Mesmo assim é possível perceber que o uso de pré-condicionadores reduziu bastante o tempo de execução do sistema. Os gráficos exibidos nas Figuras 5, 6 e 7 foram gerados a partir dos dados fornecidos pela Tabela 5. Antes de analisar as informações contidas nesses gráficos, é importante analisar a Tabela 6 que apresenta os resultados do Pré-condicionador de Jacobi, obtidos utilizando armazenamento denso.

TABELA 6 – Tempo de execução (em segundos) do Pré-condicionador de Jacobi utilizando armazenamento denso com o MPI

Nº de nós	***SN, PJ***
1	33471,767
2	3892,022
3	2632,022

Os resultados exibidos nas Tabelas 5 e 6 foram obtidos utilizando praticamente o mesmo código. A única diferença é que o resultado exibido na Tabela 6 utilizou operações realizadas utilizando armazenamento denso. Analisando a Tabela 6 é possível perceber que a implementação utilizada é eficiente, pois proporcionou uma excelente redução de tempo. Percebe-se que o tempo de execução utilizando dois nós é quase nove vezes menor que o

tempo de execução em um único nó. Como a matriz armazenada nesse caso é muito grande e ocupa muita memória, o problema fica bem mais fácil de resolver quando é realizado por mais máquinas, sendo verificado, portanto essa queda tão acentuada de tempo.

Analisando os algoritmos implementados percebe-se também que independente do esquema de armazenamento utilizado, o tamanho e a quantidade de mensagens enviadas é o mesmo para cada caso. Portanto o que se percebe na Tabela 5 é que a redução no tempo de execução obtida pela utilização de mais núcleos de processamento não foi o suficiente para transpor o custo da comunicação. A eficiência da abordagem utilizando armazenamento compacto talvez seja evidenciada se testes forem feitos com sistemas com dimensões maiores, visto que nesse caso ela não sobressaiu.

Observando a Figura 5 percebe-se nessa abordagem o método que apresentou melhor *speed-up* foi método utilizando o Pré-condicionador de Jacobi, enquanto que o pior foi apresentado pelo MGC com a matriz normalizada. A Figura 6 confirma os dados exibidos pela Figura 5. Ela mostra que, para os métodos implementados, o método utilizando o Pré-condicionador de Jacobi foi o que apresentou o melhor paralelismo, sendo o pior apresentado pelo MGC com a matriz normalizada. A Figura 7 mostra que o aumento no tempo de execução foi menor nos métodos pré-condicionados e, novamente a diferença entre os dois métodos pré -condicionados foi muito pequena.

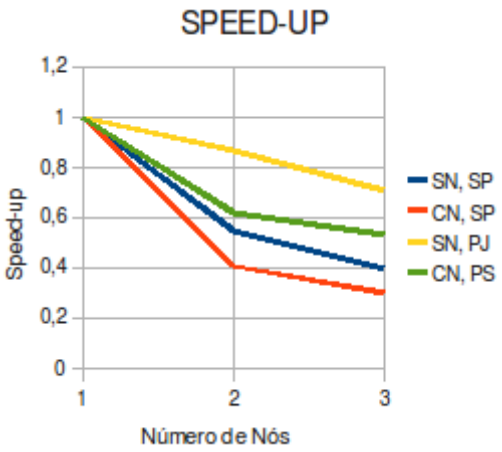

Figura 5 – Speed-up dos métodos implementados utilizando a abordagem MPI somente

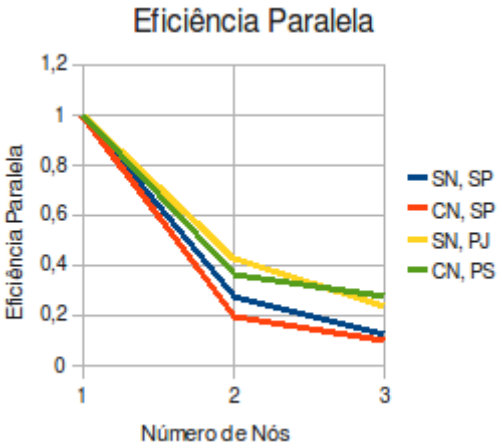

Figura 6 – Eficiência paralela dos métodos testados utilizando o MPI somente

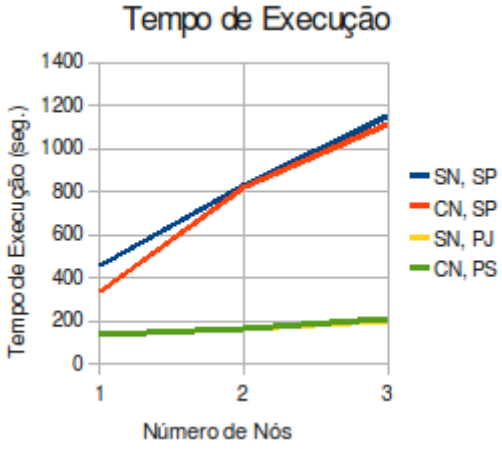

Figura 7 – Tempo de execução dos métodos estudados utilizando o MPI somente

6.5 Resultados Utilizando a Abordagem Híbrida (OpenMP +MPI)

Nesta etapa os resultados foram obtidos utilizando o esquema de armazenagem compacto e obtidos utilizando os quatro núcleos de processamento de cada máquina. O tempo de execução obtido em cada método esta exibido na Tabela 7. É possível perceber que semelhante ao MPI, ocorre um aumento do tempo de execução ao invés de uma diminuição.

TABELA 7 – Tempo de execução(em segundos) utilizando a abordagem híbrida

Nº de nós	***SN, SP***	***CN ,SP***	***SN, PJ***	***CN,PS***
1	243,3370	201,5560	118,0820	110,9070
2	761,9780	511,4300	162,1960	163,7800
3	1107,5350	778,5300	200,9870	215,7660

Novamente os métodos pré-condicionados apresentaram os menores tempos de execução e, analisando os métodos sem pré-condicionamento, novamente o método normalizado apresentou o menor tempo. As Figuras 8, 9, 10 exibem gráficos gerados a partir da Tabela 7. Porém antes de analisá-las é conveniente fazer um comparativo entre as Tabelas 5 e 7. Percebe-se que, de um modo geral a abordagem híbrida apresentou tempos de execução inferiores aos apresentados utilizando somente o MPI, sobretudo principalmente nos métodos sem pré-condicionamento. Para analisar a eficiência dessa abordagem, será necessário realizar testes com matrizes maiores para se perceber a sua eficiência.

Analisando as Figuras 8, 9 e 10 nota-se que elas apresenta informações muito semelhante às Figuras 5, 6 e 7. Nelas também pode-ser perceber o método que possui o melhor *speed-up* é o método utilizando o pré-condicionador de Jacobi e o pior é o MGC com a matriz normalizada, e que a melhor eficiência paralela é obtida pelo pré-condicionador de Jacobi e a pior pelo MGC com a matriz normalizada.

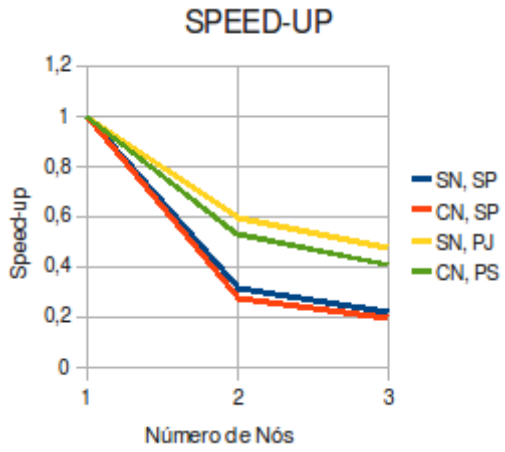

Figura 8 – Speed-up dos métodos implementados utilizando a abordagem Híbrida

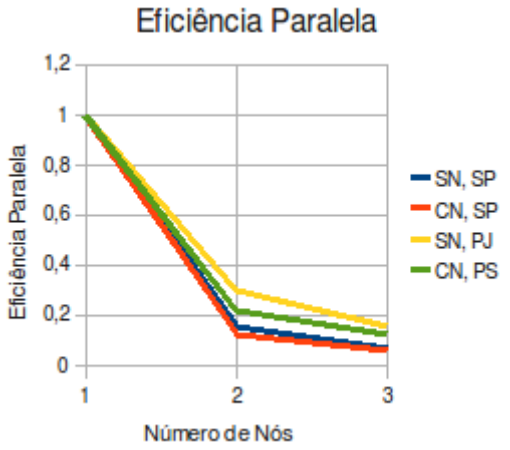

Figura 9 – Eficiência paralela dos métodos testados utilizando a abordagem Híbrida

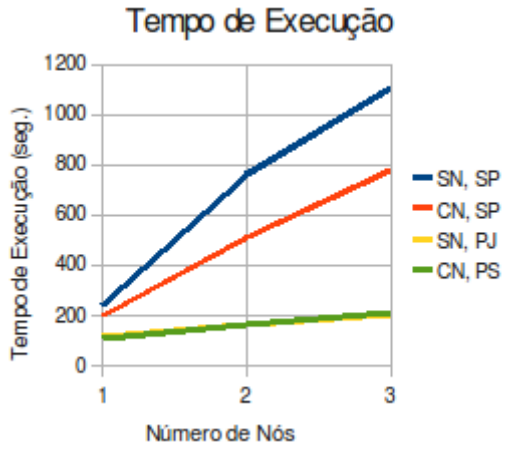

Figura 10 – Tempo de execução dos métodos estudados utilizando a abordagem Híbrida

7 CONCLUSÕES

A abordagem utilizando o OpenMP se mostrou bastante eficiente e, associado à sua fácil implementação pode ser uma excelente alternativa para execução em paralelo de sistema de pequena e média dimensão. Na abordagem OpenMP, embora não tenha apresentado o menor tempo de execução, o MGC sem normalização mostrou ser o método que apresentou o melhor paralelismo, visto que, conforme exposto nas Figuras 2 e 3, esse método apresentou o melhor *speed-up* e a melhor eficiência paralela. Analisando somente os métodos pré-condicionados, o Pré-condicionador de Jacobi mostrou ser o que possui a melhor abordagem paralela.

A abordagem utilizando o MPI mostrou ser eficaz em sistemas utilizando armazenamento denso, como pode ser visto na Tabela 6. No entanto o sistema executado não forneceu informações positivas em teste utilizando armazenamento compacto (Tabela 5). Nessa abordagem os métodos pré-condicionados apresentaram os melhores desempenhos, sendo a melhor implementação paralela apresentado pelo pré-condicionador de Jacobi.

A abordagem híbrida, embora tenha apresentado os mesmos problemas em relação à comunicação, apresentou na maioria dos casos tempo de execução inferior aos da abordagem utilizando o MPI somente.

A programação utilizando o MPI mostrou ser extremamente complexa e complicada, mas uma vez realizada, a criação de um código híbrido pode ser feita com muita facilidade, apenas inserindo as diretivas do OpenMP.

A utilização do esquema de armazenamento CRS para o sistema provocou uma expressiva redução no tempo de execução visto quando se faz um comparativo com a Tabela 5 e a Tabela 6. Comparando essas duas tabelas, percebe-se que a redução de tempo utilizando de 3 nós (máquinas) utilizando armazenamento compacto foi aproximadamente treze vezes menor que o tempo do mesmo problema utilizando armazenamento denso. Nas abordagens utilizando o MPI e a abordagem híbrida, o CRS apresentou redução no tempo de execução mas não o bastante para transpor os custos da comunicação.

A normalização foi de extrema importância, porém não é necessária quando se utiliza o pré-condicionador de Jacobi, pois para que o método seja efetivo deve haver variação nas entradas da diagonal principal da matriz.

O Pré-condicionador de Jacobi, sendo o mais simples mostrou-se extremamente eficaz no ambiente paralelo apresentando os melhores resultados nas três abordagens estudadas.

Se fosse considerado o número de iterações, o pré-condicionador da SSOR seria o melhor. Porém como o custo do pré-condicionador de Jacobi é bem inferior ao do pré-condicionador SSOR, a diferença entre eles foi muito pequena em todos os casos. Além do mais, deve-se considerar que o fator de sobre-relaxação do pré-condicionador SSOR foi dado *a priori* e, como seu cálculo não é uma tarefa trivial, provavelmente seu custo computacional ficará ainda maior quando essa rotina for inserida.

8 TRABALHOS FUTUROS

Visando a melhoria do programa desenvolvido, algumas rotinas podem ser estudas e implementadas:

- Cálculo do fator de sobre-relaxação;
  Nos testes realizados utilizando o Pré-condicionador SSOR o fator de sobre-relaxação (ω) foi dado *a priori* devido ao custo computacional dos métodos que estimam esse valor. Porém o método ficará mais completo e poderá ser utilizado com facilidade em outros problemas se um método eficiente para o cálculo desse fator for inserido no método implementado.
- Ordenação de matrizes;
  Percebeu-se durante os testes que as matrizes testadas apresentavam a banda muito larga devido a um espaçamento muito grande dos elementos de cada linha. A ordenação dessas matrizes poderá favorecer o uso dos métodos diretos para obtenção da solução dos sistemas e favorecer também a utilização dos pré-condicionadores gerados pelo processo de Fatoração incompleta na obtenção da solução iterativa;
- Solução de sistemas utilizando métodos diretos e análise de sua eficiência;
  Pela natureza do procedimento Babuska, os métodos diretos parecem ser os mais indicados. Porém para que esses métodos sejam mais efetivos, é necessário que a banda matriz de rigidez seja reduzido para que esses métodos sejam testados e analisados.
- Rotinas dos pré-condicionadores de Fatoração Incompleta;
  Os pré-condicionadores gerados por meio dos processos de fatoração são geralmente os mais utilizados e mais efetivos. Porém a paralelização eficiente desses métodos não é uma tarefa trivial
- Outros esquemas de armazenamento;
  Nesse trabalho foi estudado o CRS por ser o esquema de armazenamento mais utilizado e também pelo fato desse esquema não armazenar nenhum elemento nulo. Porém em processos que envolvem fatoração de matrizes, o tamanho da matriz não pode ser antecipadamente previsto, o que desfavorece o uso desse esquema. Portanto o estudo de outros esquemas de armazenamento é necessário para que possa utilizar os processos de fatoração no cálculo da solução do sistema.
- Influência do enriquecimento;

O enriquecimento das funções de forma do MEFG é obtido por meio do produto da PU por funções de enriquecimento. Porém o número de condicionamento da matriz de rigidez gerada é dependente do grau de enriquecimento e, quanto maior o grau da função de enriquecimento, maior o número de condicionamento da matriz. Torna-se então necessário um estudo sobre a influência do nível do enriquecimento na escolha do pré-condicionador utilizado na solução iterativa.

- Paralelização do domínio de elementos finitos;
  A solução do sistema linear gerado (processamento) é a tarefa de maior custo computacional no estudo de elementos finitos. Porém as demais etapas (pré-processamento e pós-processamento) também podem ser paralelizadas reduzindo ainda mais o tempo de simulação.

REFERÊNCIAS BIBLIOGRÁFICAS

ALLAIRE, G. **Numerical Analysis and Optimization**. New York: Oxford University Press, 2007.

AXELSSON, O.; BARKER, V.A. **Finite Element Solution of Boundary Value Problems**: Theory and Computation. Philadelphia: Society for Industrial and Applied Mathematics, 2001.

BABUSKA, I.; BANERJEE, U. OSBORN, J. E. **Generalized Finite Element Methods:** Main Ideas, Results, and Perspective. Int. J. Comput. Methods 1 (1) 67–103. 2004

BARRETT, R.; BERRY, M.; CHAN, T.F.; DEMMEL, J.; DONATO J.M.; DONGARRA, J.; EIJKHOUT, V.; POZO, R.; ROMINE, C.; VOST, H. van der. **Templates for Solution of Linear Systems**: Building Blocks for Iterative Methods. 2nd. Philadelphia: Society for Industrial and Applied Mathematics, 1994.

BATHE, K.J. **Finite Element Procedures**. New Jersey: Prentice-Hall, 1996.

BELYTSCHKO, T.; GRACIE, R.; VENTURA, G. **A Review of Extended / Generalized Finite Element Methods for Material Modelling** *Modelling and Simulation in Materials Science and Engineering*, Vol. 17, No. 4. 2009.

BELYTSCHKO, T.; LU, Y.; GU, L. **Element-free Galerkin Methods**. *International Journal for Numerical Methods in Engineering*, v. 37, p. 229–256, 1994.

BENZI, M. **Preconditioning Techniques for Large Linear Systems:** A Survey. Journal of Computational Physics 182, 418-477. 2002

BRAESS, D. **Finite Element:** Theory, Fast Solvers, and Applications in Elasticity Theory. 3rd ed. New York: Cambridge University Press, 2007.

BURDEN, R. L.; FAIRES, J. D. **Análise Numérica**. 8ª ed. São Paulo: Cengage Learning, 2008.

CHAPMAN, BARBARA; JOST, GABRIELE.; PAS, Ruud van der. **Using OpenMP**: Portable Shared Memory Parallel Programming. Massachusetts: MIT Press, 2008.

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; MCDONALD, J.; MENON, R. **Parallel Programming in OpenMP**. San Diego: Academic Press, 2001.

DEMMEL, J. **Applied Numerical Linear Algebra.** Massachusetts: MIT Press, 1996.

DONGARRA, J.J.; DUFF, L.S.; SORRENSEN, D.C.; VORST, H.A. **Numerical Linear Algebra for High-Performance Computers.** Philadelphia: SIAM, 1998.

DUARTE, C.A; BABUSKA, I.; ODEN, J.D. **Generalized Finite Element Methods for Three Dimensional Structural Mechanics Problems**. *Computers & Structures*, v. 77, n. 2, p. 215–232, 2000.

DUARTE, C. A.; ODEN, J. T. **H-p Clouds - An h-p Meshless Method.** In: *Numerical Methods for Partial Differential Equations*. [S.l.]: John Wiley & Sons, Inc., 1996. p. 1–34.

EL-REWINI, H.;ABD-EL-BARR, M. **Advanced Computer Architecture And Parallel Processing.** New Jersey: John Wiley & Sons, 2005.

ELMAN, H.C.; SILVESTRE, D.J; WATHEN, A.J. **Finite Elements and Fast Iterative Solvers**: with Applications in Incompressible Fluid Dynamics. New York: Oxford University Press, 2006.

FENNER, R.T.; **Finite Element Methods for Engineers.** London: Imperial College Press, 1996.

GOCKENBACK, M.S. **Understanding and Implementing the Finite Element Method**. Philadelphia: Society for Industrial and Applied Mathematics, 2006.

GOLUB, G.H.; LOAN ,C.F. **Matrix Computations**. 3th ed. Maryland : Johns Hopkins University Press, 1996.

GROPP, W.;LUSK, E.; THAKUR, R. **Using MPI-2**: Advanced Features of the Message-Passing Interface. Massachusetts: Mit Press, 1999.

KARNIADAKIS, G. E.; KIRBY II, R. M. **Parallel Scientific Computing in C++ and MPI**: A Seamless Approach to Parallel Algorithms and Their Implementation. Cambridge: Cambridge University Press, 2003.

KHOEI, A.R.; BIABANAKI, O.; ANAHID, M. **Extended Finite Element Method for Three–Dimensional Large Plasticity Deformations.** Computer Methods in Applied Mechanics and Engineering 2007

KIM, D.J.; DUARTE, C.A.; PROENÇA,S.P.; **Generalized Finite Element Method with global-local enrichments for nonlinear fracture analysis.** *Mechanics of Solids in Brazil* pp. 317- 330, 2009

KNABNER, P.; ANGERMANN, L. **Numerical Methods for Elliptic and Parabolic Partial Differential Equations.** New York: Springer-Verlag, 2003.

KSHEMKALYANI, A.D., SINGHAL,M. **Distributed Computing:** Principles, Algorithms and Systems. New York: Cambridge University Press, 2008.

LAX, P. **Linear Algebra**. New York: John Wiley & Sons, 1997.

MENDONÇA, P.T.R.; FANCELLO, E.A. **Análise Elasto-Plástica 3D pelo MEF Generalizado.** *Mecánica Computational* Vol. XXI , pp. 1187-1202, 2002

MEYER, C.D. **Matrix Analysis and Applied Linear Algebra**. Philadelphia: Society for Industrial and Applied Mathematics, 2000.

ODEN, J. T.; REDDY, J.N. **An Introduction to the Mathematical Theory of Finite Elements, Pure and Applied Mathematics**. New York: John Willey & Sons, 1976.

OÑATE, E. **Structural Analysis with the Finite Element Method** - Linear Statics. Volume 1. Basis and Solids, Barcelona: CIMNE, 2009.

RAO, S.S. **The Finite Element Method in Engineering.** Paris: Elsevier Science & Technology Books, 2004

SHEWCHUK, J. R. **An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.** Carnegie Mellon University, Pittsburgh, PA, 1994.

SNIR, M.;OTTO, S.;HUSS-LEDERMAN, S.;WALKER, D.; DONGARRA, J. **MPI:** The Complete Reference. 2nd ed. Massachusetts: Mit Press, 1999. v.1, The MPI Core.

STERLING, T. **Beowulf Cluster Computing with Linux**, Massachusetts: Mit Press, 2001.

STROUBOULIS, T.; BABUSKA, I.; COPPS, K. **The design and analysis of the generalized finite element method**, Computer Methods in Applied Mechanics and Engineering, 181(1-3), p. 43-69. 2000.

SUKUMAR, N.; MOES, N.; MORAN, B.; BELYTSCHKO, T. **Extended Finite Element Method for Three-dimensional Crack Modelling**. *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*, v. 48:1549-1570. 2000.

TREFETHEN, L.N.; BAU, D. **Numerical Linear Algebra**. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

VOLAKIS, J.L.; CHATTERJEE, A.; KEMPEL, L.C. **Finite Element Method for Electromagnetics**: Antennas, Microwave Circuits and Scattering Applications. New York: Institute of Electrical and Electronical Engineers, 1998

VRENIOS, ALEX. **Linux Cluster Architecture**. Indianapolis: Sams Publishing, 2002.

WRIGGERS, P. **NonLinear Finite element Methods.** Berlin: Springer-Verlag, 2008

APÊNDICE A – DEFINIÇÕES

Definição 1 A **transposta** de uma matriz $A = a[i,j],\ nXm$ é a matriz $A^T = a[j,i],\ mXn$.

Definição 2 Uma matriz **quadrada** possui o mesmo número de linhas e colunas.

Definição 3 Uma matriz quadrada A é chamada de **simétrica** se $A = A^T$.

Definição 4 Uma matriz A é **definida positiva** se $x^T Ax > 0$, para todo vetor $x \neq 0$.

Definição 5 Uma matriz A é **semi-definida positiva** se $x^T Ax \geq 0$, para todo vetor $x \neq 0$.

Definição 6 Uma matriz A é **indefinida** se $x^T Ax$ assume valores positivos ou negativos para vários valores de x.

Definição 7 Uma matriz é **densa** quando possui poucos elementos nulos e é **esparsa** quando possui muitos elementos nulos.

Definição 8 **Espaço vetorial real** é um conjunto V não vazio com duas operações: soma de vetores e multiplicação de vetor por escalar e, um elemento desse espaço é denominado **vetor**.

Definição 9 Dois vetores x e y de um espaço vetorial V são **ortogonais** se seu produto escalar (produto interno) for nulo, ou seja, $< x, y > = 0$.

APÊNDICE B – IMPLEMENTAÇÃO DE UM CLUSTER BEOWULF

Esse pequeno tutorial auxilia a configuração de um *cluster beowulf* em um sistema composto por máquinas com o sistema operacional Linux Ubuntu instalado. A configuração de um *cluster Beowulf* pode ser realizada de várias maneiras diferentes. Embora existam várias configurações, as etapas básicas são:

1 Configuração da rede

No terminal, logado como *root*, edite o arquivo */etc/hosts*. Esse arquivo contém os IPs e os *hostname* dos computadores que compõem o cluster. Esse procedimento deve ser realizado em todas as máquinas. Para editar esse arquivo, digite no terminal:

```
sudo gedit /etc/hosts
```

No arquivo que se abre, digite os 'IPs' <espaço> 'hostname' das máquinas do *cluster*, por exemplo

```
192.168.0.1 master
192.168.0.2 slave1
192.168.0.3 slave2
```

A máquina *master* corresponde à máquina portadora dos dados compartilhados. Durante a execução, a máquina que inicia o processamento de dados também é denominada *master*, embora corresponda frequentemente à mesma máquina portadora dos dados.

Edite também o arquivo */etc/hostname*. Esse arquivo contém o nome da máquina. Esse procedimento também é realizado em todas as máquinas. Digite no terminal:

```
sudo gedit /etc/hostname
```

No arquivo que se abre, digite o mesmo da máquina. Na máquina *master* ficará da seguinte forma

```
master
```

Em cada máquina esse arquivo conterá somente seu próprio nome.

2 Instalando os aplicativos necessários

É necessário a instalação de 5 pacotes, descritos a seguir:

- NFS (Network File System) → é utilizado, sobretudo, para prover compartilhamento de pastas via rede de comunicação;
- SSH Server →é utilizado para prover comunicação entre os nós;
- Build-essential → contém dentre outros pacotes, o compilador gcc (compilador c);
- Gfortran → compilador Fortran;
- MPICH2 →contém os pacotes do MPI.

Para instalar esses pacotes, digite no terminal um comando por vez (é necessário conexão com a WEB e deve ser feito em todas as máquinas)

```
sudo apt-get install nfs-kernel-server
sudo apt-get install opensshserver
sudo apt-get install build-essential
sudo apt-get install gfortran
sudo apt-get install mpich2
```

O pacote mpich2 pode ser instalado somente na máquina *master*, porem instalá-lo em todas as máquinas permite que o processamento possa ser solicitado por qualquer máquina e não somente pela máquina denominada *master*. Caso se opte pela instalação desse pacote somente no *master*, a pasta da instalação deve ser compartilhada na etapa 3.2. O pacote mpich2 também pode ser baixado e instalado manualmente o que dá uma mais flexibilidade na instalação, porém instalá-lo via terminal é bem mais fácil.

3 Outras configurações

3.1 Definindo um usuário para o ambiente paralelo

É conveniente (não obrigatório) a criação de um usuário para a utilização de um *cluster*. Para criar um usuário, digite no terminal o seguinte comando (‘nome_do_usuario’ deve ser alterado para o nome desejado):

```
sudo chown ‘nome_do_usuario’
```

3.2 Compartilhando pastas do master

Essa etapa exige configuração diferente na máquina *master* e nas maquinas *slave*. No *master* é necessário informar qual pasta será compartilhada, via NFS, com as demais máquinas. As pastas compartilhadas por meio do NFS são inseridas no arquivo */etc/exports*. Como nesse tutorial o pacote mpich2 é instalado em todas as máquinas, somente a pasta */home/’nome_do_usuario’* será compartilhada. Portanto no terminal da máquina *master* digite

```
sudo echo /home/'nome_do_usuario' *(rw,sync) >> /etc/exports
```

Nas máquinas *slave* é necessário montar as pasta de acesso aos dados compartilhado pelo *master*. A primeira alternativa é montar manualmente digitando no terminal o comando das máquinas *slave*

```
sudo mount master:/home/'nome_do_usuario' / home/'nome_do_usuario'
```

Porém essa montagem é momentânea, ou seja, só é valida para a sessão atual e, portanto, deixara de existir quando a máquina for reiniciada. Para que a montagem seja 'permanente', ou seja, montagem automática após cada login, as pastas montadas devem ser informadas no arquivo /etc/fstab. Portanto no terminal das máquinas *slave* digitar:

```
sudo gedit /etc/fstab
```

No arquivo que se abre inserir a linha:

```
master:/home/'nome_do_usuario' / home/'nome_do_usuario' none defaults 0 0
```

Para montar uma pasta compartilhada via NFS, é preciso que ela exista na máquina de destino, por isso a necessidade de se criar em cada máquina um mesmo usuário. A pasta de destino não é eliminada, ela somente é substituída se a montagem puder ser feita.

3.3 Configurando uma comunicação sem senha com o SSH

Em um cluster a comunicação deve ser feita de modo seguro, porém é necessário que ela seja feita sem a necessidade de se informar uma senha a cada solicitação. Portanto crie uma senha no *master* digitando no terminal

```
sudo ssh-keygen -t das
```

Em seguida informe para o computador que essa senha será utilizada durante o processo de comunicação. Para isso digite no terminal os dois comandos abaixo

```
sudo cd .ssh
sudo cat id_dsa.pub >> authorized_keys
```

Como a senha gerada é armazenada na pasta */home* e esta pasta é compartilhada por todas as máquinas, essa configuração não precisa ser feita em todas as máquinas. Para testar se a comunicação ocorre sem solicitação de senha, digite no terminal

```
ssh ‘nome ou IP da máquina com quem se deseja comunicar’ date
```

No terminal deve aparecer a data da máquina solicitada sem ser necessário informar uma senha. No primeiro acesso aparecerá uma mensagem informando que a máquina está sendo inserida na lista de máquinas confiáveis.

3.4 Configurando o MPD

O MPD é responsável pelo gerenciamento do MPI. Para configurá-lo, basta criar um arquivo no diretório */home* do usuário. Esse arquivo deve conter o nome das máquinas pertencentes ao cluster. Para criar o arquivo digite

```
sudo /home/'nome_do_usuario'/mpd.hosts
```

Edite o arquivo, digitando o nome das máquinas, como em

```
master
slave1
slave2
```

Para finalizar essa configuração digite no terminal os dois comandos seguintes. Eles definem uma senha para o MPD e alteram a permissão de acesso a esse arquivo.

```
echo secretword="crie_uma_senha" >> ~/.mpd.conf
chmod 600 .mpd.conf
```

4 Testando o MPI

Para testar se o sistema está configurado corretamente, digite os três comandos seguintes, um após o outro. Como resultado, deve aparecer na tela o caminho onde se encontra cada arquivo.

```
which mpd
which mpiexec
which mpirun
```

Se tudo estiver ok, inicie o MPD digitando no terminal o comando

```
mpdboot -n 'numero_de_máquinas'
```

Nessa etapa podem ocorrer alguns erros. Seguem alguns

Mensagem de Erro	Motivo
handle_mpd_output 415:	O host não está cadastrado no arquivo /etc/hosts da máquina requisitante
handle_mpd_output 407:	O host solicitante não esta cadastrado no arquivo /etc/hosts da máquina solicitada
handle_mpd_output 420:	Ocorre se o host solicitado não estiver cadastrado, ou senão for uma máquina confiável (não incluída no arquivo de máquinas confiáveis ou se a máquina estiver desligada ou fora da rede.

Se não acorre nenhum erro digite o comando abaixo. Ele imprime no terminal o nome das máquinas conectadas no momento.

```
mpdtrace
```

Para finalizar compile e execute um programa escrito com o MPI. Para compilar um código escrito utilizando o MPI, digite

```
mpif90 -o <nome para o executável> <arquivo código fonte Fortran>
```

Para executar o arquivo, digitar

```
mpirun -np 'numero_processo_a_serem_criado' ./'nome_do_executavel'
```

Ou

```
mpiexec -np 'numero_processo_a_serem_criado' ./'nome_do_executavel'
```

APÊNDICE C – TUTORIAL OPENMP

Esse pequeno tutorial apresenta as diretivas e funções do OpenMP que foram utilizados nesse trabalho e, tudo baseado no ambiente Fortran, mais precisamente a versão Fortran 95. Antes da inserção das diretivas é necessário incluir no cabeçalho do programa as biblioteca do OpenMP (omp_lib). Um exemplo pode ser visto no segmento abaixo.

```
program
    use omp_lib
implicit none
    !Bloco de código
end program
```

1 Diretivas Utilizadas

Uma diretiva OpenMP é composta por três partes: O identificador, a diretiva e os atributos (cláusulas) e possui a seguinte estrutura

No Fortran 95, o único identificador disponível é o ! $***OMP***. Todo conteúdo a ser paralelizado está inserido entre esse identificador é reconhecido somente pelo compilador do OpenMP e ignorado pelos demais compiladores.

As diretivas aparecem no código em pares, uma para iniciar e outro para finalizar o trecho, como aparece no trecho abaixo

```
!$OMP diretiva [atributos]
    !Código Fortran
!$OMP end diretiva
```

As três diretivas utilizadas nesse trabalho foram

Diretiva	Descrição
PARALLEL	Identifica um código que será executado em paralelo, ou seja, executado por múltiplos *threads*.
DO	Compartilha um "loop" entre um grupo de threads.
CRITICAL	Define uma região que deve ser executada somente por um *thread* por vez

As diretivas podem aparecer uma em cada linha ou combinadas na mesma linha. Os seguimentos abaixo apresentam o mesmo resultado

```
!$OMP PARALLEL
!$OMP DO
      !Trecho de código
!$OMP END DO
!$OMP END PARALLEL

ou

!$OMP PARALLEL DO
    !bloco de código
!$OMP END PARALLEL DO
```

As diretivas podem ou não possuir atributos. A diretiva CRITICAL, por exemplo, só possui um atributo (nome) e mesmo assim não é obrigatório. As outras duas diretivas citadas possuem dentre outras as seguintes cláusulas

Cláusula	Descrição
PRIVATE(var1, var2, ..., varn)	Indica que todas as variáveis listadas são privativas a cada *thread*
SHARED(var1, var2, ..., varn)	Indica que todas as variáveis listadas são compartilhadas por todos os *threads*.

DEFAULT (PRIVATE | SHARED | NONE)	Define qual será o atributo padrão para todas as variáveis.
REDUCTION *(operador: lista)*	Efetua uma operação de redução a um conjunto de variáveis escalares compartilhadas.

A Biblioteca do OpenMP apresenta também algumas funções e rotinas para acesso e manipulação das threads. São elas

Função	**Descrição**
OMP_SET_NUM_THREADS(p)	Define o número (definido pela variável inteira *p*) de *threads* que executarão uma região paralela. Deve ser executada antes da região paralela, na parte serial do código.
OMP_GET_NUM_THREADS()	Retorna o número de *threads* que executarão a região paralela. Deve ser chamada na região paralela.
OMP_GET_MAX_THREADS()	Retorna o número máximo de *threads* que um programa pode utilizar.
OMP_GET_THREAD_NUM()	Retorna o ID (identificação) do *thread* que está executando um determinado trecho.
OMP_GET_NUM_PROCS()	Retorna o número de processadores disponível para a execução do problema.
OMP_GET_NUM_THREADS()	Retorna o número de *threads* que está sendo executado.

2 – Compilando um Código OpenMP

O pacote do OpenMP já vem instalado nas principais distribuições do sistema operacional Linux. Para compilar um código contendo diretivas do OpenMP utilizando o compilador padrão do Linux (gfortran), basta digitar o seguinte comando na tela do terminal:

```
gfortran -o <nome para o executável> <arquivo código fonte Fortran> -fopenmp
```

Para compilar um código contendo instruções OpenMP e MPI, utilizando o compilador fornecido no pacote do MPI, o comando utilizado é

```
mpif90 -o <nome para o executável> <arquivo código fonte Fortran>  -fopenmp
```

APÊNDICE D – TUTORIAL MPI

Antes de utilizar qualquer comando MPI, é necessário incluir no cabeçalho do programa a biblioteca do MPI. No Fortran isso é realizado inserido a seguinte linha:

```
include "mpif.h"
```

TABELA 1 – Tipo de dados

MPI	Fortran
MPI_DOUBLE_PRECISION	double precision
MPI_INTEGER	integer
MPI_REAL	real

Para tabelas abaixo, considerar as seguintes variáveis

Variável	Descrição
erro	Indica o sucesso ou não de uma operação
comm	Representa um comunicador MPI
rank	ID de um processo
processos	Número de processos
nome	Hostname da máquina
info	Tamanho do nome de uma máquina
buf	Conjunto de dados a enviar ou a receber
quant	Quantidade de dados a enviar ou receber
tipo	Tipo de dado MPI que está sendo enviado ou recebido
origem	Indica o rank de quem enviou uma mensagem
dest	Indica o rank de quem receberá a mensagem
tag	Identificador de uma mensagem
bufe	Conjunto de dados a ser enviados
bufr	Conjunto de dados a ser recebido

TABELA 2 – Constantes

Constante	**Descrição**
MPI_ANY_TAG	Identificador genérico
MPI_COMM_WORLD	Comunicador pré-definido
MPI_ANY_SOURCE	Origem genérica. Indica que qualquer origem é aceitável
MPI_SUM	Representa uma operação de soma em uma operação de redução

TABELA 3 – Alguns rotinas básicas

Comando	**Descrição**
MPI_INIT(erro)	Sub-rotina que define e inicia o ambiente processamento paralelo. É a primeira instrução que deve ser informada
MPI_FINALIZE(erro)	Sub-rotina que finaliza o ambiente paralelo. Logicamente é a ultima instrução MPI executada
MPI_COMM_RANK(comm, rank, erro)	Sub-rotina que retorna em ‘rank’ o identificador de um processo dentro de um grupo de processos. O valor de rank varia de 0 a p – 1, onde p é o numero de processos. A constante MPI_PROC_NULL representa um processo nulo, portanto seu valor corresponde a -1.
MPI_COMM_SIZE(comm, processos, erro)	Sub-rotina que retorna em ‘processos’ o numero de processos dentro de um grupo de processos.
MPI_GET_PROCESSOR_NAME(nome, erro, info)	Sub-rotina que retorna em ‘nome’ o ‘hostname’ (nome) de um processador.

TABELA 4 – Rotinas de comunicação *point-to-point*

Comando	**Descrição**
MPI_SEND(buf, quant, tipo, dest, tag, comm, erro)	Sub-rotina utilizada para envio de dados de modo bloqueante, ou seja, só retorna quando um dado for enviado.
MPI_RECV(buf, quant, tipo, origem, tag, comm, iStatus, erro)	Sub-rotina utilizada para recebimento de dados de modo bloqueante, ou seja, só retorna quando um dado for recebido.
MPI_ISEND(buf, quant, tipo, dest, tag,	Sub-rotina para envio de dados não

comm, erro)	bloqueantes.
MPI_IRECV(buf, quant, tipo, origem, tag, comm, request, erro)	Sub-rotina para recebimento de dados não bloqueantes.

TABELA 5 – Rotinas de Comunicação coletiva

Comando	Descrição
MPI_REDUCE(suma, val, 1, tipo, tipo_red, 0, comm, erro)	Aplica uma operação de redução em todos os processos de um grupo
MPI_BCAST(val,1,tipo,0,comm,erro)	Envia um mesmo dado para todos os processos pertencentes a um grupo de processos.
MPI_GATHERV(aux, tam_local, tipo, x, tamanhos, limites, tipo, 0, comm,erro)	Recebe um array de dados distribuído.
MPI_SCATTERV(w, tamanhos, limites, tipo, aux_w, tam_local, tipo, 0, comm, erro)	Distribui dados de um array para todos os processos

A lista detalhada dos comandos MPI pode ser encontrada em HTTP://www.mcs.anl.gov/research/projects/mpi/www/www3/