UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Paralelização de Métodos de Resolução de Sistemas Lineares Esparsos com o DECK em um *Cluster* de PCs

por
ANA PAULA CANAL

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Dr. Tiarajú Asmuz Diverio
Orientador

Porto Alegre, dezembro de 2000.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Canal, Ana Paula
Paralelização de Métodos de Resolução de Sistemas Lineares Esparsos com o DECK em um *Cluster* de PCs / por Ana Paula Canal. – Porto Alegre: PPGC da UFRGS, 2000.
117 p. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2000. Orientador: Diverio, Tiarajú Asmuz.

1. Sistemas Lineares Esparsos. 2 . Paralelização de Algoritmos. 3. *Clusters*. 4. Matemática da Computação. I. Diverio, Tiarajú Asmuz. II. Título.

Agradecimentos

Agradeço ao CNPQ pelo auxílio financeiro destinado à realização dessa pesquisa.

Agradeço ao Prof. Tiarajú pela confiança depositada em mim, ao ter me aceito no mestrado, pelo empenho em conceder-me a bolsa de pesquisa e pelas orientações dadas.

Ao Prof. Alexandre Caríssimi pelo esclarecimento de dúvidas relacionadas a threads.

Ao Rafael Ávila, Marcos Barreto e demais integrantes do DECK Team pelo suporte na utilização do DECK e do Cluster e o empenho em solucionar os problemas.

Aos demais professores do Instituto de Informática pelos ensinamentos transmitidos e aos funcionários pela constante disponibilidade e presteza.

Aos colegas e amigos Cadinho e Rogério pelas valiosas contribuições, sugestões e críticas nesse trabalho. Obrigada pelo apoio e disponibilidade em me auxiliar sempre que precisei.

Aos amigos Cláudia, Rogério, Maristela, Cadinho, Cathy, Marilton e Júlio pelo apoio e momentos de lazer.

Ao Dhino (in memorian) pelo apoio e encorajamento em prosseguir a caminhada.

Aos meus pais Anselmo e Catarina e ao meu irmão Ivan Paulo por estarem sempre presentes, apesar da distância, com seu carinho, incentivo e compreensão.

Ao meu noivo Marcos pelo companheirismo, compreensão, apoio, paciência e ... por ter enxugado minhas lágrimas em muitos momentos dessa trajetória.

Sumário

Lista de Abreviaturas

ADI	*Alternating Direction Implicit*
AT	Algoritmo de Thomas
C.C.	Condições de Contorno
C.I.	Condições Iniciais
CD	*Conjugate Directions*
CG	*Conjugate Gradient*
CPU	*Central Processing Unit*
DECK	*Distributed Executive Communication Kernel*
EDO	Equação Diferencial Ordinária
EDP	Equação Diferencial Parcial
GMC	Grupo de Matemática da Computação
GMRES	*Generalized Minimum Residual*
GPPD	Grupo de Processamento Paralelo e Distribuído
IML++	*Iterative Methods Library*
LAN	*Local Area Network*
LAPACK	*Linear Algebra Package*
LINPACK	*Linear Package*
LU	*Lower/Upper*
MB	Megabytes
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MPI	*Message Passing Interface*
MPP	*Massively Parallel Processors*
PC	*Personal Computer*
PCAM	*Partitioning, Communication, Aglomeration and Mapping*
PCG	*Preconditioned Conjugate Gradient*
PIM	*Parallel Iterative Methods*
PVM	*Parallel Virtual Machine*
RAM	*Random Access Memory*
ScaLAPACK	*Scalable Linear Algebra Package*
SD	*Steepest Descent*
SELA	Sistema de Equações Lineares Algébricas
SIMD	*Single Instruction, Multiple Data*

SISD	*Single Instruction, Single Data*
SOR	*Sucessive Over Relaxation*
SPD	Simétrica, positiva e definida
SPMD	*Single Program, Multiple Data*

Lista de Símbolos

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
λ	Autovalor
Σ	Somatório
=	Igual
<	Menor
>	Maior
<>	Diferente
$\geq$	Maior ou igual
$\leq$	Menor ou igual
$\| . \|$	Norma
A	Matriz
a_{ij}	Elemento da matriz A na linha i e coluna j
A^{-1}	Matriz Inversa
A^{T}	Matriz Transposta
Det(A)	Determinante da matriz A
diag(A)	Diagonal da matriz A
I	Matriz Identidade
v^{i}	Vetor *v* na i-ésima iteração
v	Vetor
v_i	i-ésimo elemento do vetor *v*
$f_{(x)}$	Função
$f'_{(x)}$	Gradiente
$y^{n}_{(x)}$	Derivada de ordem *n* da função $y_{(x)}$
∂	Derivada Parcial

Lista de Figuras

Lista de Equações

Lista de Tabelas

Resumo

O objetivo desta dissertação é a paralelização e a avaliação do desempenho de alguns métodos de resolução de sistemas lineares esparsos. O DECK foi utilizado para implementação dos métodos em um *cluster* de PCs.

A presente pesquisa é motivada pela vasta utilização de Sistemas de Equações Lineares em várias áreas científicas, especialmente, na modelagem de fenômenos físicos através de Equações Diferenciais Parciais (EDPs). Nessa área, têm sido desenvolvidas pesquisas pelo GMC-PAD – Grupo de Matemática da Computação e Processamento de Alto Desempenho da UFRGS, para as quais esse trabalho vem contribuindo.

Outro fator de motivação para a realização dessa pesquisa é a disponibilidade de um *cluster* de PCs no Instituto de Informática e do ambiente de programação paralela DECK – *Distributed Execution and Communication Kernel*. O DECK possibilita a programação em ambientes paralelos com memória distribuída e/ou compartilhada. Ele está sendo desenvolvido pelo grupo de pesquisas GPPD – Grupo de Processamento Paralelo e Distribuído e com a paralelização dos métodos, nesse ambiente, objetiva-se também validar seu funcionamento e avaliar seu potencial e seu desempenho.

Os sistemas lineares originados pela discretização de EDPs têm, em geral, como características a esparsidade e a numerosa quantidade de incógnitas. Devido ao porte dos sistemas, para a resolução é necessária grande quantidade de memória e velocidade de processamento, característicos de computações de alto desempenho.

Dois métodos de resolução foram estudados e paralelizados, um da classe dos métodos diretos, o Algoritmo de Thomas e outro da classe dos iterativos, o Gradiente Conjugado. A forma de paralelizar um método é completamente diferente do outro. Isso porque o método iterativo é formado por operações básicas de álgebra linear, e o método direto é formado por operações elementares entre linhas e colunas da matriz dos coeficientes do sistema linear. Isso permitiu a investigação e experimentação de formas distintas de paralelismo.

Do método do Gradiente Conjugado, foram feitas a versão sem pré-condicionamento e versões pré-condicionadas com o pré-condicionador Diagonal e com o pré-condicionador Polinomial. Do Algoritmo de Thomas, devido a sua formulação, somente a versão básica foi feita.

Após a paralelização dos métodos de resolução, avaliou-se o desempenho dos algoritmos paralelos no *cluster*, através da realização de medidas do tempo de execução e foram calculados o *speedup* e a eficiência. As medidas empíricas foram realizadas com variações na ordem dos sistemas resolvidos e no número de nodos utilizados do *cluster*. Essa avaliação também envolveu a comparação entre as complexidades dos algoritmos seqüenciais e a complexidade dos algoritmos paralelos dos métodos.

Esta pesquisa demonstra o desempenho de métodos de resolução de sistemas lineares esparsos em um ambiente de alto desempenho, bem como as potencialidades do DECK. Aplicações que envolvam a resolução desses sistemas podem ser realizadas no *cluster*, a partir do que já foi desenvolvido, bem como, a investigação de pré-condicionadores, comparação do desempenho com outros métodos de resolução e paralelização dos métodos com outras ferramentas possibilitando uma melhor avaliação do DECK.

Palavras-Chaves: Sistemas Lineares Esparsos, Paralelização de Algoritmos, *Clusters*, Matemática da Computação, DECK

TITLE: "PARALELIZATION OF SPARSE LINEAR SYSTEMS RESOLUTION METHODS WITH DECK IN A CLUSTER OF PCS"

Abstract

The objective of this work is the paralelization and performance evaluation of some sparse linear systems resolution methods. DECK was used for their implementation in a cluster of PCs.

This research is motivated by the wide use of sparse linear systems in several scientific areas, especially, in the modeling of physical phenomena through Partial Differential Equations (PDEs). In this area, researches have been developed by GMC-PAD - Group of Computer Mathematics and High Performance Computing of UFRGS, for which this work contributes.

Another motivation factor for the realization of this research is the availability of a cluster of PCs in the Institute of Computer Science and the parallel programming environment DECK - Distributed Execution and Communication Kernel. DECK makes possible the programming in parallel environments with distributed shared memory. It is being developed by GPPD - Parallel and Distributed Processing Group and with the implementation of these methods on it, we intend to validate its operation and to evaluate its potential and its performance.

The linear systems originated by the discretization of PDEs have, in general, as characteristics the sparsity and the great number of variables. Due to the size of these systems, for its resolution it is necessary a great amount of memory and processing speed, characteristics of high performance computing.

Two resolution methods were studied and paralelized, one of the class of the direct methods, Thomas' Algorithm and another of the class of the iterative ones, the Conjugate Gradient. The paralelization of one method is completely different from the other. It happens because the iterative method is formed by basic operations of linear algebra, and the direct method is formed by elementary operations between lines and columns of the matrix of the coefficients of the linear system. This fact allowed the investigation and experimentation of different forms of parallelism.

Some different versions of the Conjugate Gradient were made. A version without preconditioning and versions preconditioned with the Diagonal preconditioner and with Polinomial preconditioner. Due to Thomas' Algorithm formulation, only the basic version was made.

After the paralelization of the resolution methods, the performance of the parallel algorithms was evaluated in the cluster, through the realization of measures of execution time and the speedup and the efficiency were calculated. The empirical measures were accomplished with variations in the order of the resolved systems and in the number of nodes used. This evaluation also involved the comparison between the complexities of the sequential algorithms and the complexity of the parallel algorithms of the methods.

This research demonstrates the performance of sparse linear systems resolution methods in a high performance environment, as well as the potentialities of DECK. Applications that involve the resolution of those systems can be developed in the cluster, as well as the investigation of preconditioners, comparison of the performance with other resolution methods and paralelization of the methods with other tools making possible a better evaluation of DECK.

Keywords: *Sparse Linear System, Algorithms Paralization, Clusters, Computational Mathematics, DECK*

1 Introdução

Muitos fenômenos físicos podem ser modelados consistentemente por equações diferenciais parciais (EDPs). Essas equações são definidas em um espaço contínuo e infinito de pontos e para serem tratadas computacionalmente, faz-se necessário transformar esse espaço em um espaço discreto e finito de pontos. O processo chamado discretização é responsável por essa transformação, gerando uma malha de pontos e obtendo o espaço discreto, permitindo, assim, a manipulação computacional das equações.

Conforme o esquema de discretização empregado na resolução de uma determinada EDP, podem ser originados sistemas de equações lineares algébricas (SELAs). Os sistemas lineares estão entre os mais freqüentes problemas tratados pela computação científica [SAA96] e há, basicamente, duas classes de métodos que os resolvem, a dos métodos diretos e a dos iterativos.

Os sistemas lineares originados pela discretização de EDPs têm, em geral, como características a esparsidade e a numerosa quantidade de incógnitas. Devido ao porte dos SELAs, para a resolução é necessária grande quantidade de memória e velocidade de processamento, característicos de computações de alto desempenho.

A paralelização de alguns métodos de resolução de sistemas lineares é o escopo dessa pesquisa. A Equação Diferencial Parcial da Difusão bidimensional, também chamada de Equação do Calor, discretizada através das Diferenças Finitas Centradas, juntamente com a técnica ADI – *Alternating Direction Implicit*, é aqui empregada como elemento motivador do trabalho e gerador dos SELAS. Como as classes de métodos que resolvem sistemas lineares abrangem um número significativo dos mesmos, foram escolhidos dois para serem paralelizados, um método direto, o Algoritmo de Thomas e um método iterativo, o Gradiente Conjugado.

A forma de paralelizar um método é completamente diferente do outro. Isso porque o método iterativo é formado por operações básicas de álgebra linear, e o método direto é formado por operações elementares entre linhas e colunas da matriz dos coeficientes do sistema linear. Isso permitiu a investigação e experimentação de formas distintas de paralelismo. Segundo Foster [FOS95], para o desenvolvimento de algoritmos paralelos não basta seguir uma simples receita, mas antes disso é necessária uma boa dose de criatividade para serem alcançados bons resultados.

Do método do Gradiente Conjugado, foram feitas a versão sem pré-condicionamento e versões pré-condicionadas com o pré-condicionador Diagonal e com o pré-condicionador Polinomial. Do Algoritmo de Thomas, devido a sua formulação, somente a versão básica foi feita.

A máquina paralela utilizada para implementação e para a realização de testes numéricos e computacionais dos algoritmos foi o *cluster* de PCs [RIG99], disponível no Instituto de Informática da UFRGS. Esse é um sistema com memória distribuída e compartilhada. Entre os nodos do *cluster*, têm-se um sistema com memória distribuída, onde há necessidade de trocas de mensagens, para que seja possível a comunicação entre os processadores. Alguns nodos possuem dois processadores, e para que esses dois processadores participem do processamento, é preciso a utilização de recursos de programação que proporcionem o compartilhamento da memória. Nesse caso, foram empregadas *threads* para a programação dos algoritmos.

A ferramenta que proporcionou a programação paralela foi o DECK – *Distributed Execution and Communication Kernel* [BAR2000]. O DECK está em desenvolvimento pelo GPPD – Grupo de Processamento Paralelo e Distribuído da UFRGS e faz parte do projeto *MultiCluster* [GPP2000].

Após a paralelização dos métodos de resolução, avaliou-se o desempenho dos algoritmos paralelos no *cluster*, através da realização de medidas do tempo de execução e foram calculados o *speedup* e a eficiência. Essa investigação também envolveu a comparação entre as complexidades dos algoritmos seqüenciais e a complexidade dos algoritmos paralelos.

As medidas empíricas foram realizadas com variações na ordem dos sistemas resolvidos e no número de nodos utilizados do *cluster*. Observou-se, para os vários tamanhos da entrada, qual o número ideal de nodos a serem usados e em quais casos há ganho de desempenho com a paralelização.

1.1 Motivação da Pesquisa

A presente pesquisa é motivada pela vasta utilização de Sistemas de Equações Lineares em várias áreas científicas, especialmente, na modelagem de fenômenos físicos através de EDPs. Nessa área, têm sido desenvolvidas pesquisas pelo GMC–PAD – Grupo de Matemática da Computação e Processamento de Alto Desempenho da UFRGS, para as quais esse trabalho vem contribuindo.

Pesquisas tem sido feitas, dentro do GMC–PAD, sobre a paralelização de modelos hidrológicos e aplicações de EDPs na modelagem de fenômenos físicos [DOR99, RIZ99]. Como exemplo, pode-se citar a modelagem computacional do escoamento e do transporte de massa no rio Guaíba [DOR2000]. Em tais aplicações, está inserida a resolução de sistemas lineares e nesse sentido o presente trabalho tem contribuído.

Nessa pesquisa, os sistemas lineares originários da EDP da Difusão são estudados e servem de exemplo para a aplicação dos métodos paralelos. Essa equação modela fenômenos físicos como a condução do calor ou fenômenos que envolvem massa ou movimento [FLE88].

Muitas pesquisas têm sido feitas sobre paralelização de métodos de resolução. Pode-se considerar que os métodos iterativos são mais favoráveis à paralelização, como tem sido encontrado na revisão bibliográfica [SAA96]. Já os métodos diretos são o maior desafio para tal, por apresentarem, em geral, dependências de dados [KUM94, DUF89]. Por esse fato, buscou-se observar o desempenho da paralelização de um método iterativo e um método direto.

Outro fator de motivação para a realização dessa pesquisa foi a validação do ambiente de programação paralela DECK, quanto ao seu seu funcionamento, seu potencial e seu desempenho.

1.2 Organização do Texto

Este texto está organizado em sete capítulos. Nesse inicial, é realizada uma introdução e apresentada a motivação do trabalho.

No segundo capítulo, uma breve explanação sobre Equações Diferenciais Parciais é realizada. Especificamente, é tratada a Equação Diferencial Parcial da Difusão,

relacionada a fenômenos físicos, como por exemplo: temperatura, velocidade ou concentração. As equações uni e bi-dimensionais são caracterizadas, discretizadas e apresentados os sistemas lineares resultantes, que podem ser resolvidos pelos métodos abordados nessa pesquisa. Nesse capítulo, também são caracterizados os sistemas lineares esparsos e mostrada sua aplicabilidade.

No terceiro capítulo, são revisados conceitos básicos de sistemas lineares, foco dessa pesquisa, com o objetivo de formalizar a nomenclatura e introduzir os métodos de resolução de sistemas. Há duas classes de métodos, e para cada uma delas, um método é estudado: para os métodos diretos, o método do Algoritmo de Thomas e para os métodos iterativos, o método do Gradiente Conjugado. Em suma, nesse capítulo são caracterizados sistemas lineares, suas aplicações, características, formas de armazenamento da matriz esparsa e métodos de resolução.

O capítulo quatro apresenta o ambiente paralelo onde esta pesquisa se desenvolveu. Esse ambiente envolve uma máquina paralela e um ambiente de programação paralela. Nesse sentido, é descrito o *cluster* de PCs e a ferramenta DECK que foi utilizada para a programação paralela dos algoritmos. Esse capítulo proporciona uma visão geral do ambiente paralelo, abordando aspectos de arquitetura e de programação.

No capítulo cinco, é tratada a paralelização dos métodos de resolução. O método iterativo do Gradiente Conjugado foi paralelizado, bem como suas versões pré-condicionadas com o precondicionador Diagonal e com o Polinomial. Para o método direto do Algoritmo de Thomas, somente uma versão do mesmo foi implementada, dadas suas particularidades. A paralelização de cada versão é descrita, juntamente com as técnicas utilizadas e problemas enfrentados. Também no Gradiente Conjugado foi abordada a utilização de *threads* no algoritmo para usufruir da capacidade dos dois processadores, nos nodos Dual Pentium.

No capítulo seis, são tecidos comentários sobre a avaliação do desempenho de cada versão paralela dos métodos. Para tanto, foram considerados tempo de execução, *speedup* e eficiência de cada versão/método. Medidas empíricas foram tomadas, considerando uma variação no tamanho da entrada e no número de nodos utilizados no processamento. São feitas, ainda, comparações quanto à complexidade dos algoritmos nos casos seqüencial e paralelo.

Por fim, no capítulo sete são apresentadas as conclusões do trabalho. Isso inclui a paralelização de métodos de resolução (direto / iterativo) de sistemas lineares esparsos no *cluster* de PCs, o uso do ambiente de programação DECK, os principais resultados obtidos pela observação e a avaliação dos métodos paralelos. Para o GMC–PAD, essa pesquisa contribuiu diretamente na área de Aplicações de Alto Desempenho.

2 Equações Diferenciais Parciais

As Equações Diferenciais Parciais possibilitam a modelagem de problemas reais em áreas como Ciências e Engenharia. Freqüentemente, são empregadas na modelagem de fenômenos físicos, como por exemplo na dinâmica dos fluidos, na previsão meteorológica, em estudos de elasticidade, eletromagnetismo, entre outros [CUN92, SAA96].

Neste capítulo é feita uma introdução às Equações Diferenciais Parciais (EDPs), particularmente à EDP da Difusão. A equação da difusão é usada na modelagem de fenômenos de difusão como, por exemplo temperatura, velocidade ou concentração. O objetivo desse capítulo é apresentar como as EDPs resultam em sistemas lineares.

Inicialmente, as Equações Diferenciais Parciais são caracterizadas e sua resolução abordada teoricamente, sendo dado enfoque à equação da Difusão, utilizada como estudo de caso para os métodos paralelizados. Dessa forma, identificam-se esquemas de discretização das equações, percorrendo todo o caminho até se chegar aos sistemas de equações lineares, cuja resolução em paralelo é objeto de estudo.

Uma Equação Diferencial é constituída pela relação das variáveis com suas derivadas. Boa parte dos fenômenos físicos é modelada por essas equações [CLA94]. Uma Equação Diferencial envolve uma função desconhecida e algumas de suas derivadas, ou seja, estabelece relações entre uma variável que depende de duas ou mais variáveis independentes e as derivadas parciais.

2.1 Equação Diferencial Parcial da Difusão

A classe de EDPs é muito abrangente. Como forma de aplicar os métodos paralelos a um problema real, optou-se por trabalhar com a Equação Diferencial Parcial da Difusão.

A Equação Diferencial Parcial da Difusão, segundo [FLE88], possui o mesmo mecanismo dissipativo que é encontrado em efeitos de condução de calor. Essa equação pode ser uni ou multidimensional. Qualquer que seja sua dimensão ela sempre se refere ao espaço. Além disso, essa equação é dita não-estacionária, pois evolui com o tempo.

Na solução de EDPs há condições de contorno e condições iniciais que devem ser observadas. As condições de contorno (C.C.) são condições sobre o valor da solução e suas derivadas no bordo da região [IOR91]. As C.C. de *Dirichlet* são dadas por constantes associados aos valores de contorno da região e as de *Neumann*, geralmente por uma derivada normal à fronteira da superfície. As C.C. de *Dirichlet* consideram que a região esteja isolada e as de *Neumann*, que a região sofre influência do meio onde está inserida.

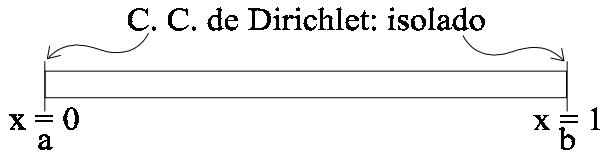
As condições iniciais (C.I.), tendo-se presente a existência de mais de uma variável nas EDPs, consistem em fixar uma das variáveis, por exemplo $t = 0$, e impor o valor solução às demais variáveis e suas derivadas parciais, em relação à variável fixa [IOR91].

2.2 EDP da Difusão Unidimensional

A EDP da Difusão Unidimensional (ou Equação do Calor unidimensional) pode ser representada por:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2}\right) \tag{2.1}$$

A variável T nessa equação pode ser velocidade, temperatura ou concentração. Isso depende do tipo de difusão que se está tratando: momento, calor ou massa. Conforme comentado anteriormente, são necessárias condições de contorno (C.C.) e condições iniciais (C.I.) para a resolução da EDP. Um exemplo de C.C. de Dirichlet, para a EDP da Difusão Unidimensional, pode ser observado na figura 2.1.



$$\text{C.C.}\begin{cases} T(0,t) = a \\ T(1,t) = b \end{cases} \quad 0 \le x \le 1$$

$$\text{C.I.}\{T(x,0) = T_0(x)$$

FIGURA 2.1 - Condução de Calor em uma Barra

As condições de contorno de *Dirichlet*, para esse exemplo, são definidas da seguinte forma: em qualquer tempo, quando *x=0*, a temperatura (C.C.) é igual a *a* e quando *x=1*, a temperatura é igual a *b*. As condições iniciais indicam que, no tempo zero, em qualquer *x*, há uma função $T_0(x)$. A solução analítica (ou exata) satisfaz as condições de contorno e as condições iniciais da equação, aplicadas a *x=0* e *x=1*.

Para obter uma solução aproximada, a EDP da Difusão é discretizada. De forma simplificada, esse processo pode ser entendido como a definição de uma malha sobre o domínio contínuo da equação, que contém um número infinito de pontos, fazendo-se uma aproximação da solução. A equação resultante desse processo é manipulada para gerar a equação que aproxima a solução, em um nível de tempo, em termos da solução já conhecida dos níveis anteriores e das condições de contorno.

Nesse estudo, a discretização é feita por Diferenças Finitas Centradas [FLE88]. O método das diferenças é uma aproximação de uma EDP, no sentido que as derivadas em um ponto são aproximadas por quocientes em diferenças para pequenos intervalos. Na figura 2.2, tem-se a representação do esquema de Diferenças Finitas Centradas de 1ª ordem.

Um esquema explícito de discretização, adiantado no tempo e centrado no espaço, aplicado à Equação da Difusão unidimensional, resulta em: $\frac{T_j^{n+1} - T_j^n}{\Delta t} = \alpha\left(\frac{T_{j-1}^n - 2T_j^n + T_{j+1}^n}{\Delta x^2}\right)$. Para esquemas implícitos o termo espacial $\frac{\partial^2 T}{\partial x^2}$ da equação é avaliado no nível desconhecido de tempo *(n+1)*.

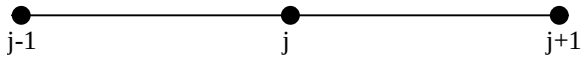


FIGURA 2.2 - Estêncil para Diferenças Finitas Centradas

2.2.1 Esquema de Discretização Crank-Nicolson

O mais simples esquema implícito de diferenças finitas que representa a Equação da Difusão, dada em (2.1) e reescrita como $\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2}\right) = 0$, é:

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} - \alpha\left(\frac{T_{j-1}^{n+1} - 2T_j^{n+1} + T_{j+1}^{n+1}}{\Delta x^2}\right) = 0 \qquad (2.2)$$

Para se obter a equação em uma forma mais conveniente, multiplica-se (2.2) por Δt, chegando-se a:

$$\left(T_j^{n+1} - T_j^n\right) - \frac{\alpha\,\Delta t}{\Delta x^2}\left(T_{j-1}^{n+1} - 2T_j^{n+1} + T_{j+1}^{n+1}\right) = 0$$

Fazendo $\frac{\alpha\,\Delta t}{\Delta x^2} = \mathrm{s}$, tem-se:

$$T_j^{n+1} - T_j^n - sT_{j-1}^{n+1} + 2sT_j^{n+1} - sT_{j+1}^{n+1} = 0 \quad \Rightarrow$$
$$(1+2s)T_j^{n+1} - T_j^n - sT_{j-1}^{n+1} - sT_{j+1}^{n+1} = 0 \quad \Rightarrow$$
$$-sT_{j-1}^{n+1} + (1+2s)T_j^{n+1} - sT_{j+1}^{n+1} = T_j^n \qquad (2.3)$$

A equação (2.3) é a equação resultante. Para resolvê-la, é necessário considerar todos os nós *j* e as correspondentes equações. Como as condições de contorno envolvem *j*=1 e *j*=J, pois são o bordo da região, e são dadas por $\text{C.C.} = \begin{cases} T(1,t) = d_1 \\ T(J,t) = d_J \end{cases}$, onde $d_1 = T_1^{n+1}$, $d_J = T_J^{n+1}$ e $n+1$ é o nível de tempo desconhecido, o processo é o seguinte:

$$j = 2$$
$$-sT_1^{n+1} + (1+2s)T_2^{n+1} - sT_3^{n+1} = T_2^n \Rightarrow (1+2s)T_2^{n+1} - sT_3^{n+1} = T_2^n + sT_1^{n+1} = d_2$$
$$\left.\begin{array}{l} j = 3 \\ -sT_2^{n+1} + (1+2s)T_3^{n+1} - sT_4^{n+1} = T_3^n \\ j = 4 \\ -sT_3^{n+1} + (1+2s)T_4^{n+1} - sT_5^{n+1} = T_4^n \\ \vdots \end{array}\right\} d_j = T_j^n$$
$$j = J-1$$
$$-sT_{J-2}^{n+1} + (1+2s)T_{J-1}^{n+1} - sT_J^{n+1} = T_{J-1}^n \Rightarrow -sT_{J-2}^{n+1} + (1+2s)T_{J-1}^{n+1} = T_{J-1}^n + sT_J^{n+1} = d_{J-1}$$

O sistema de equações lineares algébricas resultante da discretização da EDP da Difusão unidimensional tem matriz do tipo banda, tridiagonal conforme figura 2.3. Os

$d_i s$ formam o vetor de termos independentes e dependem diretamente das condições de contorno da EDP e do valor das variáveis no passo de tempo anterior.

$$\begin{pmatrix} (1+2s) & -s & & & & \\ -s & (1+2s) & -s & & & \\ & -s & (1+2s) & -s & & \\ & & & \cdots & & \\ & & & & -s & (1+2s) \end{pmatrix} \begin{pmatrix} T_2^{n+1} \\ T_3^{n+1} \\ \vdots \\ \\ T_{J-1}^{n+1} \end{pmatrix} = \begin{pmatrix} d_2 \\ d_3 \\ \vdots \\ \\ d_{J-1} \end{pmatrix}$$

FIGURA 2.3 – SELA Resultante da Discretização da Equação da Difusão Unidimensional

2.3 EDP da Difusão Multidimensional

A EDP da Difusão Multidimensional é usualmente aplicada à problemas bi e tridimensionais. Aqui, é enfocada a Equação Diferencial Parcial da Difusão bidimensional que pode ser representada por:

$$\frac{\partial T}{\partial t} = \alpha_x \left(\frac{\partial^2 T}{\partial x^2} \right) + \alpha_y \left(\frac{\partial^2 T}{\partial y^2} \right) \tag{2.4}$$

A figura 2.4 mostra uma região bidimensional, representando a EDP da Difusão, com as condições de contorno de Dirichlet.

Na discretização de equações unidimensionais no espaço com avanço no tempo, esquemas implícitos como, por exemplo o utilizado anteriormente, Crank-Nicolson [FLE88], são suficientes para esse propósito, pois o sistema de equações resultante é tridiagonal e há métodos numéricos eficientes para sua resolução [RIZ98]. Contudo, quando a equação é bidimensional, o sistema resultante não é tridiagonal e, apesar de ser incondicionalmente estável, há dificuldade de obtenção de uma solução econômica [FLE88]. Assim, torna-se interessante a aplicação de esquemas que permitam ou transformem o sistema de equações lineares algébricas em um sistema banda, tridiagonal.

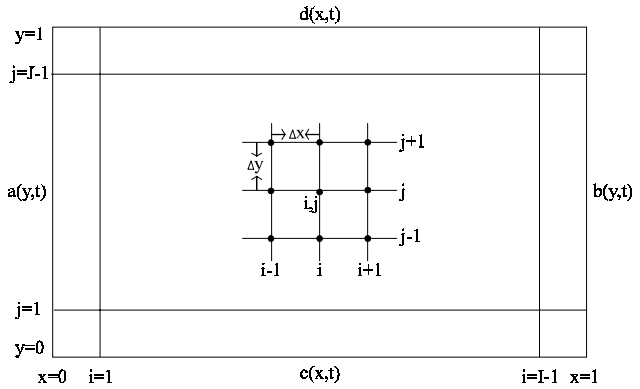
A equação resultante de um esquema Crank-Nicolson usado para a discretização da EDP da Difusão bidimensional é a equação (2.6). Ela é obtida a partir de (2.5), fazendo $\alpha_x = \alpha_y$. A aplicação do esquema implícito para o caso bidimensional, é semelhante ao da EDP da Difusão unidimensional, diferenciando-se por ser em um espaço de duas dimensões. Aqui é apresentado apenas o resultado da discretização, e não, todos os passos, como no caso anterior.

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = \alpha_x \frac{\frac{1}{2}\left(T_{i+1,j}^{n+1} + T_{i+1,j}^{n}\right) + \frac{1}{2}\left(-2T_{i,j}^{n+1} - 2T_{i,j}^{n}\right) + \frac{1}{2}\left(T_{i-1,j}^{n+1} + T_{i-1,j}^{n}\right)}{(\Delta x)^2} + \alpha_y \frac{\frac{1}{2}\left(T_{i,j+1}^{n+1} + T_{i,j+1}^{n}\right) + \frac{1}{2}\left(-2T_{i,j}^{n+1} - 2T_{i,j}^{n}\right) + \frac{1}{2}\left(T_{i,j-1}^{n+1} + T_{i,j-1}^{n}\right)}{(\Delta y)^2} \tag{2.5}$$

A equação (2.5) pode ser escrita como:

$$-s_x T_{i-1,j}^{n+1} + (1+2s_x+2s_y)T_{i,j}^{n+1} - s_x T_{i+1,j}^{n+1} - s_y T_{i,j-1}^{n+1} - s_y T_{i,j+1}^{n+1} = T_{i,j}^{n} \qquad (2.6)$$

O sistema de equações lineares nessa discretização é também do tipo banda, com cinco diagonais. Como ressaltado anteriormente, sistemas lineares tridiagonais possibilitam a utilização de métodos de resolução eficientes. Uma solução para transformar o sistema linear com cinco diagonais em sistemas tridiagonais é dividir a equação da solução, resultante do esquema implícito, em dois "meios" passos para compor um passo no tempo. A cada meio passo, somente os termos associados com uma direção de coordenada são tratados implicitamente. Como conseqüência, somente três termos implícitos aparecem, resultando em um sistema tridiagonal. O procedimento para tratar cada passo de tempo como uma seqüência de subpassos mais simples é chamado de particionamento do tempo. Na próxima seção é vista uma técnica desse tipo.



$$\text{C.C.} = \begin{cases} T(0,y,t) = a(y,t) \\ T(1,y,t) = b(y,t) \\ T(x,0,t) = c(x,t) \\ T(x,1,t) = d(x,t) \end{cases}$$

$$\text{C.I.} = T(x,y,0) = T_0(x,y)$$

FIGURA 2.4 - Domínio Discreto para a Equação da Difusão Bidimensional

2.3.1 Discretização com a Técnica ADI

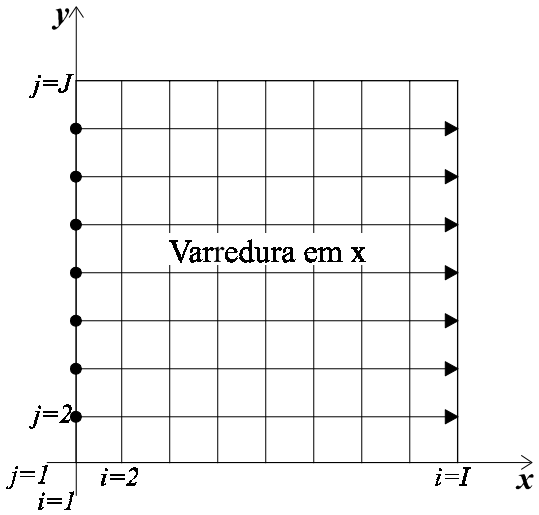
A técnica ADI (*Alternating Direction Implicit*) é uma técnica de discretização, com particionamento de tempo, bastante conhecida [FLE88, AND95]. No caso bidimensional, particiona o tempo em dois e a equação é resolvida em dois passos, um implícito em *x* e outro em *y*. Por exemplo, no primeiro passo de tempo, os termos em *x* são implícitos e os em *y* são explícitos e no segundo passo, vice-versa.

A EDP bidimensional da Difusão é discretizada através das Diferenças Finitas Centradas juntamente com a técnica ADI. Essa técnica permite manipular a equação (2.5) com três incógnitas de cada vez, dando origem a um sistema tridiagonal, que é o que se busca. A EDP da Difusão pode ser representada como em (2.7), considerando-se $\alpha = \alpha_x = \alpha_y$.

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{2.7}$$

No primeiro passo de tempo $\frac{\Delta t}{2}$, é usada a equação (2.8). A solução de *T* é conhecida no nível de tempo *n*, mas desconhecida no nível $n+\frac{1}{2}$. Contudo, os valores $T^{n+\frac{1}{2}}$ são associados somente com a direção *x*, ficando os valores em *y* "congelados", ou seja, são explícitos pois na direção *y* são usados os valores T^n, já conhecidos. A ilustração da varredura em *x* pode ser vista na figura 2.5.

$$\frac{T_{i,j}^{n+\frac{1}{2}} - T_{i,j}^{n}}{\frac{\Delta t}{2}} = \alpha\frac{\left(T_{i+1,j}^{n+\frac{1}{2}} - 2T_{i,j}^{n+\frac{1}{2}} + T_{i-1,j}^{n+\frac{1}{2}}\right)}{(\Delta x)^2} + \alpha\frac{\left(T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}\right)}{(\Delta y)^2} \tag{2.8}$$



$$t = t + \frac{\Delta t}{2}$$

FIGURA 2.5 - Técnica ADI – Varredura em *x*

Em uma malha *I* x *J*, *y* é explícito, e para cada valor da coordenada *y*, a coordenada *x* varia *I* vezes. A equação (2.8) pode ser reduzida à forma tridiagonal (2.9).

$$AT_{i-1,j}^{n+\frac{1}{2}} - BT_{i,j}^{n+\frac{1}{2}} + AT_{i+1,j}^{n+\frac{1}{2}} = K_i \qquad 2 \le i \le \mathrm{I} - 1 \tag{2.9}$$

Onde:

$$A = \alpha\frac{\Delta t}{2(\Delta x)^2} \qquad B = 1 + \alpha\frac{\Delta t}{(\Delta x)^2} \qquad K_i = -T_{i,j}^{n} - \alpha\frac{\Delta t}{2(\Delta y)^2}\left(T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}\right)$$

Nesse passo, cada *y* é explícito e na direção *x* é feita a varredura.

A solução é obtida assim: para uma dada coordenada *y*, faz-se uma varredura na direção *x* usando (2.9) para todos os valores de i ($2 \le i \le I-1$). Se há *I* pontos na malha na direção *x*, a varredura é feita de *i = 1* até *I*. Se há *J* pontos em *y*, o procedimento é repetido *J* vezes (*J* varreduras em *x*), tendo-se assim *J* sistemas lineares para serem resolvidos, ou seja, um para cada linha. No final desse passo, os valores de $T_{i,j}^{n+\frac{1}{2}}$ são conhecidos em todos os pontos e a segunda etapa da técnica ADI pode ser feita.

Para o segundo passo, o tempo é $t + \Delta t$, somente as variáveis em *y* são tratadas implicitamente e *x* é explícito. A discretização é dada por (2.10) e a figura 2.6 ilustra a varredura em *y*.

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n+\frac{1}{2}}}{\frac{\Delta t}{2}} = \alpha \frac{\left(T_{i+1,j}^{n+\frac{1}{2}} - 2T_{i,j}^{n+\frac{1}{2}} + T_{i-1,j}^{n+\frac{1}{2}} \right)}{(\Delta x)^2} + \alpha \frac{\left(T_{i,j+1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j-1}^{n+1} \right)}{(\Delta y)^2} \qquad (2.10)$$

A equação (2.10) reduzida à forma tridiagonal, resulta na equação (2.11):

$$CT_{i,j+1}^{n+1} - DT_{i,j}^{n+1} + CT_{i,j-1}^{n+1} = L_i \qquad 2 \le \text{j} \le \text{J -1} \qquad (2.11)$$

Onde:

$$C = \alpha \frac{\Delta t}{2(\Delta y)^2} \quad D = 1 + \alpha \frac{\Delta t}{(\Delta y)^2} \quad L_i = -T_{i,j}^{n+\frac{1}{2}} - \alpha \frac{\Delta t}{2(\Delta x)^2} \left(T_{i+1,j}^{n+\frac{1}{2}} - 2T_{i,j}^{n+\frac{1}{2}} + T_{i-1,j}^{n+\frac{1}{2}} \right)$$

Como $T^{n+\frac{1}{2}}$ é conhecido em todos os pontos da malha pelo primeiro passo, a equação (2.11) determina a solução para $T_{i,j}^{n+1}$, mantendo o *i* fixo e fazendo-se a varredura na direção *y* para todos os valores de *j* ($2 \le j \le J-1$). Esse cálculo é repetido *I* passos na direção *x*, aplicando o método que resolve o sistema tridiagonal

A formação do sistema linear, conforme as varreduras em *x* ou em *y*, é exemplificada a seguir para o caso de varreduras em *x*. Para o caso de varreduras em *y*, o procedimento é semelhante. Assim, para as condições de contorno em *x*, usando a expressão genérica (2.9), tem-se:

Com j=2 fixo e fazendo-se varreduras em *x* ($2 \le i \le I-1$)

i = 2 $\quad AT_{1,2}^{n+\frac{1}{2}} - BT_{2,2}^{n+\frac{1}{2}} + AT_{3,2}^{n+\frac{1}{2}} = K_2 \Rightarrow -BT_{2,2}^{n+\frac{1}{2}} + AT_{3,2}^{n+\frac{1}{2}} = K_2 - AT_{1,2}^{n+\frac{1}{2}}$

i = 3 $\quad AT_{2,2}^{n+\frac{1}{2}} - BT_{3,2}^{n+\frac{1}{2}} + AT_{4,2}^{n+\frac{1}{2}} = K_3$

i = 4 $\quad AT_{3,2}^{n+\frac{1}{2}} - BT_{4,2}^{n+\frac{1}{2}} + AT_{5,2}^{n+\frac{1}{2}} = K_4$

...

i = I-1 $\quad AT_{I-2,2}^{n+\frac{1}{2}} - BT_{I-1,2}^{n+\frac{1}{2}} + AT_{I,2}^{n+\frac{1}{2}} = K_{I-1} \Rightarrow AT_{I-2,2}^{n+\frac{1}{2}} - BT_{I-1,2}^{n+\frac{1}{2}} = K_{I-1} - AT_{I,2}^{n+\frac{1}{2}}$

Com j=J-1 fixo e fazendo-se varreduras em x ($2 \le i \le \mathrm{I}-1$)

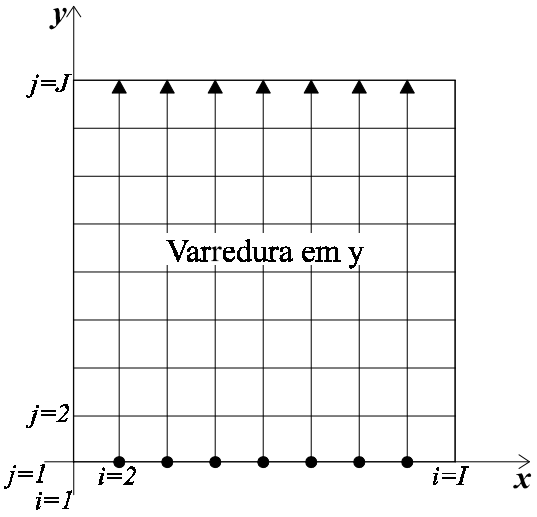
$i = 2$ $\quad AT_{1,J-1}^{n+\frac{1}{2}} - BT_{2,J-1}^{n+\frac{1}{2}} + AT_{3,J-1}^{n+\frac{1}{2}} = K_2 \Rightarrow -BT_{2,J-1}^{n+\frac{1}{2}} + AT_{3,J-1}^{n+\frac{1}{2}} = K_2 - AT_{1,J-1}^{n+\frac{1}{2}}$

$i = 3$ $\quad AT_{2,J-1}^{n+\frac{1}{2}} - BT_{3,J-1}^{n+\frac{1}{2}} + AT_{4,J-1}^{n+\frac{1}{2}} = K_3$

$i = 4$ $\quad AT_{3,J-1}^{n+\frac{1}{2}} - BT_{4,J-1}^{n+\frac{1}{2}} + AT_{5,J-1}^{n+\frac{1}{2}} = K_4$

...

$i = \mathrm{I}\text{-}1$ $\quad AT_{I-2,J-1}^{n+\frac{1}{2}} - BT_{I-1,J-1}^{n+\frac{1}{2}} + AT_{I,J-1}^{n+\frac{1}{2}} = K_{I-J-1} \Rightarrow AT_{I-2,J-1}^{n+\frac{1}{2}} - BT_{I-1,J-1}^{n+\frac{1}{2}} = K_{I-1} - AT_{I,J-1}^{n+\frac{1}{2}}$



$t = t + \Delta t$

FIGURA 2.6 - Técnica ADI – Varredura em y

O sistema de equações lineares tridiagonal com j fixo e varreduras em x, para a EDP da Difusão bidimensional, é mostrado a seguir. Esse é o sistema linear genérico para cada j explícito.

$$\begin{pmatrix} B & -A & & & & \\ -A & B & -A & & & \\ & -A & B & -A & & \\ & & & \cdots & & \\ & & & & -A & B \end{pmatrix} \begin{pmatrix} T_{2,j}^{n+\frac{1}{2}} \\ T_{3,j}^{n+\frac{1}{2}} \\ T_{4,j}^{n+\frac{1}{2}} \\ \vdots \\ T_{I-1,j}^{n+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_{I-1} \end{pmatrix}$$

onde: $A = \alpha \dfrac{\Delta t}{2(\Delta x)^2}$ $\quad B = 1 + \alpha \dfrac{\Delta t}{(\Delta x)^2}$ $\quad K_i = -T_{i,j}^n - \alpha \dfrac{\Delta t}{2(\Delta y)^2}\left(T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n\right)$

$$d_2 = -\left(K_2 - AT_{i,j}^{n+\frac{1}{2}}\right) \quad d_{I-1} = -\left(K_{I-1} - AT_{I,j}^{n+\frac{1}{2}}\right) \quad d_i = -K_i, \quad \text{com} \quad 2 \le i \le I-1.$$

2.4 Considerações Finais

Neste capítulo foi introduzida a Equação Diferencial Parcial da Difusão cuja discretização origina sistemas lineares. A EDP da aplicação é bidimensional e não estacionária. Foi utilizada a técnica ADI, como forma de evitar os sistemas lineares pentadiagonais, para que a discretização chegasse a sistemas tridiagonais, possibilitando a utilização de métodos de resolução computacionalmente econômicos.

Para a EDP da Difusão discretizada utilizando-se da técnica ADI, o sistema linear resultante é tridiagonal. Os métodos de resolução que serão paralelizados se destinam também a outros tipos de sistemas lineares, inclusive gerados por outras equações diferenciais.

No próximo capítulo os sistemas lineares são apresentados, e os sistemas aqui vistos são caracterizados com relação ao armazenamento e métodos de resolução. No capítulo 6, onde é feita a análise dos resultados obtidos, há o resultado da aplicação dos métodos paralelos na resolução dos sistemas lineares da Equação Diferencial Parcial da Difusão, discutida nesse capítulo.

3 Sistemas Lineares

Sistemas Lineares ou Sistemas de Equações Lineares Algébricas (SELAs) estão presentes em várias aplicações como programação linear, dinâmica dos fluidos, modelagem do clima e previsão meteorológica, aparecendo basicamente em problemas de engenharia e computação científica [KUM94, CUN92, SAA96, KEL95]. Tais sistemas são geralmente grandes, podendo chegar a milhares de incógnitas, e esparsos, dependendo do modelo matemático que os origina.

No capítulo anterior foi visto que a resolução de EDPs que modelam fenômenos físicos também pode ser feita através de sistemas lineares esparsos. Os sistemas lineares do tipo banda [DUF89] abrangem uma classe de sistemas cuja matriz é esparsa e concentra seus elementos não nulos em diagonais próximas à diagonal principal. Exemplos desses são os sistemas tridiagonais e pentadiagonais gerados na discretização da EDP da Difusão.

O estudo de sistemas esparsos tem sua importância não somente pelo fato de serem encontrados freqüentemente em problemas da computação científica, mas também por envolverem algoritmos e estruturas de dados mais complexas que os sistemas densos [KUM94]. Nesse capítulo são caracterizados sistemas lineares, abordadas formas de armazenamento de matrizes esparsas e os métodos de resolução desses sistemas.

Os métodos de resolução de sistemas lineares podem ser agrupados em duas classes: a dos métodos diretos e a dos iterativos. Os métodos diretos caracterizam-se por encontrarem a solução exata do sistema linear, salvo erros de arredondamento, através de um número pré-definido de passos. Já os métodos iterativos buscam a solução através de aproximações sucessivas dos valores das incógnitas do sistema até um limite aceitável de erro ou um número máximo de iterações ser alcançado.

São abordados dois métodos de resolução, um de cada classe: o Algoritmo de Thomas e o método do Gradiente Conjugado. Esses métodos foram escolhidos dentre outros métodos de suas respectivas classes, por serem eficientes na resolução de sistemas lineares do tipo banda [FLE88, SAA96, SHE94]. Por fim, algumas bibliotecas de álgebra linear são referenciadas, proporcionando uma visão geral de ferramentas para a resolução de sistemas lineares disponíveis e suas características principais.

3.1 O que são Sistemas Lineares?

Sistemas Lineares são formados por várias equações lineares. Se existirem *n* equações em um sistema, cada uma com *m* incógnitas, essas equações podem ser representadas matricialmente por $A_{n\,x\,m}x_{m\,x\,1} = b_{n\,x\,1}$, onde *A* é a matriz dos coeficientes, *x* é o vetor das incógnitas, que se deseja encontrar e *b* é o vetor dos termos independentes. Na figura 3.1, tem–se a representação de um sistema linear com *n* equações e *m* variáveis.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n \end{cases} \Leftrightarrow Ax = b \Leftrightarrow \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

FIGURA 3.1- Representação de Sistemas Lineares

Os sistemas lineares esparsos diferenciam-se dos outros por possuírem muitos elementos nulos na matriz dos coeficientes. Na figura 3.2, tem-se um exemplo de um sistema linear esparso.

Na resolução de um sistema linear há três situações distintas:

a) o sistema é possível e determinado: a matriz dos coeficientes é não singular, ou seja, o determinante da matriz é diferente de zero; portanto há uma única solução para o sistema;

b) o sistema é possível e indeterminado: a matriz dos coeficientes é singular e o sistema possui infinitas soluções;

c) o sistema é impossível: não há solução para o sistema.

Para esse estudo, considera-se que a matriz dos coeficientes *A* seja quadrada (possua o mesmo número de linhas e colunas, ou seja, ordem *n* x *n*), não singular (o determinante da matriz é diferente de zero), simétrica (a matriz transposta de *A* é a própria matriz *A*), positiva definida (para todo $x \neq 0$, a relação $x^T Ax > 0$ é verdadeira), e diagonalmente dominante (para cada elemento a_{ii} , tem-se que $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$), além de ser esparsa [SAA96]. Essas considerações vão de encontro às características dos sistemas lineares originados da discretização de muitas EDPs.

3.2 Matrizes Esparsas

Uma matriz é esparsa quando a maioria dos seus elementos são nulos. A esparsidade proporciona otimizações no armazenamento e, frequentemente, otimizações também nas operações de álgebra linear, como por exemplo, na multiplicação matriz – vetor, que no caso esparso não precisa ter um laço que varra todos os elementos da matriz, mas somente os elementos não nulos.

Conforme a estrutura da matriz esparsa, definida pela distribuição dos seus elementos não nulos, as matrizes são classificadas. Um tipo de matriz esparsa é a matriz banda, cujos elementos diferentes de zero se encontram posicionados nas diagonais próximas à diagonal principal. Diz-se que essas matrizes são regularmente estruturadas [SAA96].

Matrizes banda podem ser classificadas pelo número de diagonais que possuem. Se a matriz dos coeficientes do sistema possui três diagonais: a principal, a inferior e a superior, ela é chamada tridiagonal. Na figura 3.2, há um exemplo de sistema linear cuja matriz é tridiagonal, de ordem 5.

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{pmatrix}_{5 \times 5} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}_{5 \times 1} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}_{5 \times 1}$$

FIGURA 3.2 - Exemplo de Sistema Linear Tridiagonal

3.2.1 Estruturas de Armazenamento

Há diversas formas de armazenamento para otimizar o armazenamento dos elementos não nulos de matrizes esparsas. Por exemplo, armazenamento por linha ou coluna esparsa comprimida, linha esparsa modificada, formato coordenado, formato diagonal, representação por listas, dentre outros que podem ser encontrados em [DUF89, SAA96, STE94, CAN99]. Para cada um deles, as operações básicas de álgebra linear com matrizes adquirem particularidades.

Ao se escolher uma forma de otimizar o armazenamento, deve-se levar em consideração a arquitetura do computador que está sendo usado e como o algoritmo será afetado pela utilização dessa estrutura, especialmente se esse for paralelo. Optou-se por armazenar a matriz no formato diagonal ou *array* retangular, porque, conforme será caracterizado a seguir, esse formato proporciona uma estrutura flexível de armazenamento para matrizes do tipo banda e também por ser uma estrutura que mantém a relação entre as linhas da matriz, facilitando o particionamento dos dados entre os processadores na paralelização.

3.2.1.1 Formato Diagonal

Na estrutura array retangular ou formato diagonal são necessários uma matriz M (de ordem *n x diag*) e um vetor de inteiros Dif (de ordem *diag x 1*, onde *n* é o número de equações e *diag* é o número de diagonais da matriz esparsa). Cada coluna da matriz *M* armazena uma diagonal da matriz esparsa *A*. O vetor *Dif* armazena a diferença da posição de cada diagonal em relação à diagonal principal.

A relação entre os elementos da matriz dos coeficientes *A* e da matriz otimizada *M* é dada por $m_{i,j} = a_{i,i+Dif_j}$. Essa relação é melhor ilustrada na figura 3.3 com um exemplo numérico.

$$A = \begin{pmatrix} 3 & 1 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 & 3 \end{pmatrix}_{5\,x\,5} \Leftrightarrow M = \begin{pmatrix} 0 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 0 \end{pmatrix}_{5\,x\,3} \quad Dif = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}_{1\,x\,3}$$

FIGURA 3.3 - Formato Diagonal de Armazenamento de Matriz Esparsa

Na multiplicação matriz esparsa pelo vetor, com o formato diagonal, cada elemento resultante do produto *Mb=c*, onde *b* e *c* são vetores, é dado abaixo. Essa equação não se aplica quando i=1, j=1 e quando i=*n* e j=*diag* (*diag* é o número de diagonais da matriz dos coeficientes), pois nesses casos não se tem elementos nessas posições, na matriz *M*, a serem multiplicados.

$$c_i = \sum_{j=1}^{Diag} m_{i,j} * b_{i+Dif_j}$$

Aqui, cada uma das diagonais é multiplicada pelo vetor *b* e o resultado adicionado ao vetor *c*. Do ponto de vista da paralelização ou vetorização, esse algoritmo é propício

para ser usado, comparado aos outros esquemas de armazenamento [SAA96], pois não apresenta dependências de dados.

Esse mesmo formato de armazenamento pode ser feito somente com vetores. Ao invés de usar o vetor *Dif* e a matriz M, utilizam-se tantos vetores quantos são as diagonais da matriz esparsa. Para o caso tridiagonal, por exemplo, são utilizados três vetores de ordem n. A desvantagem dessa abordagem está na não flexibilidade quando a matriz banda possui um número maior de diagonais não nulas.

3.3 Método Direto de Resolução de Sistemas Lineares

Os métodos diretos de resolução de sistemas lineares fornecem diretamente a solução através de um número pré-definido de passos. Basicamente, os métodos diretos trabalham em duas etapas: a primeira, também conhecida como *forward*, consiste em reorganizar a estrutura da matriz dos coeficientes, geralmente triangularizando-a, e a segunda chamada retrossubstituição ou *backward* faz a substituição das variáveis.

Essas etapas são realizadas por operações elementares entre linhas e colunas da matriz dos coeficientes, as quais alteram a estrutura da mesma. Devido a esse fato, há dificuldade em ser adotado um esquema de armazenamento que otimize a ocupação do espaço.

Essa dificuldade é encontrada nos métodos diretos clássicos como Eliminação de Gauss e Gauss-Jordan, além disso, a complexidade seqüencial desses é $O(n^3)$ [WES68, CAN99]. Já o Algoritmo de Thomas (AT), particularmente aplicável a sistemas cujas matrizes possuem três ou cinco diagonais adjacentes, permite otimizar o armazenamento e tem complexidade menor, sendo $O(n)$ [FLE88].

3.3.1 Algoritmo de Thomas

O Algoritmo de Thomas é uma variação do método de Eliminação de Gauss que trabalha com matrizes esparsas. A versão básica do método é destinada a sistemas tridiagonais, havendo também versões para resolução de outros sistemas lineares tipo banda.

Esse método, por ser da classe dos diretos, possui duas etapas: na primeira, denominada *forward*, os elementos da diagonal principal da matriz são normalizados, eliminando os elementos da diagonal inferior; na segunda, *backward*, o sistema resultante é resolvido por retrossubstituição [HIR88].

A segunda etapa desse método é mais rápida que a primeira porque possui um número menor de operações aritméticas, mas a complexidade das duas etapas é $O(n)$. Segundo [WES68], esse algoritmo para sistemas tridiagonais envolve um número de operações aritméticas proporcional a n, menor que $1/3n^3$ como exigido pela Eliminação de Gauss, em uma matriz genérica.

Nesse método, a matriz dos coeficientes é armazenada com o esquema por vetores, fazendo uso de três: *a, b, c*, que armazenam respectivamente as diagonais inferior, principal e superior. Desse modo, o sistema linear pode ser representado como na figura 3.4.

$$\begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & . & . & . & & & \\ & & a_i & b_i & c_i & & \\ & & & . & . & . & \\ & & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ . \\ x_i \\ . \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ . \\ d_i \\ . \\ d_{n-1} \\ d_n \end{pmatrix}$$

FIGURA 3.4 - Sistema Linear Tridiagonal

Na primeira etapa, os elementos da diagonal inferior são eliminados e os da diagonal principal normalizados para a unidade, ficando a matriz como na figura 3.5.

$$\begin{pmatrix} 1 & c_1' & & & & & \\ & 1 & c_2' & & & & \\ & & . & . & & & \\ & & & 1 & c_i' & & \\ & & & & . & . & \\ & & & & & 1 & c_{n-1}' \\ & & & & & & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ . \\ x_i \\ . \\ . \\ x_n \end{pmatrix} = \begin{pmatrix} d_1' \\ . \\ . \\ d_i' \\ . \\ . \\ d_n' \end{pmatrix}$$

FIGURA 3.5 - Sistema Linear resultante após o *forward* do AT

Para a primeira equação do sistema, tem-se $c_1' = \frac{c_1}{b_1}$ e $d_1' = \frac{d_1}{b_1}$. Para as demais:

$c_i' = \frac{c_i}{b_i - a_i c'_{i-1}}$, com $i = 2...(n-1)$ e $d_i' = \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}$ com $i = 2...n$.

A etapa da retrossubstituição é dada por $x_n = d_n'$ e $x_i = d_i' - x_{i+1} c_i'$ com $i = (n-1)...1$.

O Algoritmo de Thomas seqüencial pode ser visto no Anexo 1. Esse método sempre encontrará a solução do sistema linear, se o mesmo for diagonalmente dominante [FLE88], ou seja, $|b_i| > |a_i| + |c_i|$, com i=2..(n-1) e $|b_1| > |c_1|$ e $|b_n| > |a_n|$.

Os métodos diretos em geral são usados por serem genéricos e robustos. A paralelização desses métodos não tem alcançado resultados satisfatórios e, segundo Kumar [KUM94], por duas razões:

a) a quantidade de computação em relação ao tamanho do problema é muito pequena;

b) a maioria das tentativas de paralelizar esses métodos é baseada em formulações seqüenciais muito eficientes.

Kumar ainda acrescenta que a minimização da memória utilizada e do número de operações, objetivos de algoritmos seqüenciais, podem não ser apropriados para o caso paralelo. O desenvolvimento de eficientes formulações paralelas para métodos diretos é

uma área atual, onde alguns pesquisadores, como Duff [DUF99] e Dongarra [DON2000]. têm se dedicado.

Apesar do Algoritmo de Thomas seqüencial ser muito eficiente [FLE88, AND95, WES68], há somente uma quantidade limitada de paralelismo em sua formulação. Mesmo assim, pode ser desejável executá-lo em um ambiente paralelo devido, por exemplo, à quantidade de memória disponível em um único processador ser insuficiente para acomodar o problema inteiro.

3.4 Método Iterativo de Resolução de Sistemas Lineares

Os métodos iterativos de resolução de sistemas lineares têm sido bastante utilizados nos últimos anos, pois no trabalho com matrizes esparsas de grande porte, possibilitam otimizações no armazenamento e proporcionam a resolução de forma eficiente.

Particularmente, para sistemas do tipo banda e diagonalmente dominantes, os métodos dessa classe são bons, convergindo rapidamente [WES68]. Ao contrário dos métodos diretos que resolvem o sistema por um número pré-definido de passos, os iterativos calculam gradualmente a solução, até um critério de parada ser alcançado. Também, por essa característica é que são preferidos, em relação aos diretos, na resolução de grandes sistemas.

Os algoritmos iterativos, em geral, fazem uma aproximação inicial da solução do sistema e a cada nova iteração, uma nova aproximação conforme as regras do método. As iterações chegam ao fim quando o critério de parada é satisfeito.

Os métodos iterativos podem ser estacionários, também chamados clássicos, ou não estacionários. Nos estacionários, cada iteração não envolve informações da iteração anterior e manipulam componente a componente (variáveis) do sistema linear durante a resolução, através de operações elementares entre linhas e colunas da matriz. Alguns exemplos são Jacobi, Gauss-Seidel e SOR [DIV90, CLA94]. Os não estacionários trabalham sob a ótica da minimização da função quadrática ou por projeção, manipulando os vetores e matrizes inteiros, e incluem hereditariedade em suas iterações por exemplo, o CG, a cada iteração, calcula um resíduo que é usado na iteração subsequente. Outro exemplo de método não estacionário é o GMRES.

Como se está trabalhando com máquinas digitais, as quais possuem aritmética finita de ponto flutuante, os critérios de parada das iterações podem ser [DIV90]:

a) número máximo de iterações;

b) solução aproximada ter um número mínimo de algarismos significativos corretos;

c) limite de tolerância do erro.

No caso c), para os métodos estacionários, pode-se adotar que o erro de cada componente (incógnita) seja menor, em módulo, que um determinado valor. Já para os métodos não estacionários, trabalha-se com a norma do resíduo, conforme será detalhado no método do Gradiente Conjugado.

Para matrizes em geral, os métodos não estacionários não estão tão bem compreendidos como os estacionários, apesar de nos últimos anos ter ocorrido um avanço significativo, tanto na teoria quanto na aplicação dos mesmos. No caso

específico de matrizes simétricas, positivas e definidas (SPD), o grau de compreensão é maior, principalmente na teoria [SAA96]. O Gradiente Conjugado é um desses métodos.

Nas próximas seções serão apresentados os fundamentos do Gradiente Conjugado. A maioria das definições foram baseadas em [SHE94], pois apesar de Saad [SAA96] e Kelley [KEL95] proporcionarem um embasamento teórico completo, tem uma abordagem muito complexa, enquanto Shewchuk [SHE94] diferencia-se por explicar a teoria aliada à interpretação geométrica.

3.4.1 Gradiente Conjugado

O método do Gradiente Conjugado (CG) é um método iterativo não estacionário e considerado como um dos métodos iterativos mais eficientes para resolução de sistemas lineares de grande porte e esparsos [SHE94, KEL95, SAA96]. O CG enquadra-se na classe de métodos do subespaço de Krylov [SAA96], pois se baseia na minimização da função quadrática e não na intersecção de hiperplanos como os métodos iterativos clássicos. Ser um método não estacionário implica que o CG herda informações das iterações anteriores e as considera para a realização de cada iteração.

O Gradiente Conjugado parte do princípio de que o gradiente, que é um campo vetorial, aponta sempre na direção mais crescente da função quadrática. Como será visto nas próximas seções, o gradiente da função quadrática, se a matriz é simétrica, positiva e definida (SPD) [SHE94], resume-se à solução do sistema de equações lineares.

O gradiente aponta na direção “mais” crescente da função. Geometricamente, isso significa que na base da parabolóide formada pela função quadrática, o gradiente é zero, que é a solução do sistema linear.

De forma simples, a idéia do método do Gradiente Conjugado é ir dando passos, em cada iteração, na direção oposta a do gradiente (campo vetorial), de tal forma que a direção já pesquisada não seja repetida, até encontrar o mínimo estrito e global. A minimização ocorre sobre certos espaços de vetores, chamados de subespaço de pesquisa (espaços de *Krylov*), gerados a partir dos resíduos de cada iteração. O mínimo estrito e global de uma função quadrática ocorre na solução do sistema linear, como provado matematicamente em Slaviero [SLA97].

Esse método, em seu algoritmo, possui operações entre vetores e matrizes, como por exemplo soma e produto escalar de vetores e multiplicação matriz vetor. Essa última terá uma influência maior no desempenho do método, devido a sua complexidade.

Para o melhor entendimento do CG, são vistos alguns conceitos como forma quadrática e métodos base para o CG como o método do *steepest descent* (SD) e o das Direções Conjugadas (CD).

3.4.1.1 Forma Quadrática

A forma quadrática é um escalar, função quadrática de um vetor, conforme equação (3.1). Nessa, *A* é uma matriz, *b* e *x* vetores e *c* é um escalar.

O gradiente $f'(x)$ da forma quadrática é um campo vetorial que para um dado *x*, aponta para a direção “mais” crescente da função quadrática. O Gradiente é dado pela equação (3.2).

$$f(x) = \frac{1}{2} x^T Ax - bx + c \tag{3.1}$$

A representação gráfica dessa função, para um sistema de ordem dois (um vetor de duas posições), gera uma parabolóide (figura 3.6), em cuja base o gradiente é zero, ou seja, $f(x)$ é minimizada quando $f(x) = 0$. Se a matriz A é simétrica $A = A^T$, o gradiente da forma quadrática reduz-se a (3.3) que minimizado é justamente a solução do sistema linear:

$$f'(x) = \frac{1}{2} A^T x + \frac{1}{2} Ax - b \tag{3.2}$$

$$f'(x) = Ax - b = 0 \tag{3.3}$$

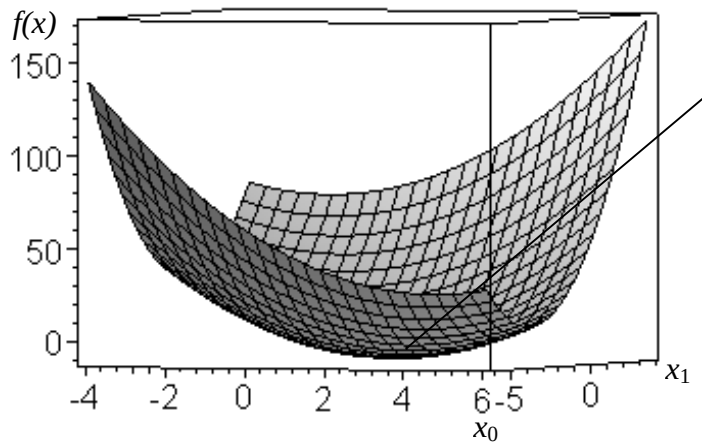


FIGURA 3.6 - Parabolóide gerada pela Função Quadrática

3.4.1.2 Método do *Steepest Descent*

O método do *Steepest Descent* (SD) é um método iterativo que serve como base de entendimento para o método do CG. Ele inicia a partir de um ponto arbitrário x^0 e vai descendo até a base da parabolóide, ou seja, é gerada uma seqüência de passos x^1, x^2, ..., x^n até que a aproximação atinja a exatidão desejada. Para a tomada de um passo, é escolhida a direção em que *f* diminui mais rapidamente e, como o gradiente aponta para a direção crescente da parabolóide, a escolha é feita na direção oposta do gradiente: $-f'(x^i) = b - Ax^i$.

Algumas definições são essenciais para o melhor entendimento do método, como:

a) erro: é um vetor que indica quão distante se está da solução.

$$e^i = x^i - x$$

b) resíduo: é um vetor que indica quão distante se está do correto valor de b.

$$r^i = b - Ax^i$$

O resíduo também pode ser representado como a direção de decrescimento (steepest descent):

$$r^i = -f'(x^i)$$

c) nova aproximação: a aproximação do vetor *x* de incógnitas, na iteração *i* (x^i) é feita na direção do *steepest descent* (r^{i-1})

$$x^i = x^{i-1} + \alpha r^{i-1}$$

O resíduo é usado para construir as direções de pesquisa. Cada nova aproximação herda informações da anterior pelos resíduos. A questão é determinar o tamanho do passo (α) a ser dado. A direção de procura escolhe um α que minimiza *f* e o gradiente do x^i é ortogonal ao gradiente do x^{i-1}.

Em suma, a iteração do método do *Steepest Descent* é dada a seguir, pelo resíduo, tamanho do passo e pela aproximação de *x*. Maiores detalhes podem ser vistos em [SHE94].

Resíduo: $$r^i = b - Ax^i \quad (3.4)$$

Tamanho do Passo: $$\alpha^i = \frac{r^i r^i}{r^i A r^i} \quad (3.5)$$

Aproximação de *x*: $$x^{i+1} = x^i + \alpha^i r^i \quad (3.6)$$

A equação (3.4) pode ser substituída pela (3.7). A demonstração dessa equivalência é encontrada em [SHE94]. No algoritmo, essa substituição elimina uma multiplicação matriz vetor, já que o produto Ar^i é utilizado em duas equações (3.5 e 3.7). Contudo, há a desvantagem da solução ser calculada sem *feedback*, ficando sujeita a erros ainda maiores. Para evitar isso, periodicamente pode-se usar a equação (3.4) para corrigir o resíduo, amenizando esse problema.

$$r^{i+1} = r^i - \alpha^i A r^i \quad (3.7)$$

3.4.1.3 Método das Direções Conjugadas

O método do *Steepest Descent*, no decorrer da resolução, faz vários passos na mesma direção. O ideal seria a execução de um único passo em determinada direção e, preferencialmente, na direção certa, durante as iterações. Isto é feito pelo método das direções conjugadas (CD).

Para que seja executado somente um passo em cada direção, utiliza-se para gerar a direção de pesquisa, o conjunto de direções de pesquisa ortogonais ($d^0, d^1, ..., d^{n-1}$). Cada passo será do tamanho exato para alinhar-se com *x*, fazendo com que após *n* passos, o método chegará à aproximação correta da solução.

Em cada passo, se escolhe um ponto, conforme (3.8), onde α^i é o tamanho do passo e d^i é a direção de pesquisa.

$$x^{i+1} = x^i + \alpha^i d^i \quad (3.8)$$

Como o erro da iteração atual é ortogonal à direção anterior, isso é utilizado para que o α seja encontrado:

$d^i e^{i+1} = 0$, usando 3.8 para o erro, tem - se

$$d^i \left(e^i + \alpha^i d^i\right) = 0 \quad \Rightarrow$$

$$d^i e^i + \alpha^i d^i d^i = 0 \quad \Rightarrow \qquad (3.9)$$

$$\alpha^i = -\frac{d^i e^i}{d^i d^i}$$

Com essa equação, para calcular α, se faz necessário o erro (e) que ainda não foi encontrado. Se já estivesse sido, a solução já seria conhecida e não seria necessário encontrar α. Para resolver esse impasse, uma alternativa é fazer as direções de pesquisa serem A-ortogonais ao invés de ortogonais.

Dois vetores d^i e d^j são A-ortogonais ou conjugados se $d^i A d^j = 0$. Deseja-se que e^i seja A-ortogonal a d^i ($e^i A d^i = 0$). Essa condição de ortogonalidade é equivalente a encontrar o ponto mínimo ao longo da direção d^i, como no método do *Steepest Descent*. [SHE94]

Assim, α pode ser calculado sem o erro ser conhecido, através de (3.10). Observa-se que se o vetor direção (d) fosse o resíduo (r), essa fórmula seria idêntica à usada pelo método SD.

$$\alpha^i = \frac{d^i r^i}{d^i A d^i} \qquad (3.10)$$

A criação das direções conjugadas ou A-ortogonais (d^is) é feita através do processo de conjugação de Gram-Schmidt [SHE94, SAA96]. Porém, há uma dificuldade em utilizar esse processo, uma vez que todos os vetores de direção precisam ser armazenados para a construção de um novo (isso requer $O(n^3)$ operações) [SHE94]. O uso do método das Direções Conjugadas é pequeno, uma vez que o método do Gradiente Conjugado é o próprio CD sem esses problemas.

3.4.1.4 De volta ao Gradiente Conjugado

O método do Gradiente Conjugado é o método das Direções Conjugadas, utilizando a conjugação dos resíduos para construir os vetores de pesquisa (d^is). Essa escolha tem algumas razões. Como os resíduos foram usados no método SD e não no DC e como o resíduo é ortogonal à direção anterior, garante-se uma nova e linearmente independente direção a não ser quando o resíduo é zero, mas nesse caso, a solução já foi encontrada.

A definição dos d^is implica em construir um subespaço pela combinação linear dos resíduos:

$$d^i = gerado\left\{r^0, r^1, ..., r^{i-1}\right\} \qquad (3.11)$$

Como cada resíduo é ortogonal à direção anterior, ele também é ortogonal ao resíduo anterior, então, $r^i r^j = 0,\ i \neq j$. Logo, no CG cada novo resíduo é ortogonal a todos os resíduos anteriores e direções. Aqui, tem-se a herança de informações das iterações anteriores, o que torna o método do Gradiente Conjugado um método não estacionário.

Cada nova direção é construída, como em (3.12), a partir do subespaço dos resíduos, para ser A-ortogonal a todos os resíduos e direções anteriores. Isso garante a não repetição da direção de pesquisa.

$$d^i = r^i + \beta d^{i-1} \qquad (3.12)$$

onde β é a razão entre as normas do resíduo r^i e r^{i-1}.

Dessa forma, não é necessário armazenar vetores de direção antigos para garantir a A-ortogonalidade dos novos vetores, como no método das Direções Conjugadas. Esse é o maior avanço e faz com que o Gradiente Conjugado seja um algoritmo eficiente em termos de complexidade de tempo e armazenamento por iteração.

Resumindo, o método do Gradiente Conjugado é dado por:

a) Tamanho do passo α:

$$\alpha^i = \frac{r^i r^i}{d^i A d^i} \tag{3.13}$$

b) Razão entre as normas do resíduo β, usada para calcular a nova direção de pesquisa.

$$\beta^{i+1} = \frac{r^{i+1} r^{i+1}}{r^i r^i} \tag{3.14}$$

c) Resíduo *r*. Na primeira iteração a direção de pesquisa é igual ao resíduo ($d^0 = r^0$). A cada iteração o resíduo é calculado conforme (3.15), mas isso requer uma multiplicação matriz vetor a mais no algoritmo, então esse cálculo é feito somente a cada *k* iterações, sendo substituído por (3.16), porque o *Ad* já foi calculado, diminuíndo o número de operações do algoritmo.

$$r^i = b - Ax^i \tag{3.15}$$

$$r^{i+1} = r^i - \alpha^i A d^i \tag{3.16}$$

d) Aproximação do valor de x^{i+1}

$$x^{i+1} = x^i + \alpha^i A d^i \tag{3.17}$$

e) Cálculo da nova direção de pesquisa d^{i+1}, que é a combinação linear do resíduo com a direção anterior.

$$d^{i+1} = r^{i+1} + \beta^{i+1} d^i \tag{3.18}$$

O algoritmo do Gradiente Conjugado envolve operações básicas entre matriz e vetores. Isso é fator favorável à paralelização do mesmo. No algoritmo do CG, como entrada tem-se a matriz dos coeficientes *A*, o vetor *b*, uma aproximação inicial de x^0, o número máximo de iterações i_{max} e a tolerância do erro $\varepsilon < 1$ (fração da norma do resíduo), que na prática $\varepsilon \cong 10^{-8}$. Esse algoritmo está no Anexo 1.

3.4.1.5 Convergência do Método

O Gradiente Conjugado converge no máximo depois de *n* passos, mas devido à acumulação de erros de arredondamento, gradativamente perde-se a exatidão. A primeira iteração do CG é idêntica à do SD e suas condições de convergência são as mesmas. A taxa de convergência geral pode ser definida como a norma da energia [SHE94]:

$$\|e\|_A = \sqrt{eAe}$$

Para $k = \frac{\lambda_{\max}}{\lambda_{\min}}$, onde $\lambda_{\max}$ e $\lambda_{\min}$ são respectivamente o maior e o menor autovalor, pode-se encontrar em [SAA96, SHE94] , que a convergência é dada por:

$$\left\| e^i \right\|_A \leq 2\left(\frac{k-1}{k+1} \right)^i \left\| e^0 \right\|_A \tag{3.19}$$

3.4.1.6 Início e Término do Método

Na primeira aproximação do Gradiente Conjugado, quando não se tem uma boa estimativa para o x^0, faz-se a aproximação inicial igual a zero ($x^0 = 0$). Como critério de parada das iterações utiliza-se a norma do resíduo: se a norma do resíduo r^i , que é a distância da solução do sistema, for menor que uma determinada fração ε da norma do resíduo inicial r^0 (3.19) ou um número máximo de iterações for atingido, a iteração é terminada.

$$\left\| r^i \right\| < \varepsilon \left\| r^0 \right\| \tag{3.20}$$

3.4.2 Gradiente Conjugado Pré-condicionado

A idéia de pré-condicionamento não se limita a sistemas lineares. Sistemas não lineares, problemas de otimização e problemas de autovalores são algumas das áreas em que os pré-condicionadores são indispensáveis [ARA97]. Essa técnica é usada para aumentar a convergência do método. Ela consiste em transformar o sistema em um outro com a mesma solução porém com propriedades mais favoráveis à convergência.

O método do CG, se a matriz dos coeficientes *A* possui *n* autovalores distintos, converge em no máximo *n* iterações, supondo não haver erros de arredondamento. Se a matriz *A* possui muitos autovalores distintos que variam largamente em magnitude, esse método provavelmente convergirá necessitando de um grande número de iterações.

Essa convergência pode ser acelerada pelo pré-condicionamento da matriz *A*. A matriz pré-condicionadora é escolhida de tal forma que $A' = CAC^T$, onde *C* é não singular e dita pré-condicionador [KUM94]. O método CG com pré-condicionador é chamado Gradiente Conjugado Pré-condicionado (PCG).

Podem acontecer alguns problemas na aplicação do pré-condicionador em um sistema $Ax = b$. Um deles é a não preservação da esparsidade da matriz dos coeficientes, que dificultará a utilização de esquemas de armazenamento da matriz esparsa, e outro, é que as multiplicações matriciais envolvidas com o pré-condicionador podem tornar-se caras.

A taxa de convergência do CG depende da distribuição dos autovalores da matriz *A*. Essa taxa aumentará se houver um agrupamento de autovalores em torno da unidade. O pré-condicionador é usado para que a matriz dos coeficientes tenha seus autovalores com essas características [CUN92].

Há vários tipos de pré-condicionadores que podem ser encontrados em [SAA96, KEL95]. Contudo, com a utilização de paralelismo podem, algumas vezes, tornarem-se instáveis numericamente e não muito propícios para esse tipo de processamento, devido à natureza de suas formulações [CUN92]. Pode-se citar como exemplo desses pré-condicionadores *Cholesky* Incompleto e Fatoração LU.

Essas fatorizações são mais usadas para processamento vetorial e máquinas com memória compartilhada. Para o caso de máquinas com memória distribuída, a performance não é muito boa, segundo Holter, citado por [CUN92]. Assim, a escolha de um pré-condicionador, depende também da arquitetura que está sendo usada.

O problema permanece em encontrar um pré-condicionador que aproxima *A*, bem o suficiente para possibilitar uma convergência que compense o custo de calcular o produto $C^{-1}A$ em cada iteração. O pré-condicionamento tenta fazer com que a forma quadrática pareça, geométricamente, mais esférica, ou seja, os autovalores estejam fechados uns com os outros.

No algoritmo PCG, além dos dados de entrada do algoritmo do CG, tem-se o pré-condicionador C. Esse algoritmo está no Anexo 1. Nessa pesquisa, o Gradiente Conjugado será pré-condicionado com dois pré-condicionadores: o Diagonal (ou de Jacobi) e o Polinomial.

3.4.2.1 Pré-condicionador Diagonal

O mais simples pré-condicionador é o pré-condicionador Diagonal (ou Jacobi). Nesse, a matriz pré-condicionadora *C*, é uma matriz diagonal e seus elementos correspondem aos elementos da diagonal principal da matriz dos coeficientes *A*.

Para o cálculo do pré-condicionador, considera-se somente a diagonal da matriz *A* do sistema. Como no algoritmo PCG, quer-se a inversa de *C*, ou seja, C^{-1}, o processo de inversão fica simplificado, resumindo-se a $c_{ij}^{-1} = 1/a_{ij}$.

O pré-condicionador diagonal pode ser obtido também através do pré-condicionador polinomial, quando o grau do polinomio for igual a *zero*. Nessa pesquisa, o pré-condicionador diagonal é obtido conforme descrito nessa seção.

3.4.2.2 Pré-condicionador Polinomial

O pré-condicionador polinomial tem recebido atenção nos últimos anos, porque pode ser paralelizado eficientemente em arquiteturas com memória distribuída, através de operações paralelas de álgebra linear [CUN92, DUF99]. Um pré-condicionador polinomial é expresso em termos de polinômios da matriz dos coeficientes.

Há várias formas de se obter pré-condicionadores polinomiais. Uma delas, apresentada em [DUB79], é o pré-condicionador baseado em séries truncadas de Neumann. Nessa abordagem, a matriz *A* é particionada da seguinte forma:

$A = P - Q = P(I - P^{-1} Q)$

onde *P* é a diagonal da matriz *A*.

Como precisa-se da inversa do pré-condicionador no algoritmo PCG, segundo [DUB79], a inversa do pré-condicionador polinomial é dada em (3.21). Outros detalhes sobre o pré-condicionador polinomial estão no capítulo 5, que trata da paralelização, e no 6, onde são apresentados os resultados obtidos.

$$C^{-1} = \left(\sum_{i=0}^{m} \left(I - P^{-1}A \right)^i \right) P^{-1}; \qquad P^{-1} = \left(diag(A) \right)^{-1} \tag{3.21}$$

3.5 Bibliotecas de Resolução de Sistemas Lineares

Há várias bibliotecas de resolução de sistemas de equações lineares, disponíveis livremente, que podem ser utilizadas na computação paralela. Como a resolução de sistemas é o escopo dessa dissertação, cabe fazer uma pequena introdução a algumas bibliotecas. Maiores detalhes podem ser encontrados nas respectivas referências de cada biblioteca e em [EIJ99] que faz um panorama sobre as mesmas.

Nos últimos anos, a mudança arquitetural da computação de alto desempenho tem sido contínua e percebe-se que a dificuldade principal está em produzir *software* adequado a tais transformações. A inovação mais recente são os *clusters* que combinam memória distribuída com memória compartilhada nos nodos.

A união da programação com *threads,* utilizada em máquinas paralelas com memória compartilhada, e da programação paralela através de trocas de mensagens, utilizada em máquinas com memória distribuída, é a tendência para o desenvolvimento de *software* às novas máquinas de alto desempenho. Das muitas bibliotecas de resolução de SELAs existentes, poucas misturam paralelismo de memória compartilhada com memória distribuída [BRO2000].

TABELA 3.1 - Bibliotecas de Álgebra Linear – Informações Básicas

Biblioteca	**Autores**	**Linguagem**	**Paralelismo**	**Última Versão**
Aztec [SAN2000]	Scott Hutchinson, John Shadid, Ray Tuminaro, Mike Heroux	C com MPI	Sim	1999
IML++ [IML2000]	Jack Dongarra, Andrew Lumsdaine, Roldan Pozo e Karin Remington.	C++	Paralelismo é possível, mas de responsabilidade do usuário	1996
Itpack [EIJ99]	David R. Kincaid, Thomas C. Oppe e David M. Young.	Fortran	Possui versão vetorial	1997
Laspack [SKA2000]	Tomás Skalický	C	Não	1996
PCG [JOU2000]	W.D.Joubert, G.F.Carey, NA Berner, A. Kalhan, H. Khli, A. Lober, RT Mclay e Y. Shen.	Fortran	Seqüencial; uma versão para Cray Y-MP e uma versão MPI em construção.	1996
PIM [CUN2000]	Rudinei Dias da Cunha e Tim Hopkins	Fortran com MPI	Sim	1997
P_Sparslib [SAA2000]	Yousef Saad, Andrei V. Maleosky, Sergey Kuznetsov, Masha Sosonkina, Irene Moulista e Gen-Ching Lo	Fortran com MPI	Sim	1999
Splib [EIJ99]	Randall Bramley e Xiaoge Wang	Fortran	Não	1999
LINPACK [LIN2000]	J. Dongarra, J. Bunch, C. Moler, P. Stewart	Fortran	Supercomputadores déc. 1970 e 1980	1984
LAPACK [LAP2000]	J. Dongarra, E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen.	Fortran	Sucessora da LINPACK para máquinas com memória compartilhada e distribuída.	1999
ScaLAPACK [SCA2000]	L.S.Blackford, J.Choi, A.Clearly, E.D'Azevedo, J. Demmel, I.Dhillon, J.Dongarra, S.Hammarling, G.Henry, A.Petitet, K.Stanley, D.Walker, R.C.Whaley	Fortran com PVM e/ou MPI	Para arquiteturas MIMD.	1997

Dentre as bibliotecas relacionadas nessa seção, a LAPACK mescla os dois paradigmas de programação citados anteriormente. Nas tabelas 3.1 e 3.2, cada biblioteca é caracterizada sob os seguintes aspectos:

a) tipo de método de resolução usado;

b) tipo de pré-condicionador usado;

c) execução paralela;

d) estrutura de dados utilizada para o trabalho com matrizes esparsas;

e) linguagem em que foi implementada.

Para ser alcançada a computação transparente em *clusters* são necessários algoritmos que adaptem-se apropriadamente às configurações das máquinas. Combinar modelos de memória compartilhada com modelos de memória distribuída é uma forte tendência à obtenção de desempenho nessas arquiteturas. Frente a isso, prevê-se o surgimento de novas bibliotecas ou a reformulação das mesmas como uma progressão natural do *software* a fim de disponibilizar bibliotecas numéricas para computação em *clusters.*

TABELA 3.2 - Bibliotecas de Álgebra Linear – Principais Características

Biblioteca	**Métodos de Resolução**	**Pré-condicionadores**	**Estrutura de Dados**
Aztec	CG, GMRES, CGS, BiCGstab	Jacobi com ILU em subblocos e Schwarz.	Linha Comprimida e Linha de bloco variável. Sendo desenv. interface livre de matriz.
IML++	BiCG, CGstab, CG, GMRES, CGS, QMR.	Jacobi, ILU e Cholesky Incompleto	Linha/coluna comprimida e formato coordenado
Itpack	CG, SOR, SSOR dentre outros métodos iterativos	Jacobi, SOR, SSOR.	Linha comprimida e formato *ellpack*
Laspack	Jacobi, SOR, SSOR, CG, GMRES, BiCG dentre outros e métodos *multigrid*.	Jacobi, SSOR, ILU(0)	Possui vários tipos, mas não foram encontrados.
PCG	Métodos iterativos não estacionários	Richardson, Jacobi e Polinomial	Armazenamento por grade regular.
PIM	CG, BiCG, CGS, BiCGSTAB, RBi-CGSTAB, GMRES, GCR dentre outros iterativos.	Fatorização LU, $ILU^{(0)}$, $IDLU^{(0)}$, Polinomial	Usuário define as estruturas de dados. Flexibilidade.
P_Sparslib	GMRES, CG, BiCG, BiCGstab dentre outros iterativos.	Schwarz, Complemento de Schur	Não documentado.
Splib	Métodos iterativos estacionários e não estacionários	ILU, SSOR, ILU(0)	Linha comprimida.
LINPACK	Decomposição QR, Cholesky, Eliminação de Gauss	–	Linha comprimida.
LAPACK	Fatorização LU, Cholesky e suas variantes.	–	*Packed* (por colunas em um array unidimensional), formato diagonal e por vetor.
ScaLAPACK Utiliza pacotes: BLAS, PBLAS, LAPACK e BLACS.	Fatorização LU, Cholesky e suas variantes. Utiliza a LAPACK, mas há diferenças em alguns algoritmos.	–	Como em LAPACK.

3.6 Considerações Finais

O objetivo desse capítulo é proporcionar uma visão geral de sistemas lineares, especificamente os tridiagonais e descrever como os métodos escolhidos de cada classe trabalham para a resolução. Procurou-se introduzir a terminologia utilizada e especificar o funcionamento de cada método para uma melhor compreensão de sua paralelização, que será abordada no capítulo cinco.

Buscou-se proporcionar uma visão geral de bibliotecas de álgebra linear disponíveis. Cabe lembrar que o objetivo dessa pesquisa é a investigação de formas de paralelizar métodos de resolução de sistemas lineares e realizar uma avaliação do desempenho desses, no *cluster*, utilizando o ambiente de programação paralela DECK. A maioria dessas bibliotecas é fruto de anos de pesquisa de grupos reconhecidos mundialmente na comunidade científica, como o grupo do Prof. Dr. Jack Dongarra, que elaborou a biblioteca LINPACK e suas sucessoras [DON2000].

Com relação aos métodos de resolução, percebe-se desde já que os métodos diretos, por sua natureza, têm a tendência de gerar algoritmos vetorizáveis e o maior gargalo com essa forma de construção desses algoritmos é o pouco paralelismo existente. Esse pode ser o fator originário de muitos esforços na paralelização desses métodos e pode ser verificado no capítulo 6, onde estão alguns resultados. Já os métodos iterativos são geralmente paralelizados por suas operações de álgebra linear e como os computadores podem manipular os produtos internos de uma forma a se obter bons resultados, isso é fator positivo à paralelização dos mesmos.

No próximo capítulo é descrito o ambiente paralelo onde foram avaliados os algoritmos dos métodos de resolução dos sistemas lineares. Nesse sentido são abordados aspectos da arquitetura e do ambiente de programação paralela.

Apresentou-se também, nesse capítulo, o Gradiente Conjugado pré-condicionado. No capítulo 6, conseguir-se-á observar em que situações é, ou não, vantajosa a utilização de pré-condicionadores, bem como aspectos relacionados à otimização do armazenamento e operações extras necessárias ao algoritmo pré-condicionado.

4 Ambiente Paralelo

A utilização de computação de alto desempenho tem sido motivada não só por simulações numéricas de sistemas complexos (tempo, clima, circuitos eletrônicos...), mas, atualmente, também por aplicações comerciais, que manipulam uma grande quantidade de dados a serem processados. Essas aplicações, segundo Foster [FOS95], incluem vídeo conferência, diagnósticos médicos auxiliados por computador, bancos de dados paralelos, realidade virtual, computação gráfica, dentre outras.

Esse capítulo tem por objetivo apresentar o ambiente paralelo onde os algoritmos dos métodos de resolução de sistemas lineares foram paralelizados. Essa abordagem envolve alguns aspectos relacionados à arquitetura e outros relacionados à programação paralela.

Inicialmente, é comentada a evolução das máquinas até se chegar aos *Clusters*. Após, apresentada uma visão geral de máquinas paralelas através da taxonomia feita por Flynn [FLY72]. A arquitetura genérica de *Clusters* e o *Cluster* utilizado são caracterizados.

Com relação à programação paralela, paradigmas de programação e uma metodologia de desenvolvimento dos algoritmos paralelos são identificados. Por fim, o DECK - *Distributed Execution and Communication Kernel*, ambiente usado para a programação, é descrito.

4.1 Arquitetura Paralela

Há séculos a ciência tem seguido um paradigma básico: primeiro observar, teorizar e então testar a teoria através da experimentação. Em certos casos, ou na maioria deles, a experimentação “real” tem um custo muito elevado ou é inviável. Esse fato faz com que a utilização de simulação numérica através do computador tenha aumentado nos últimos anos [PAC97], sendo aplicada também na indústria e comércio, além de nas já tradicionais aplicações de alto desempenho como previsão do tempo, por exemplo.

A supercomputação moderna foi marcada na segunda metade da década de 70 pela introdução de computadores vetoriais. Na primeira metade da década de 80, a integração de computadores vetoriais com sistemas convencionais tornou-se mais importante, pois o desempenho foi aumentado pelo aperfeiçoamento dos *chips* e pela produção de computadores multiprocessados com memória compartilhada. No final da década de 80, as máquinas paralelas dedicadas (também conhecidas como MPP – *Massively Parallel Processors*) com memória distribuída emergiram e na metade da década de 90 foram superadas por sistemas com multiprocessadores simétricos. A partir de então, graças a fatores como o desenvolvimento de redes locais de alta velocidade e baixo custo, formou-se a base para a atual utilização de *clusters* [STR99].

Um computador paralelo é um conjunto de processadores capazes de trabalhar cooperativamente para resolver um problema. Um *cluster* é um tipo de computador paralelo e consiste de um conjunto de computadores interligados por uma rede de alta velocidade, para possibilitar a execução de códigos em paralelo [RIG99]. Reúnem-se pequenas máquinas para fazer o trabalho de máquinas grandes, dado que os microprocessadores tem aumentado muito a capacidade de processamento. Na lista dos TOP500, cujo *benchmark* é realizado com a biblioteca LINPACK, percebe-se a

tendência crescente de utilização de *clusters* e o desempenho obtido com eles [TOP2000].

C*lusters* tornaram-se conhecidos com um projeto da NASA [SIL99], em 1994. A questão a ser resolvida era a existência de muitos dados para serem processados e somente U$ 50.000,00 disponíveis para tal aplicação. A solução encontrada foi agrupar várias máquinas e fazer a aplicação ser executada nelas, de forma paralela. Essa idéia foi positiva principalmente porque foi, e é, uma alternativa ao processamento de alto desempenho, pela redução de custos, tanto pelo *hardware* quanto pelo *software* (existem sistemas operacionais, linguagens e bibliotecas de comunicação gratuitas).

4.1.1 Classificação de *Flynn*

Michael Flynn [FLY72] apresentou uma classificação das máquinas, em 1972, conforme o fluxo de instruções e o fluxo de dados. Nessa, são encontradas quatro classes de computadores SISD, SIMD, MISD e MIMD. Atualmente, essa classificação poderia ter acrescentadas outras classes e/ou subclasses, devido às várias arquiteturas disponíveis, conforme propõem alguns autores como Quinn [QUI94].

O computador de *von Neumann* é um modelo simples de máquina, característico dos computadores seqüenciais. É constituído de uma CPU, que comanda a execução de programas e executa as operações desses e uma unidade de memória, responsável por armazenar os dados e instruções. Segundo Flynn, essa máquina é enquadrada na classe SISD – *Single Instruction, Single Data*, que possui um único fluxo de dados e um único fluxo de instruções.

Computadores SIMD – *Single Instruction, Multiple Data* possuem uma CPU dedicada ao controle e uma grande quantidade de elementos de processamento com sua própria memória local, caracterizando um sistema com um único fluxo de instruções e múltiplos fluxos de dados. Durante cada ciclo de instrução, o processador fornece uma instrução aos elementos de processamento, que a executam com os dados que possuem na memória. É um processamento completamente síncrono. As máquinas SIMD podem reduzir a complexidade do *hardware* e do *software*, mas são apropriadas a aplicações com alto grau de regularidade, como por exemplo, processamento de imagens e simulações numéricas, especialmente as que fazem uso de operações de álgebra linear.

Os computadores MISD – *Multiple Instruction, Single Data* possuem diversos fluxos de instruções sobre um único fluxo de dados. Para a maioria dos autores, não existem máquinas reais que se enquadram nessa classe.

Nos computadores MIMD – *Multiple Instruction, Multiple Data* os vários processadores são independentes, ou seja, executam seus próprios processos sobre seu conjunto de dados, e não há um *clock* global para sincronismo. Esses processadores quando compartilham a memória são chamados multiprocessadores e quando possuem memória distribuída, multicomputadores [PAC97].

Um multicomputador é constituído por um número de computadores de *von Neumann* ou nodos ligados por uma rede de interconexão. Cada computador executa seu próprio programa, pode acessar sua própria memória e enviar/receber mensagens pela rede para/de outro nodo [FOS95].

Conforme [HWA98], os modelos de arquitetura paralelas estão concentrados no modelo MIMD. Há vários categorias dentro desse: Máquinas Vetoriais (PVP – *Parallel Vector Processors*), Multiprocessadores Simétricos (SMP – *Symmetric Muliprocessors*),

Máquinas Massivamente Paralelas (MPP – *Massively Parallel Processors*), Multiprocessadores com Memória Compartilhada (DSM – *Distributed Shared Memory Multiprocessors*) e *Clusters* (COW – *Clusters of Workstation*). São abordados a seguir as MPP e COW, devido ao enfoque desse trabalho.

As MPP são máquinas com diversos processadores interligados através de uma rede de interconexão, normalmente proprietária. Nos nodos há microprocessadores, podendo esses serem expandidos a milhares desses nós, ou mais. Já os *Clusters* – COW são uma variação de baixo custo das MPPs. Essas máquinas possuem um conjunto de microprocessadores interconectados por uma rede de alta velocidade, com custo menor, comparado às MPPs e há sempre um disco local, o que pode estar ausente em um nodo MPP [HWA98].

Os *Clusters* de PCs são um tipo de máquina paralela que enquadra-se, hierarquicamente, na seguinte classificação: MIMD, MPP e COW [PFI98]. Conforme o conceito de *cluster* [PFI98] visto anteriormente, esse tipo de máquina paralela em geral são multicomputadores. Alguns *clusters* podem também trabalhar com memória compartilhada, como será visto na seção 4.1.3. Esse tipo de máquina tem sido uma tendência no desenvolvimento de computadores paralelos escaláveis [PFI98].

4.1.2 Arquitetura Genérica

Um *cluster* é um tipo de sistema de processamento paralelo ou distribuído que consiste de uma coleção de computadores interconectados trabalhando juntos como um único e integrado recurso de computação [SIL99]. Os nodos de um *cluster* podem ser um simples processador ou multiprocesssadores com memória local, dispositivos de entrada/saída e sistema operacional.

Os nodos podem estar em um único gabinete ou separados fisicamente e conectados via rede local (LAN – *Local Area Network*). Para o usuário, um *cluster* representa um único sistema de processamento do qual podem ser obtidos benefícios que tem historicamente sido encontrados somente em máquinas paralelas dedicadas.

Arquiteturas baseadas em clusters podem ser homogêneas (todos os nodos que a compõem possuem a mesma arquitetura e sistema operacional) ou heterogêneas (os nodos possuem processadores diferentes e/ou sistemas operacionais diferentes). A rede que interconecta os nodos de um *cluster* geralmente é uma rede que proporciona comunicação rápida, com taxas de transmissão próximas as das arquiteturas dedicadas. A Myrinet é um exemplo de rede de alto desempenho [MYR2000]. Mesmo assim, *clusters* podem ser usados para a execução de aplicações tanto paralelas quanto seqüenciais.

Algumas vantagens na utilização de *clusters* são o desempenho que pode ser obtido, baixo custo em relação às grandes máquinas paralelas, software sem custos, facilidade de expansão e até mesmo o baixo risco, uma vez que se não der certo o *cluster,* os nodos podem ser usados como máquinas seqüenciais.

Na figura 4.1, adaptada de [SIL99], tem-se uma arquitetura genérica de *clusters.* Nessa, além dos aspectos comentados anteriormente, observa-se o *Middleware.* O *middleware* é um conjunto de *software* personalizado, desenvolvido para as necessidades específicas de um cliente e/ou para as características próprias de um determinado computador.

Nesse trabalho, o *cluster* é utilizado tanto para rodar as implementações paralelas, quanto as seqüenciais. Na próxima seção ele é descrito e a ferramenta de programação paralela será descrita no item 4.2.3.

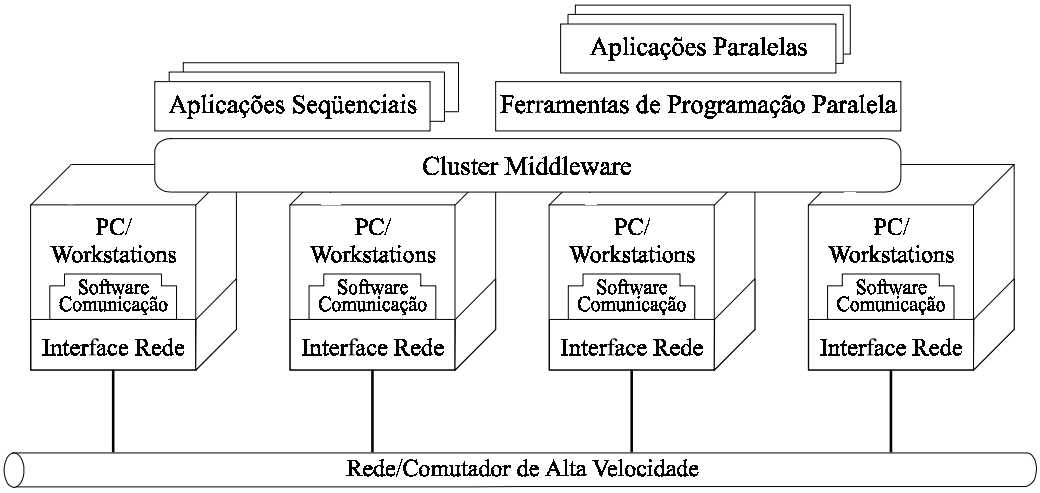


FIGURA 4.1 - Arquitetura Genérica de *Clusters*

4.1.3 O *Cluster* do Instituto de Informática

O *cluster* utilizado nessa pesquisa é o do Instituto de Informática da UFRGS que é constituído por onze PCs (onze nodos). Na tabela 4.1, tem-se a configuração das máquinas do *cluster* e na figura 4.2, uma ilustração do mesmo.

A programação paralela no *cluster* faz uso de trocas de mensagens e *threads.* A primeira é utilizada para proporcionar comunicação entre os nodos, e as *threads* são utilizadas para ser aproveitada a capacidade dos dois processadores, nos nodos Dual Pentium.

Os nodos são interconectados por duas redes de comunicação: a *Fast-Ethernet* e a *Myrinet* [MYR2000], sendo que a execução dos programas paralelos pode acontecer em qualquer uma das redes. A *Myrinet* [MYR2000] é um tipo de rede que utiliza uma tecnologia baseada na comunicação através de pacotes. É considerada uma rede de alto desempenho por ter canais robustos de comunicação com baixa latência e camadas de software que disponibilizam interfaces para mapear a rede, rotas selecionadas, tradução de endereços da rede para essas rotas, bem como manipulação de tráfego de pacotes e comunicação direta entre os processos a nível de usuário e a rede.

4.2 Programação Paralela

Para o desenvolvimento de aplicações paralelas, há necessidade de ferramentas que proporcionem esse tipo de programação. Porém, antes disso, estratégias, paradigmas e metodologias contribuem fortemente ao desenvolvimento de algoritmos paralelos e à obtenção de desempenho na aplicação.

TABELA 4.1: Configuração das Máquinas do *Cluster*

Máquina	CPU	RAM	Rede
Scliar	Pentium II 266 MHz	128 MB	*Fast-Ethernet* e *Myrinet*
Verissimo	Dual Pentium Pro 200 MHz	128 MB	*Fast-Ethernet* e *Myrinet*
Quintana	Dual Pentium Pro 200 MHz	128 MB	*Fast-Ethernet* e *Myrinet*
Dionelio	Dual Pentium Pro 200 MHz	128 MB	*Fast-Ethernet* e *Myrinet*
Euclides	Dual Pentium Pro 200 MHz	128 MB	*Fast-Ethernet* e *Myrinet*
Ostermann	Pentium Pro 200 MHz	64 MB	*Fast-Ethernet* e *Myrinet*
Luft	Pentium Celeron 300 MHz	64 MB	*Fast-Ethernet* e *Myrinet*
Braun	Pentium Celeron 300 MHz	64 MB	*Fast-Ethernet*
Meyer	Pentium Celeron 300 MHz	64 MB	*Fast-Ethernet*
Bombacha	Pentium Pro 200 MHz	32 MB	*Fast-Ethernet*
Pago	Pentium Pro 133 MHz	32 MB	*Fast-Ethernet*

Algumas estratégias de desenvolvimento de aplicações paralelas são paralelização automática do código seqüencial e utilização de bibliotecas paralelas. A primeira, livra o programador de "pensar" sobre as tarefas paralelas. A segunda, permite encapsular no programa seqüencial rotinas paralelas das bibliotecas, especialmente rotinas matemáticas. Há uma última estratégia que é a construção do código paralelo pelo programador, desde o início do algoritmo. Essa proporciona uma liberdade maior ao programador que pode escolher a linguagem e o modelo de programação a ser utilizado. Contudo é uma tarefa complexa em relação às outras estratégias [BUY99].

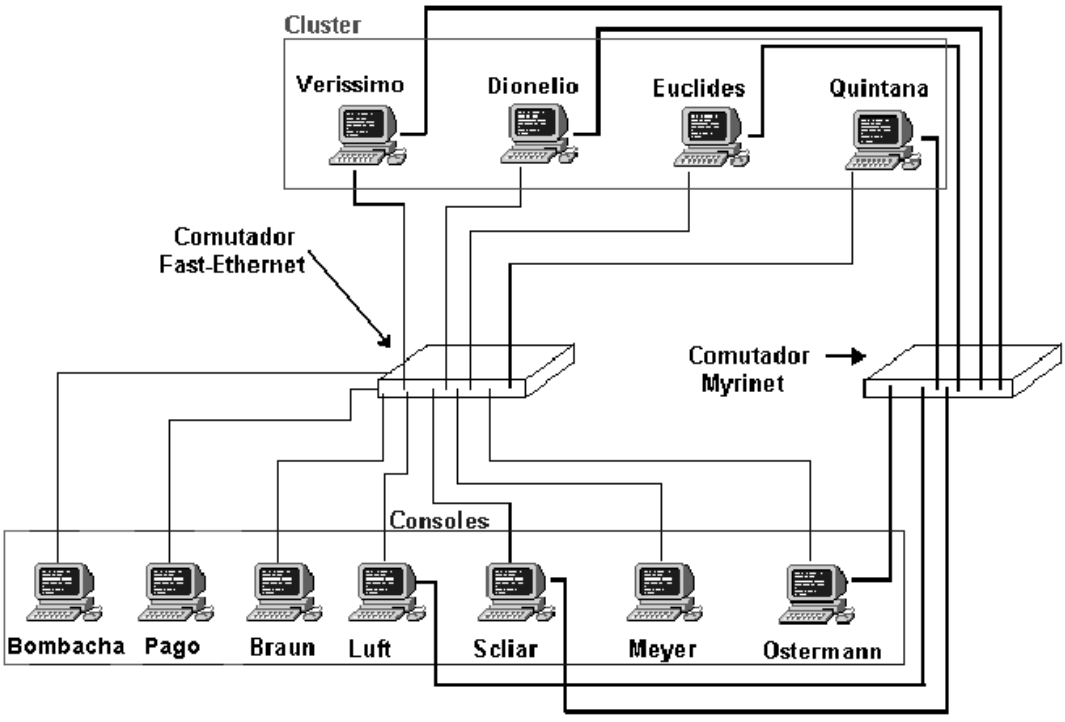


FIGURA 4.2 - Representação do *Cluster* do Instituto de Informática

Para a programação paralela, há basicamente duas aproximações: a programação implícita ou a explícita. A programação implícita pode ser diretamente relacionada com a primeira e segunda estratégias apresentadas no parágrafo anterior, onde algumas rotinas paralelas ou toda a paralelização é implícita ao programador, pois o compilador ou a biblioteca são responsáveis por isso.

Na aproximação explícita, o programador é responsável pela construção da maior parte ou, geralmente, de toda a paralelização da aplicação, como por exemplo decomposição, mapeamento das tarefas entre os processadores e formulação das comunicações. Todos os detalhes são importantes para o sucesso da paralelização e o programador precisa dar especial atenção a cada um deles.

4.2.1 Paradigmas de Programação Paralela

A paralelização de aplicações está intimamente relacionada ao tipo de arquitetura onde está sendo desenvolvida e às ferramentas que estão sendo usadas para tal. Há vários paradigmas para a programação paralela, alguns desses são Mestre/Escravo [BUY99], SPMD (*Single Program Multiple Data*) [BUY99], Pipeline e Divisão e Conquista [DIV90], MPMD (*Multiple Program Multiple Data*) [FOS95].

Como a ferramenta utilizada para proporcionar a programação paralela adota o paradigma SPMD, esse é caracterizado. O paradigma SPMD é o paradigma geralmente usado. Como o próprio nome sugere, um único programa e vários dados, cada processo executa o mesmo programa, porém sobre porções diferentes dos dados [BUY99]. Isso envolve o particionamento dos dados da aplicação entre os nodos. Esse tipo de paralelismo também é dito paralelismo geométrico, decomposição de domínio ou paralelismo de dados. Cabe ressaltar que para alguns autores, como Foster [FOS95], decomposição de domínio é a decomposição do domíno do problema, não sendo equivalente ao paralelismo de dados.

Nesse paradigma, os nodos trocam mensagens para se comunicarem. Essa comunicação pode ser local, quando a troca acontece entre os nodos vizinhos, ou global, quando envolve todos os nodos.

Aplicações com o paradigma SPMD podem ser muito eficientes se os dados estão bem distribuídos entre os nodos e o sistema é homogêneo (razão pela qual também é chamado de paralelismo geométrico). Caso contrário, para ser alcançado um bom desempenho, podem ser necessários mecanismos que façam o balanceamento da carga entre os nodos, até mesmo durante a execução da aplicação, conhecido também por balanceamento dinâmico de carga. A utilização, ou não, do balanceamento dinâmico dependerá do tipo da aplicação.

4.2.2 Metodologia de Desenvolvimento de Algoritmos Paralelos

Foster [FOS95] sugere a metodologia PCAM (*Partitioning, Communication, Agglomeration and Mapping*) ao desenvolvimento de algoritmos paralelos. Essa metodologia desconsidera, primeiramente, aspectos da arquitetura da máquina. Cada etapa é sucintamente apresentada a seguir.

A primeira etapa é o particionamento que diz respeito à decomposição de computações e dados do algoritmo em tarefas menores. Pode-se ter duas decomposições: a de dados/domínio, onde os dados ou domínio da aplicação são

divididos entre os vários nodos, e a funcional que decompõe as computações no caso da aplicação permitir a execução de computações distintas em paralelo.

A etapa de comunicação é responsável por verificar como as tarefas/dados decompostos no particionamento se comunicarão. Há várias formas de comunicação que podem ser utilizadas, por exemplo, local/global, estruturada/não estruturada, estática/dinâmica.

A comunicação local acontece quando cada nodo comunica-se com um número reduzido de nodos, geralmente seus vizinhos. Já na comunicação global todos os nodos, ou a maioria deles, participam da comunicação. Na comunicação estruturada, a comunicação entre os nodos é entre os nodos vizinhos, formando uma estrutura regular. Há também a comunicação estática, onde a identidade dos nodos não muda ao longo do processamento e a comunicação dinâmica, onde acontece o contrário [FOS95].

A aglomeração é a fase em que as tarefas e estrutura de comunicação definidas nas etapas anteriores são avaliadas em termos de desempenho e custo de implementação. Assim, se necessário, tarefas são agrupadas em tarefas maiores e comunicações locais eliminadas, por exemplo. Isso, geralmente, aumenta a granularidade da computação e diminui a comunicação, mas dependerá do tipo da aplicação, da arquitetura e do particionamento definido.

A última etapa da metodologia PCAM é o Mapeamento que consiste em designar tarefas aos nodos de tal forma que a utilização dos recursos computacionais seja maximizada e a comunicação minimizada. O mapeamento pode ser feito estático ou dinâmico, quando realizado em tempo de execução.

Na aplicação dessa pesquisa, essa metodologia de desenvolvimento de algoritmos paralelos, proposta por Foster [FOS95], foi utilizada em parte devido às características da aplicação. A aglomeração entremeou-se com as etapas de particionamento e comunicação. Para aplicações de maior porte que a realizada aqui, essa metodologia é utilizada por inteiro e suas etapas acontecem de forma evidente.

4.2.3 DECK

Essa seção destina-se a descrever a ferramenta de programação paralela utilizada, DECK - *Distributed Execution and Communication Kernel,* que está em desenvolvimento pelo GPPD – Grupo de Processamento Paralelo e Distribuído da UFRGS e faz parte do projeto *MultiCluster* [GPP2000]. O DECK é uma camada que fica entre o sistema operacional e as aplicações paralelas, como pode ser observado na figura 4.3.

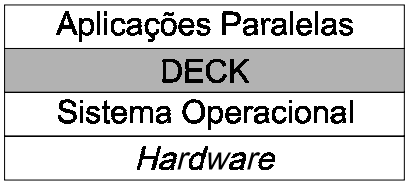


FIGURA 4.3 - Localização do DECK

DECK é um ambiente de programação que permite a construção e execução de aplicações paralelas e é composto por duas camadas: o µDECK e a camada superior [BAR2000]. O µDECK é formado pelos seguintes módulos:

a) *thread* : permite a manipulação de threads na aplicação paralela;

b) sincronização: responsável pela sincronização da aplicação entre nodos (*barrier*) e entre threads (semáforos);

c) comunicação: comunicação por troca de mensagens ponto-a-ponto, entre os nodos, através da utilização de *mail boxes*. A postagem de mensagens em uma *mail box* é assíncrona, mas o recebimento é síncrono.

A camada superior do DECK, é composta pelos módulos:

a) *Naming* : servidor de nomes;

b) *Message* : serviço de comunicação entre os vários *clusters*, tendo como foco a conversão de dados para o trabalho com *MultiClusters*;

c) *Group Communication* : rotinas de comunicação e sincronização de grupo;

d) *Tolerância a Falhas*

O DECK ainda está em processo de desenvolvimento e destas duas camadas, neste trabalho, basicamente foram utilizados todos os módulos do μDECK e, da camada superior, somente o módulo *Naming*. Como o módulo *Group Communication* possuía somente o *broadcast* implementado e quando este foi concluído, as rotinas próprias do algoritmo da aplicação já estavam implementadas, esse foi somente testado mas não utilizado na aplicação. Dessa forma, todas as rotinas como *gather, scatter*, e outras necessárias à comunicação de grupo foram implementadas nos algoritmos, como será descrito no capítulo 5.

As aplicações DECK seguem o modelo de execução SPMD (*Single Program Multiple Data*), ou seja, o mesmo processo é disparado simultaneamente em todos os nodos. Como parâmetros de inicialização, o processo recebe o número do nodo onde foi disparado, o número total de nodos e os endereços IP desses nodos. O número de cada nodo é dado pela ordem em que aparece na lista de nodos: para *p* nodos, a numeração varia de zero a *p*-1 [BAR98].

Algumas funções disponíveis no DECK e suficientes para desenvolver aplicações são listadas na figura 4.4. Outras funções podem ser consultadas no manual do DECK, disponível em [GPP2000].

deck_init	-	inicia uma aplicação DECK
deck_done	-	finaliza uma aplicação DECK
deck_node	-	retorna o número do nodo
deck_numnodes	-	retorna o número total de nodos na aplicação
deck_barrier	-	sincroniza todos os processos
deck_thread_create	-	criação de *thread*
deck_sem_create	-	criação de semáforo
deck_msg_create	-	criação de mensagem
deck_msg_pack	-	empacota mensagem
deck_msg_unpack	-	desempacota mensagem
deck_mbox_create	-	criação de *mail box*
deck_mbox_post	-	envia mensagem para *mail box*
deck_mbox_retrv	-	recebe mensagem da *mail box*

FIGURA 4.4 - Funções Básicas do DECK

Para multiprogramação, o DECK fornece primitivas básicas para manipulação de *threads*, como *create*, *sleep* e *destroy*. As *threads* dentro de cada processo podem criar *mail boxes* (recurso oferecido pelo DECK para trocas de mensagens entre os nodos) para se comunicarem pelo envio e recebimento de mensagens e são sincronizadas através da utilização de semáforos. As *mail boxes* são criadas por um servidor de nomes que é responsável por cadastrar e endereçar cada *mail box* criado. Isso é feito automaticamente quando uma aplicação é disparada [BAR2000].

A programação paralela utilizando o DECK é completamente explícita. Além das primitivas básicas existentes, estão sendo desenvolvidas algumas primitivas de comunicação coletiva, como *broadcast*, *gather* e *scatter*. O programador tem papel importante no desenvolvimento de aplicações em DECK pois precisa determinar explicitamente todas as transações necessárias à programação dos algoritmos paralelos.

Na descrição da comunicação entre os nodos, pode ser observado como o DECK manipula a troca de mensagens. Para cada processo de cada nodo, o programador cria uma *mail box*, cadastrando-a no servidor de nomes. Assim que um processo necessita enviar mensagens para outro, buscará a *mail box* do nodo receptor no servidor de nomes. Depois disso, empacota a mensagem e faz a postagem da mesma na *mail box* destino. Concorrentemente a isso, o nodo que receberá a mensagem deverá realizar um *retrieve* (verificar) em sua *mail box* se há mensagens a serem recebidas.

Atualmente, tem-se o DECK para aplicações na *Fast-Ethernet* e o DECK-BIP para a execução na *Myrinet.* Surgiram alguns problemas com o DECK durante o transcorrer desse trabalho. Por exemplo, problemas com o servidor de nomes; sincronização dos processos nos diferentes nodos (*deck_barrier*); na comunicação de grupo, quando o *broadcast* era feito todos para todos; desempenho do DECK-BIP.

Com relação à utilização de *threads*, também houveram questões que exigiram maior atenção, as quais serão explicitadas no capítulo 6. Todos esses problemas que emergiram são absolutamente normais no desenvolvimento de um *kernel*, e nesse caso, principalmente por esta aplicação também estar validando o funcionamento do DECK e avaliando o seu desempenho.

Nos casos ocorridos, a maioria dos problemas foi solucionada pela equipe do DECK e novas versões do mesmo foram disponibilizadas. Como será visto no capítulo 6, na situação que o DECK se encontra (em desenvolvimento), consegue-se obter bons resultados com relação ao desempenho. Dessa forma, é percebida a necessidade de serem continuadas aplicações em DECK, para que o mesmo possa ser ainda mais aprimorado e outros usuários comecem a utilizar esse ambiente.

Apesar de muitos estudos nessa área, a programação que inclui trocas de mensagens ainda está em um baixo nível porque a maioria das tarefas de paralelização são de responsabilidade do programador. Nesses casos, o programador define a comunicação e sincronização entre os processos, particionamento e distribuição dos dados, mapeamento dos processos entre os nodos e estruturas de dados utilizadas.

Com relação à parte lógica do ambiente paralelo de desenvolvimento dos algoritmos, cabe lembrar que as máquinas do *cluster* possuem o sistema operacional *Linux* 2.2.1 e também são disponibilizadas bibliotecas de comunicação paralela como PVM - *Parallel Virtual Machine* do *OAK Ridge National Laboratory* [PVM2000] e MPI - *Message Passing Interface* definida pelo MPI *Forum* [MPI2000]. Para a programação paralela, juntamente com o DECK, foi utilizado o compilador gcc versão 2.91.60 (egcs-1.1.1).

4.3 Considerações Finais

Para se chegar à computação transparente em *clusters* são necessários algoritmos que possam se adaptar à configuração apropriada de cada *cluster*. Estudos recentes mostram que há benefícios a serem obtidos se for repensada a paralelização combinando modelos de memória compartilhada com memória distribuída [BAK2000]. Essa combinação pode ser feita em aplicações executadas no *cluster* do Instituto de Informática e no capítulo 6, resultados relativos a ela são apresentados.

Clusters estão sendo cada vez mais utilizados e abre-se espaço a pesquisas relacionadas a isso, tanto a nível de aplicações quanto ferramentas que favoreçam o desenvolvimento das mesmas. Essa pesquisa é uma aplicação e por isso a breve descrição, nesse capítulo, do ambiente paralelo com relação à arquitetura e à programação paralela.

Outras informações sobre o *Cluster* podem ser encontradas em [RIG99] e sobre o DECK em [BAR98, BAR2000]. A página do GPPD [GPP2000], também é fonte de pesquisa a esses dois assuntos.

5 Paralelização dos Métodos de Resolução de Sistemas Lineares

A paralelização de algoritmos envolve basicamente duas etapas. A primeira delas está relacionada à paralelização propriamente dita, que pode ser desenvolvida para determinada aplicação ou feita sobre o algoritmo seqüencial; a segunda, é a implementação desse algoritmo na máquina paralela, utilizando um ambiente para programação. Em ambas etapas é necessário considerar a arquitetura da máquina paralela, pois essa influirá fortemente no algoritmo. A arquitetura interfere também no ambiente de programação, que pode ser uma linguagem paralela, a extensão de uma linguagem, uma biblioteca de trocas de mensagens ou uma linguagem seqüencial com paralelização automática de código.

Nesse estudo, os algoritmos paralelos são obtidos a partir do respectivo algoritmo seqüencial. Como descrito no capítulo anterior, o cluster de PCs é um sistema com memória distribuída, porém possui alguns nodos multiprocessados que compartilham memória. Na paralelização isso refletirá nos algoritmos implementados.

Nas próximas seções, é descrita a paralelização dos métodos de resolução de sistemas lineares esparsos: Algoritmo de Thomas e Gradiente Conjugado. Para mapear um algoritmo em um sistema paralelo, três pontos básicos devem ser considerados: identificação do paralelismo do algoritmo; particionamento do algoritmo ou do domínio do problema e distribuição das tarefas entre os processadores [MOL93]. Assim, antes de tratar dos algoritmos paralelos propriamente ditos, particionamento dos dados, formas de comunicação e operações específicas ao algoritmo são apresentados.

No método do Gradiente Conjugado, o uso de *threads* é enfocado. Além disso, versões pré-condicionadas do método, com os pré-condicionadores Diagonal e Polinomial, baseado em Séries Truncadas de Neumann, são apresentadas. O pré-condicionador Diagonal é equivalente ao pré-condicionador polinomial com grau do polinômio igual a *zero*. Para o pré-condicionador polinomial são experimentados polinômios com grau 1, 2, 3 e 4.

5.1 Paralelização do Algoritmo de Thomas

Como explicitado por Kumar [KUM94], os métodos diretos de resolução de sistemas lineares são o desafio à paralelização. O pequeno número de operações por iteração faz com que, no algoritmo paralelo, a comunicação prevaleça sobre o processamento.

Em alguns casos, além do aspecto considerado anteriormente, devido a existência de dependência de dados, o algoritmo não ocupa todo o tempo disponível dos nodos para o processamento. Conforme a abordagem dada ao método do Algoritmo de Thomas (AT), pode-se defrontar com essa situação.

Duas abordagens principais podem ser identificadas. A primeira delas é a Decomposição de Domínio, onde se decompõe o domínio físico do problema e faz-se a distribuição dos sub-domínios entre os nodos. Para cada sub-domínio, tem-se um sistema linear, consequentemente, menor que o sistema linear do domínio sem a decomposição física. Para cada sistema, é aplicado o Algoritmo de Thomas (AT) como um resolvedor local, ou seja, a versão seqüencial do método. Cada nodo executa o

método independentemente e a comunicação é feita após os vários sistemas lineares estarem solucionados.

A segunda abordagem considera a paralelização do método de resolução. Há um único sistema linear para todo o domínio físico. Este é particionado entre os nodos e o algoritmo paralelo do método é executado. Isso implica na existência de comunicação entre os nodos durante o transcorrer da execução.

Como o escopo dessa pesquisa enquadra-se na segunda abordagem, nessa seção é descrito o método do Algoritmo de Thomas distribuído. Antes disso, abordam-se o particionamento da matriz e o tipo de comunicação necessária.

5.1.1 Particionamento de Dados

O particionamento dos dados envolve o particionamento da matriz do sistema linear, que é esparsa do tipo banda, do vetor dos termos independentes e do vetor das incógnitas. O particionamento adotado é por contigüidade e por linhas, pois como os nodos do *cluster* comunicam-se diretamente uns com os outros, essa é a forma mais simples de particionar os dados.

Seja *N* a ordem da matriz e do vetor e *Nnodes* o número de nodos na aplicação, o particionamento é feito atribuindo-se *k* linhas da matriz e elementos do vetor para cada nodo, onde $k = N / Nnodes$. Por exemplo, sendo $N = 1000$ e $Nnodes = 4$, $k = 250$, ou seja, cada nodo receberá 250 linhas da matriz e 250 elementos dos vetores.

Cada nodo é identificado por um número *p* que varia de *zero* a *Nnodes* – 1. Como o particionamento é contínuo, o nodo *p* receberá as linhas da matriz e os elementos do vetor que estiverem no intervalo $[p * k; ((p+1) * k) - 1]$. Na figura 5.1, esse particionamento é ilustrado com a matriz no formato otimizado diagonal, pois somente as diagonais não nulas são armazenadas.

A carga dos dados entre os processadores estará balanceada com essa forma de particionamento. Para os casos em que *N* não é múltiplo de *Nnodes*, o procedimento adotado foi designar um elemento a mais dos vetores e uma linha a mais da matriz aos *r* primeiros nodos, onde *r* é o resto da divisão $N / Nnodes$. Por exemplo, tendo-se $N = 1010$ e $Nnodes = 4$, os dois primeiros nodos receberão 253 linhas da matriz e elementos dos vetores e os dois restantes, 252.

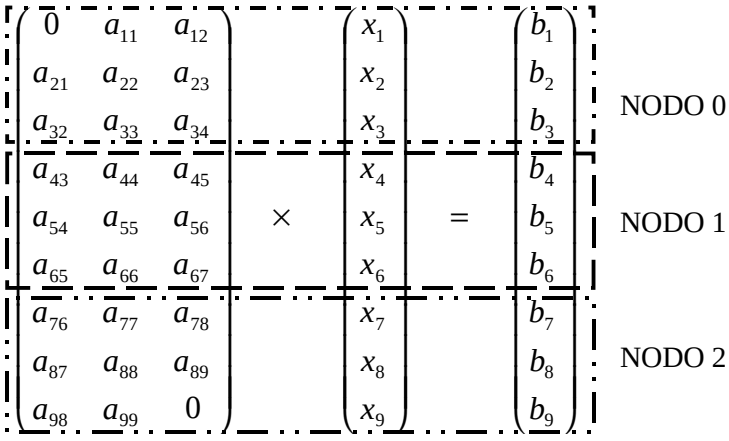


FIGURA 5.1 – Particionamento dos Dados

5.1.2 Comunicação

As operações de comunicação necessárias ao Algoritmo de Thomas envolvem somente comunicação local. Esse tipo de comunicação caracteriza-se por envolver alguns nodos, e não todos os da aplicação, sendo geralmente o nodo remetente e seus vizinhos [FOS95].

Basicamente, a comunicação envolvida nesse algoritmo é a transferência simples de mensagem entre dois nodos: o remetente e o receptor que, dependendo do procedimento a ser executado, será o nodo anterior ou posterior ao remetente. Em cada operação de comunicação, o conteúdo da mensagem é no máximo dois escalares.

```
#include <deck.h>
int main(int argc, char **argv)
{
        char mailbox[5];
        double vlr;
        deck_mbox_t mb;
        deck_msg_t m;

        deck_init(&argc. &argv);
        sprintf(mailbox, “MAIL%d”, deck_node());
        deck_mbox_create(&mb);
        deck_mbox_bind(&mb, mailbox);
        deck_msg_create(&m, sizeof(vlr), NULL);
        if (deck_node() == 0)
                {
                        vlr = 5000;
                        deck_msg_pack(&m, DECK_DOUBLE, &vlr, 1);
                        while(deck_mbox_fetch(&mb, “MAIL1”) != DECK_EOK)
                                deck_thread_sleep(1);
                        deck_mbox_post(&mb, &m);
                }
        else if (deck_node() ==1)
                {
                        deck_mbox_retrv(&mb, &m);
                        deck_msg_reset(&m);
                        deck_msg_unpack(&m, DECK_DOUBLE, &vlr, 1);
                }
        deck_msg_destroy(&m);
        deck_mbox_destroy(&mb);
        deck_done();
        return(0);
}
```

FIGURA 5.2 – Exemplo de Troca de Mensagem em DECK

Para acontecer a troca de mensagem entre dois nodos, utilizando o DECK, inicialmente, cada nodo deverá ter criado uma *mail box* (*deck_mbox_create*) e tê-la cadastrado no servidor de nomes (*deck_mbox_bind*). Ambos nodos criam uma mensagem com o tamanho desejado (*deck_msg_create*). Para esse caso de envio e recebimento de um único escalar, o nodo remetente empacota o valor a ser enviado na mensagem criada (*deck_msg_pack*) e procura, no servidor de nomes, a *mail box* do nodo que receberá a mensagem (*deck_mbox_fetch*). Assim que a *mail box* for encontrada, o nodo envia a mensagem (*deck_mbox_post*).

Paralelamente, o nodo receptor executa uma operação de verificação de *mail box* (*deck_mbox_retrv*), recuperando a mensagem postada. Para que haja o desempacotamento o nodo executa um *reset* (*deck_msg_reset)* na mensagem e somente após, a desempacota (*deck_msg_unpack*). Assim, conclui-se o processo de troca de mensagem entre dois nodos. Na figura 5.2, há um trecho de código em DECK, onde o nodo 0 envia um escalar ao nodo 1.

5.1.3 Algoritmo de Thomas

O método do Algoritmo de Thomas foi descrito na seção 3.3.1 e seu algoritmo seqüencial está no Anexo 1. Esse método possui duas etapas: o *Forward* e o *Backward* (ou retrossubstituição).

Antes de ser iniciado o processamento do mesmo no *cluster*, a matriz do sistema linear, os vetores das incógnitas e dos termos independentes são particionados entre os nodos, conforme o particionamento exposto na seção 5.1.1. Na primeira etapa do método, apenas duas linhas de comando são executadas, como pode ser visto no algoritmo no anexo 1, e na segunda etapa, somente uma. Nas duas etapas há dependência de dados, além do reduzido número de operações por iteração. Consequentemente, em cada instante de tempo somente um processador pode estar executando processamento útil, ou seja, executando o método de resolução. Devido a essa característica, o Algoritmo de Thomas é dito distribuído.

O nodo *zero* começa a execução do *Forward* e assim que concluir, envia os dois escalares necessários ao próximo nodo para que esse também inicie o processamento. Cada nodo repete esse ciclo. Ao terminarem o processamento da primeira etapa, os nodos permanecem aguardando dados para que seja dado início ao *Backward*.

O último nodo, após terminar a primeira etapa, imediatamente inicia o *backward* e, quando completo, envia um escalar ao nodo anterior. Esse último, recebendo o escalar, continua a etapa do *backward* e repete o envio ao nodo anterior. Esse procedimento é feito até que o primeiro nodo execute a etapa de retrossubstituição, estando o sistema linear resolvido.

O método do Algoritmo de Thomas é apresentado a seguir. Essa descrição é enumerada, para uma melhor compreensão, e considera-se *Nnodes* como sendo o número de nodos participantes na aplicação e *p* o número que identifica o nodo. Na figura 5.3 há o diagrama de execução desse algoritmo.

Passo 1. Definido o particionamento e distribuídos a matriz e vetores do sistema linear, inicia-se a solução do sistema;

Passo 2. Execução da primeira etapa do método – *Forward*

Passo 2.1. Nodo *zero*:

$c'_0 = c_0 / b_0$
$d'_0 = d_0 / b_0$
$c'_i = c_i / (b_i - a_i * c'_{i-1})$
$d'_i = (d_i - a_i * d'_{i-1}) / (b_i - a_i * c'_{i-1})$, onde $1 \leq i \leq (p * N / Nnodes) - 1$
Enviar os escalares *c* e *d* relativos a última linha que o nodo contém para o próximo nodo;

Passo 2.2. Próximo nodo:

Receber os escalares *c* e *d* do nodo anterior;

$c'_i = c_i / (b_i - a_i * c'_{i-1})$

$d'_i = (d_i - a_i * d'_{i-1}) / (b_i - a_i * c'_{i-1})$ onde $0 \leq i \leq (p * N / Nnodes) - 1$

Passo 2.2.1 Se (não for o último nodo) então

Enviar os dois escalares para o próximo nodo
Executar o passo 2.2

Passo 2.2.2 senão Executar passo 3

Passo 3. Execução da segunda etapa do método – *Backward*

Passo 3.1. Último nodo:

$x_{(p * N / Nnodes) -1} = d'_{(p * N / Nnodes) -1}$

$x_i = d'_i - x_{i+1} * c'_i$, onde $(p * N / Nnodes) - 2 \geq i \geq 0$

Enviar o valor de *x* correspondente à primeira linha que possui para o nodo anterior;

Passo 3.2. Nodo anterior:

Receber o escalar *x* do nodo posterior;

$x_i = d'_i - x_{i+1} * c'_i$, onde $(p * N / Nnodes) - 1 \geq i \geq 0$

Passo 3.2.1. Se (não for o primeiro nodo) então

Enviar o valor do escalar *x*, correspondente a primeira linha que possui, ao nodo anterior
Executar passo 3.2

Passo 3.2.2 senão Executar passo 4

Passo 4. Sistema linear resolvido.

Nessa abordagem do método do Algoritmo de Thomas, devido ao pequeno número de operações e à existência de dependência de dados presente em todas as etapas, a execução distribuída do algoritmo é prejudicada. Para resolução de um único sistema linear, com esse método, os nodos permanecem ociosos, uma parte significativa do tempo de processamento!

Se houverem vários sistemas lineares que não dependam entre si, essa versão do Algoritmo de Thomas pode ser expandida para o Thomas Pipeline. A idéia é a cada tempo de execução ser iniciada a resolução de um novo sistema linear, aproveitando os nodos disponíveis e, consequentemente, diminuindo o tempo ocioso. No tempo 2 de execução (figura 5.3), por exemplo, enquanto o primeiro nodo está ocioso, esse pode iniciar a resolução de outro sistema linear que não dependa do que está sendo resolvido no momento. Esse processo é repetido sucessivamente, até o pipeline estar "cheio".

Povitsky [POV98a, POV98b] realizou um estudo aprofundado dessa abordagem e constatou que a mesma pode retornar resultados satisfatórios. Ele investigou estratégias de aumentar a eficiência do Algoritmo de Thomas, uma vez que isso não pode ser alcançado diretamente. Especialmente, centrou-se na ordem em que a resolução dos

sistemas é iniciada, o que inclui as etapas *Forward* e *Backward*, chegando a um número ótimo como função dos tempos de computação e comunicação.

Outro trabalho, encontrado na revisão bibliográfica, que utiliza o Algoritmo de Thomas na resolução paralela de sistemas lineares foi o de Vollebregt [VOL97]. Nesse é utilizada a decomposição de domínio, com grande sobreposição de sub-domínios, gerando-se vários SELAs tridiagonais que são distribuídos entre os processadores. O Algoritmo de Thomas é aplicado localmente em cada processador. Juntamente com a discretização, é utilizado um esquema de iterações pares e ímpares.

Tempo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Nodo 0	F	-	-	-	-	-	-	B
Nodo 1	-	F	-	-	-	-	B	-
Nodo 2	-	-	F	-	-	B	-	-
Nodo 3	-	-	-	F	B	-	-	-

F	Etapa *Forward* do Algoritmo de Thomas
B	Etapa *Backward* do Algoritmo de Thomas
-	Tempo Ocioso

FIGURA 5.3 – Diagrama Execução do Algoritmo de Thomas

5.2 Paralelização do Método do Gradiente Conjugado

O algoritmo seqüencial do Gradiente Conjugado é formado por operações básicas de álgebra linear, como pode ser verificado em sua descrição no capítulo 3, seção 3.4.1. Por essa formulação, a paralelização do Gradiente é feita a partir da paralelização dessas operações básicas. Como o sistema linear que se está trabalhando é esparso, as operações de álgebra linear adquirem novas peculiaridades que incorporam a esparsidade e o paralelismo.

Para que se obtenha o algoritmo paralelo do Gradiente Conjugado, basicamente três pontos devem ser considerados: particionamento dos dados, operações básicas paralelas de álgebra linear e rotinas de comunicação. O particionamento diz respeito à distribuição dos elementos do sistema linear (matriz e vetores). As operações básicas paralelas envolvem soma/subtração de vetores, multiplicação de um escalar por vetor, produto escalar de dois vetores e multiplicação matriz esparsa por vetor. A comunicação consiste nas trocas de mensagens entre os nodos. Esses três aspectos são descritos nas próximas seções.

5.2.1 Particionamento dos Dados

Além da matriz, do vetor dos termos independentes e do vetor de incógnitas do sistema linear, no método do Gradiente Conjugado são necessários vetores, com a mesma ordem dos vetores do sistema, para armazenar o resíduo, a direção de pesquisa e outros dados necessários ao método. O particionamento adotado para distribuição desses elementos entre os nodos é o mesmo usado no Algoritmo de Thomas, descrito na seção 5.1.1, ou seja, particionamento por contigüidade e por linhas.

Cada nodo armazena uma parte dos vetores e um bloco de linhas da matriz do sistema linear. O formato diagonal é utilizado para o armazenamento da matriz nos nodos.

5.2.2 Rotinas de Comunicação

Nesta seção são descritas rotinas de comunicação como *broadcast*, *gather*, *scatter*, redução e comunicação par-ímpar. A operação *gather* é utilizada quando está se trabalhando com matrizes densas. Todas as demais operações são necessárias ao algoritmo paralelo do Gradiente Conjugado que manipula sistemas lineares esparsos.

5.2.2.1 *Broadcast*

Uma forma de comunicação que envolve todos os processos da aplicação é uma comunicação coletiva. Um *broadcast* é uma comunicação coletiva onde um simples processo envia os mesmos dados (escalar, vetor simples e/ou estruturado) para todos os outros processos. Ao final dessa comunicação existirá uma cópia dos dados em cada nodo. Na figura 5.4 (a), há um diagrama do *broadcast*. No caso de todos os processos necessitarem enviar seus dados para todos os outros processos, tem-se o chamado *broadcast* todos para todos ou transmissão todos para todos, ilustrado na figura 5.4. (b). Nessa figura, *a*, *b*, *c*, ... e *p* são escalares.

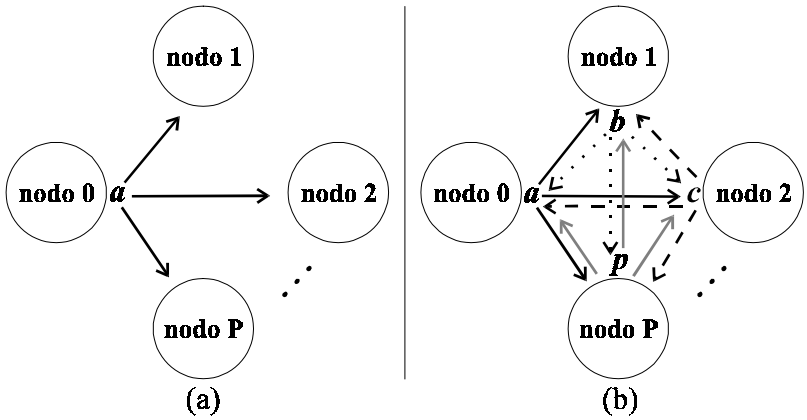


FIGURA 5.4 – *Broadcast*

Essas rotinas foram implementadas no DECK, a partir das primitivas de comunicação entre nodos, conforme visto na figura 5.4, pois até então o DECK não disponibilizava rotinas de comunicação de grupo. Algum tempo depois, a rotina do *broadcast* do DECK foi criada. O *broadcast* no DECK e o implementado nesse trabalho são feitos a partir de uma seqüência de mensagens enviadas a partir de um nodo para os demais nodos participantes da aplicação.

5.2.2.2 *Gather*

Um *Gather* é a comunicação que consiste em um único processo coletar uma estrutura de dados distribuída entre os outros processos da aplicação [PAC97]. A figura 5.5 ilustra a operação de comunicação *gather*. Por exemplo, na multiplicação matriz densa por vetor, estando a matriz e o vetor particionados, é necessário um *gather* do vetor para cada nodo, a fim de ter o vetor completo em todos os nodos e a multiplicação paralela poder ser realizada.

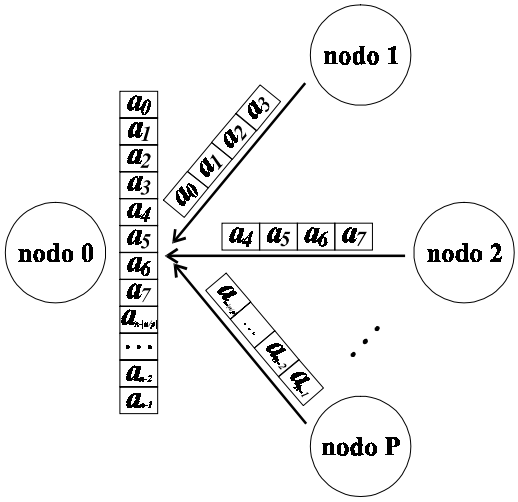


FIGURA 5.5 – *Gather*

5.2.2.3 *Scatter*

Na comunicação *scatter*, a estrutura de dados armazenada em um único nodo é distribuída para os outros nodos [PAC97], conforme o exemplo da figura 5.6. Esse tipo de comunicação é utilizado, na distribuição inicial dos dados entre os nodos que participarão da execução da aplicação.

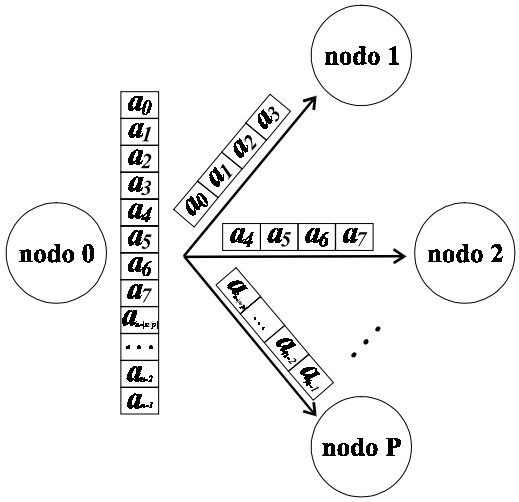


FIGURA 5.6 – *Scatter*

5.2.2.4 Redução

Em uma comunicação do tipo redução todos os processos da aplicação enviam dados a um determinado processo a fim de fazer determinada operação [PAC97]. Algumas dessas operações são acumulação dos valores, valor máximo/mínimo, produto, dentre outras.

No algoritmo paralelo do método do Gradiente Conjugado, a operação de redução é utilizada no produto escalar de dois vetores para acumular valores. A ilustração da operação de redução está na figura 5.7.

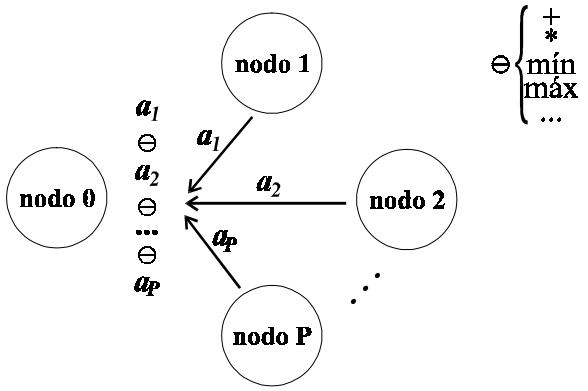


FIGURA 5.7 –Redução

5.2.2.5 Comunicação Par – Ímpar

Na multiplicação paralela matriz densa por vetor, com particionamento dos dados por linha, é preciso que o vetor esteja inteiro (com todos os seus elementos) em cada nodo, para que o produto escalar de cada linha da matriz com o vetor possa ser realizado. Como o vetor se encontra distribuído, através da comunicação *gather* é possível a reunião dos elementos do vetor que estão faltando em cada nodo.

Os sistemas lineares tratados nesse trabalho possuem matrizes tridiagonais, portanto não há necessidade de se ter o vetor, que será multiplicado pela matriz, inteiro nos nodos. Antes da multiplicação, somente os elementos do vetor que serão usados na multiplicação são transferidos de um nodo para o outro. A comunicação *par – ímpar* reorganiza o vetor nos nodos para tal operação.

Na figura 5.12, pode-se observar que o particionamento da matriz e dos vetores, para a multiplicação, é por linhas, porém, o vetor que multiplica a matriz esparsa necessita alguns elementos adicionais no nodo em que está, mas não todos os elementos do vetor. Para o caso tridiagonal, os nodos primeiro e último necessitam um elemento a mais; e nos demais nodos são necessários dois elementos a mais, um no início e outro no final do vetor que cada nodo possuí. Já para o caso pentadiagonal, o primeiro e o último nodo necessitam 2 elementos a mais, e os demais 4. Para matrizes com 7 diagonais, o primeiro e o último nodo necessitam 3 elementos a mais e os demais nodos 6.

A partir disso, pode-se concluir que, em uma matriz com D diagonais, para a multiplicação esparsa, o primeiro e último nodos necessitam $(D - 1)/2$ elementos adicionais do vetor e os demais nodos $(D - 1)$ elementos a mais do vetor. Trabalhando-se com sistemas esparsos, é possível assim otimizar a comunicação envolvida nas operações de álgebra linear.

A comunicação *par-ímpar* é responsável por organizar os elementos do vetor antes da multiplicação pela matriz que será descrita em 5.2.3.4. Há necessidade dessa comunicação pela impossibilidade dos nodos enviarem e receberem mensagens ao mesmo tempo, uma vez que a comunicação não é bufferizada.

A comunicação *par-ímpar* é feita em dois passos: no primeiro, os nodos pares enviam os elementos necessários do vetor para os nodos ímpares, conforme a figura 5.8(a); e no segundo passo, ilustrado na figura 5.8(b), os nodos ímpares enviam e os pares recebem os elementos. Os nodos recebem os escalares e os colocam na posição correspondente no vetor que participará da multiplicação (vetor *b*).

Para uma matriz tridiagonal, no primeiro passo, os nodos pares enviam o último elemento do vetor para o próximo nodo ímpar e o primeiro elemento do vetor para o nodo ímpar anterior. No segundo passo, os ímpares enviam o primeiro elemento do vetor para o nodo par anterior e o último elemento do vetor para o próximo nodo par. Para matrizes esparsas com um número maior de diagonais, nessa comunicação são enviados (D – 1)/2 elementos de cada vez, onde D é o número de diagonais da matriz

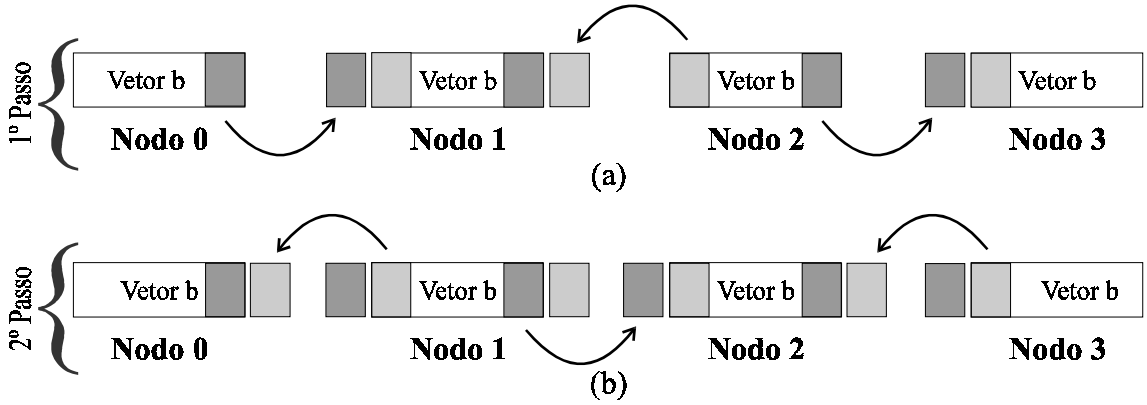


FIGURA 5.8 – Comunicação Par–Ímpar

5.2.3 Operações de Álgebra Linear

O algoritmo do Gradiente Conjugado é composto basicamente por operações de álgebra linear que envolvem matrizes e vetores. Dentre essas, estão operações de soma/subtração de vetores, produto escalar (ou produto interno) de dois vetores, multiplicação de um escalar por vetor e multiplicação matriz esparsa por vetor.

Uma parte significativa das operações entre vetores são executadas sem que haja comunicação entre os nodos, proporcionando um aumento na velocidade de execução, pelo paralelismo existente. A seguir, são caracterizadas cada uma das operações paralelas de álgebra linear do Gradiente Conjugado, considerando-se que os dados já estão particionados e, assim, cada nodo possui uma parte dos vetores e blocos de linhas da matriz do sistema linear.

5.2.3.1 Soma/Subtração Paralela de Vetores

Para ser realizada a soma ou subtração de dois vetores, os nodos executam o algoritmo seqüencial de soma/subtração sobre a porção de dados dos vetores que lhes cabe, não sendo preciso trocar informações entre nodos. A soma de dois vetores *b* e *c*,

acumulada no vetor *d* é dada por $d_i = b_i + c_i$, onde $1 \leq i \leq ne$ e *ne* é o número de elementos dos vetores em cada nodo. A figura 5.9, ilustra essa operação paralela.

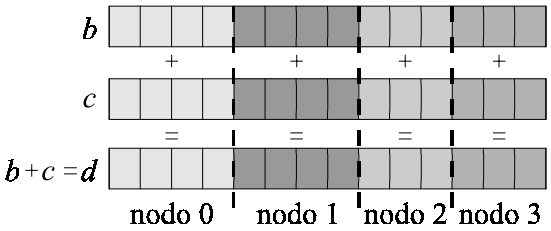


FIGURA 5.9 – Soma Paralela de Vetores

5.2.3.2 Multiplicação Paralela de Escalar por Vetor

Na multiplicação de um escalar por vetor, da mesma forma como na soma de vetores, cada nodo executa a multiplicação do escalar pela parte do vetor que possui. Seja *b* um vetor e *k* um escalar, o vetor *c* é o vetor resultante da multiplicação, obtido por $c_i = k * b_i$, onde $1 \leq i \leq ne$ e *ne* é o número de elementos dos vetores em cada nodo.

Na figura 5.10, tem-se a representação da multiplicação paralela de um escalar por vetor. Essa operação é a aplicação do algoritmo seqüencial de multiplicação escalar por vetor, simultaneamente, em vários blocos do vetor, não havendo comunicação entre nodos.

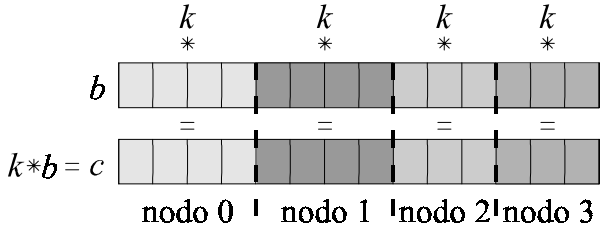


FIGURA 5.10 – Multiplicação Paralela de Escalar por Vetor

5.2.3.3 Produto Escalar Paralelo

No produto escalar de dois vetores há necessidade de comunicação. Cada nodo calcula a sua parte do produto escalar com os dados que possui, resultando em um escalar, chamado produto escalar parcial (*pep*).

Esse cálculo é dado por $pep_j = \sum_{i=1}^{ne} b_i * c_i$, onde *b* e *c* são os vetores, $1 \leq i \leq ne$, *ne* é o número de elementos dos vetores em cada nodo, $0 \leq j \leq Nnodes - 1$ e *Nnodes* é o número de nodos. O produto escalar (*pe*) dos vetores é o somatório de cada produto escalar parcial (*pep*): $pe = \sum_{j=0}^{Nnodes-1} pep_j$

Depois disso, o produto escalar final precisa ser calculado, acumulando-se os produtos parciais de cada nodo em um único nodo. Para essa acumulação uma soma dos produtos parciais é feita, ficando produto escalar final em um nodo. Essa operação de acumulação é a operação de redução. Depois disso, um *broadcast* do produto escalar é

feito para que esse seja enviado a todos os outros nodos. A figura 5.11 ilustra essa operação.

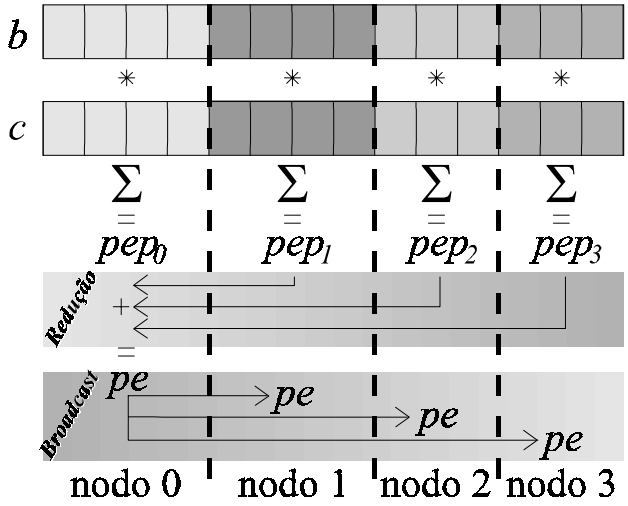


FIGURA 5.11 – Produto Escalar Paralelo

5.2.3.4 Multiplicação Paralela Matriz Esparsa por Vetor

A operação de maior custo computacional e que, por isso, determinará a complexidade do algoritmo do Gradiente Conjugado é a multiplicação de uma matriz esparsa por vetor. Nesse caso, para a elaboração do algoritmo com o mínimo de comunicação possível, foi observada a localização de cada elemento do vetor resultante dessa multiplicação. Na figura 5.12, tem-se a representação do particionamento da multiplicação $Ab = c$, sendo A uma matriz tridiagonal, b um vetor e c o vetor resultante da multiplicação.

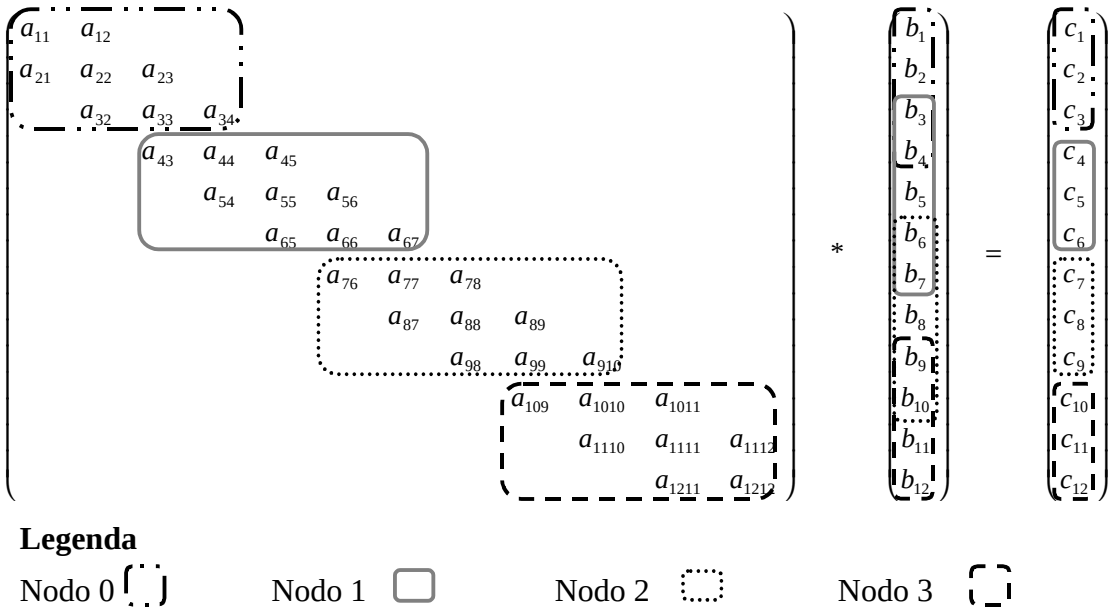


FIGURA 5.12 - Multiplicação Matriz Esparsa – Vetor

O particionamento do vetor b precisa sofrer uma reorganização para eliminar a necessidade de comunicação durante o cálculo de elementos do vetor c. Essa re-distribuição do vetor é feita pela comunicação *par-ímpar*, descrita em 5.2.2.4.

A matriz tridiagonal é armazenada no formato diagonal. Com isso, cada nodo terá uma parte da matriz do sistema, tratando-a como uma sub-matriz. Isso está representado na figura 5.13 e está relacionado com a figura 5.12, onde a matriz não é armazenada no formato otimizado.

$$M = \begin{pmatrix} 0 & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \\ m_{51} & m_{52} & m_{53} \\ m_{61} & m_{62} & m_{63} \\ m_{71} & m_{72} & m_{73} \\ m_{81} & m_{82} & m_{83} \\ m_{91} & m_{92} & m_{93} \\ m_{101} & m_{102} & m_{103} \\ m_{111} & m_{112} & m_{113} \\ m_{121} & m_{122} & 0 \end{pmatrix} \qquad Dif = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

Legenda

Nodo 0 Nodo 1 Nodo 2 Nodo 3

FIGURA 5.13 - Matriz Tridiagonal Armazenada no Formato Diagonal

No formato diagonal, conforme foi descrito no capítulo 3, seção 3.2.1.1, cada elemento da matriz *M*, que armazena a matriz *A* de forma otimizada, corresponde seus elementos com os elementos de *A*, da seguinte forma: $m_{i,j} = a_{i,i+Dif_j}$. Assim, m_{52} corresponde ao elemento $a_{5,5}$. Os elementos m_{11} e m_{1212} possuem o valor *zero* porque não correspondem a nenhum elemento da matriz *A*.

Para a realização da multiplicação, em primeiro lugar, deve-se considerar que a matriz é tridiagonal (figura 5.12). Em segundo lugar, precisa-se lembrar que essa matriz é armazenada no formato diagonal (figura 5.13) para otimizar o armazenamento. Por fim, observar que o particionamento da matriz, para distribuição entre os nodos, é feito sobre o formato diagonal, e consequentemente cada nodo tratará cada parte da matriz como uma sub-matriz, ou seja, os índices de linha e coluna dos elementos da matriz original não são mantidos (figura 5.14), tendo cada sub-matriz os seus próprios índices.

Realizada a comunicação *par-ímpar*, cada nodo pode calcular os elementos do vetor *c* que contém, como em uma rotina de multiplicação seqüencial. Para uma multiplicação densa seqüencial, o cálculo de cada elemento de *c* pode ser obtido por:

$$c_i = \sum_{j=1}^{n} a_{ij} * b_j \qquad (5.1)$$

onde $1 \leq i \leq n$ e *n* é a ordem do sistema.

Como a matriz está armazenada, em cada nodo, no formato diagonal e a execução é paralela, essa multiplicação adquire algumas particularidades. Considera-se que *nd* é o

número de diagonais do sistema linear, *nl* é o número de linhas que cada nodo contém da matriz e $1 \leq i \leq nl$. Para o nodo *zero*, os elementos do vetor *c* são obtidos por (5.2).

$$c_i = \sum_{j=1}^{nd} a_{ij} * b_{j+i-2} \tag{5.2}$$

Para os demais nodos (nodos diferentes do nodo *zero*), os elementos do vetor *c* são obtidos por (5.3).

$$c_i = \sum_{j=1}^{nd} a_{ij} * b_{j+i-1} \tag{5.3}$$

Há duas exceções, nessa multiplicação, nas quais não são aplicadas as equações anteriores. A primeira delas é no nodo *zero*, para o elemento a_{11} e a segunda, no último nodo, para o elemento $a_{nd\,nd}$. Em ambos os casos, devido ao formato de armazenamento diagonal, esses elementos não correspondem a nenhum elemento da matriz do sistema linear.

A tabela 5.1 mostra como cada matriz e vetor é armazenado e qual a equação que define a multiplicação nos respectivos nodos, conforme a descrição anterior. Cada nodo armazena a matriz *A* na matriz *M* que está otimizada, como pode ser visto na figura 5.13. O vetor *b* é o vetor que multiplica a matriz e o vetor *c* é o vetor resultante.

TABELA 5.1 – Multiplicação Paralela de Matriz Tridiagonal por Vetor

Nodo	Matriz *M*	Vetor *b*	Vetor *c*	Equação	Exceção
0	$\begin{pmatrix} 0 & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$	$\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$	$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$	$c_i = \sum_{j=1}^{nd} a_{ij} * b_{j+i-2}$	Para o cálculo de c_1, não é incluido o elemento m_{11}, pois esse não tem correspondente na matriz *A*.
1	$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$	$\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}$	$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$	$c_i = \sum_{j=1}^{nd} a_{ij} * b_{j+i-1}$	Não há.
2	$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$	$\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}$	$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$	$c_i = \sum_{j=1}^{nd} a_{ij} * b_{j+i-1}$	Não há.
3	$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & 0 \end{pmatrix}$	$\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$	$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$	$c_i = \sum_{j=1}^{nd} a_{ij} * b_{j+i-1}$	Para o cálculo de c_3, não é incluido o elemento m_{33}, pois esse não tem correspondente na matriz *A*.

Essa multiplicação paralela de matriz tridiagonal por vetor pode ser estendida para as matrizes esparsas com um número maior de diagonais adjacentes. As equações da tabela 5.1 são substituídas por equações genéricas.

Para os nodos diferentes de *zero*, as equações permanecem as mesmas, porém há outras exceções. Uma vez que a matriz é esparsa e armazenada no formato diagonal, ao invés de se ter somente um elemento que não corresponda com a matriz densa do sistema, como no caso tridiagonal, no último nodo, existirá um número maior de elementos. Dessa forma, a equação definida para multiplicação nos nodos diferentes de zero deve ser aplicada para os casos onde $(j+i-1) <$ *número de elementos do vetor b no nodo* .

Para o nodo *zero*, assim como na multiplicação tridiagonal, a equação somente deverá ser aplicada aos elementos de *M* que possuem correspondentes na matriz *A*. A equação utilizada deve ser a equação (5.4).

$$c_i = \sum_{j=0}^{nd} a_{ij} * b_{j+i-k-1} \qquad (5.4)$$

onde $k = (nd - 1)/2$ e *nd* é o número de diagonais da matriz do sistema linear.

5.2.4 Algoritmo Paralelo do Gradiente Conjugado

O algoritmo paralelo do Gradiente Conjugado é descrito baseado no algoritmo seqüencial da seção 3.4. Essa versão do Gradiente Conjugado paralelo destina-se a processamento paralelo que utiliza-se de memória distribuída.

Passo 1. Definida a forma de particionamento conforme a ordem do sistema (*n*) e o número de nodos na aplicação, a matriz dos coeficientes (*A*) e o vetor dos termos independentes (*b*) do sistema linear são distribuídos. Os demais vetores necessários ao algoritmo paralelo do Gradiente Conjugado são particionados da mesma forma;

Passo 2. Cada nodo faz a aproximação inicial do vetor de incógnitas *x*, que encontra-se particionado, ou seja, inicializa a parte do vetor *x* que possui;

Passo 3. Inicializa contador de iterações $i = 1$

Passo 4. $r = b - Ax$

Passo 4.1. Multiplicação matriz esparsa por vetor Ax
(em paralelo, conforme seção 5.2.3.4)

Passo 4.2. Subtração de vetores $b - Ax$
(em paralelo, como na seção 5.2.3.1)

Passo 5. $d = r$ (em paralelo)

Passo 6. $\delta_{novo} = r^T r$
Produto escalar paralelo
(como na seção 5.2.3.3)

Passo 7. $\delta_0 = \delta_{novo}$

Início das Iterações : Enquanto $i < i_{\max}$ e $\delta_{novo} > \varepsilon^2 \delta_0$ faça

Passo 8. $q = Ad$

Passo 8.1. Comunicação *par-ímpar* para atualizar o vetor *d*
(conforme seção 5.2.2.4)

Passo 8.2. Multiplicação matriz esparsa por vetor *Ad*
(em paralelo, conforme seção 5.2.3.4)

Passo 9. $\alpha = \dfrac{\delta_{novo}}{d^T q}$

Passo 9.1. Produto escalar paralelo $d^T q$
(como na seção 5.2.3.3)

Passo 9.2. Cálculo de α

Passo 10. $x = x + \alpha q$

Passo 10.1. Multiplicação escalar vetor (αq)
(em paralelo, conforme seção 5.2.3.2)

Passo 10.2. Soma de vetores $x = x + \alpha q$
(em paralelo, conforme seção 5.2.3.4)

Se i é divisível por 50

Passo 11. $r = b - Ax$

Passo 11.1. Comunicação *par-ímpar* para atualizar o vetor *x*
(conforme seção 5.2.2.4)

Passo 11.2. Subtração de vetores $b - Ax$
(em paralelo, como na seção 5.2.3.1)

senão

Passo 12. $r = r - \alpha q$

Passo 12.1. Multiplicação escalar vetor (αq)
(em paralelo, conforme seção 5.2.3.2)

Passo 12.2. Subtração de vetores $r = r - \alpha q$
(em paralelo, conforme seção 5.2.3.4)

Passo 13. $\delta_{velho} = \delta_{novo}$

Passo 14. $\delta_{novo} = r^T r$

Produto escalar paralelo
(como na seção 5.2.3.3)

Passo 15. $\beta = \dfrac{\delta_{novo}}{\delta_{velho}}$

Passo 16. $d = r + \beta d$

Passo 16.1. Multiplicação escalar vetor (βd)
(em paralelo, conforme seção 5.2.3.2)

Passo 16.2. Soma de vetores $d = r + \beta d$
(em paralelo, conforme seção 5.2.3.4)

Passo 17. Incrementa contador de iterações $i = i + 1$

Fim da Iteração

Passo 18. Sistema linear resolvido.

5.2.5 Utilização de *Threads*

Threads são unidades básicas de execução que utilizam a CPU. Uma *thread* é uma seqüência de instruções que são executadas dentro de um processo. Todas as *threads* dentro de um processo compartilham um mesmo espaço de endereço e tem igual acesso a todos os recursos do processo [PRA97]. Por esse motivo, *threads* tem tempo de criação inferior ao tempo de criação dos processos.

Cada *thread* é um fluxo diferente de controle que pode executar suas instruções independentemente, permitindo a existência de um processo com mais de uma *thread*, que é também é conhecido como processo *multithreaded* [PRA97].

Dentre as várias razões para serem utilizadas *threads*, tem-se: obtenção de ganho de desempenho em máquinas multiprocessadas com memória compartilhada, onde a aplicação é dividida em tarefas menores executadas por *threads;* diminuição da latência da rede, proporcionando, por exemplo, a sobreposição de operações de entrada e saída, com processamento na CPU; necessidade de criação de vários processos, sendo esses substituídos por *threads*, dentro outras.

Para algoritmos assíncronos, a utilização de *threads* permite um paralelismo de grão médio [PRA97], pois cada componente da aplicação é manipulado por uma *thread* diferente. Como as *threads* são independentes, ocorrerá um aumento no desempenho do algoritmo.

Em alguns casos em que o problema a ser resolvido pode ser dividido em problemas menores, a utilização de *threads* torna o algoritmo simplificado, além de proporcionar um modelo de concorrência mais barato. Por outro lado, as *threads* podem aumentar a complexidade do algoritmo (dependendo do problema), sendo possível não ser obtido ganho de desempenho nas aplicações. Não somente o aumento da complexidade do algoritmo em si, mas também da complexidade de sincronização dos dados [PRA97] são fatores desfavoráveis ao desempenho. Outro aspecto a ser ponderado, a respeito de *threads*, é a dificuldade de depuração do programa.

Segundo Prasad [PRA97], no desenvolvimento de programas *multithreaded*, a sincronização de dados pode causar gargalos, especialmente em máquinas com memória compartilhada. Isso faz perder os benefícios mais importantes das *threads*, que é a execução concorrente.

Em algumas aplicações, onde o problema é dividido em componentes menores e esses executados por *threads*, o tempo gasto na execução com memória compartilhada pode tornar-se maior que o tempo exigido para resolver o problema real. O ideal é ter-se um método eficiente de comunicação entre *threads* e um algoritmo favorável à execução das mesmas – que não exija demasiada sincronização.

Em máquinas com memória compartilhada, as *threads* são usadas para tirar proveito da capacidade de todos os processadores da máquina. Existindo várias *threads*, o sistema operacional encarrega-se de escaloná-las entre os processadores que estão compartilhando a memória.

Para o caso do *cluster*, os algoritmos paralelos executados nele podem ser formulados sob a ótica da troca de mensagens e da utilização de *threads*. Na seção 5.2.4, o algoritmo descrito foi baseado em trocas de mensagens.

Para a execução paralela do Gradiente Conjugado, com memória compartilhada, *threads* foram utilizadas. O algoritmo paralelo possui todas as funções para execução em ambiente de troca de mensagens entre nodos, além de utilizar as *threads*. Como a cada nodo foi designado uma porção de elementos dos vetores e linhas da matriz do sistema linear, esses dados foram divididos pelo número de *threads* utilizadas, proporcionando um paralelismo ainda maior.

Para cada operação de álgebra linear, foram utilizadas *nth threads*, onde *nth* é o número de *threads*, e cada uma delas manipula uma parte do vetor e da matriz. Por exemplo, na soma de vetores, ao invés de uma única *thread* calcular o vetor resultante, tem-se *nth threads*, cada uma calculando uma porção diferente do vetor. Se, por exemplo, o nodo possui *k* elementos do vetor, cada *thread* executará a operação sobre *k/nth* elementos do vetor. Na figura 5.14, a soma de vetores é ilustrada utilizando-se duas *threads*.

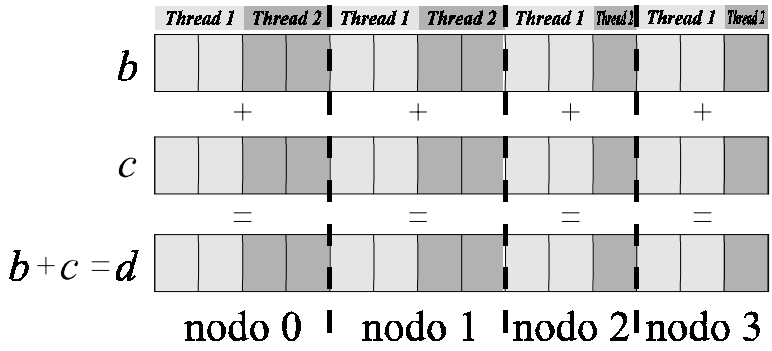


FIGURA 5.14 - Representação da Soma de Vetores com *Threads*

Teoricamente, isso diminui o tempo de execução proporcionalmente ao número de processadores que estão compartilhando a memória. Para o caso do *cluster*, onde os nodos possuem dois processadores, o tempo de execução com *threads* deveria ser reduzido pela metade, se não houvesse *overhead* no sistema.

No algoritmo paralelo do Gradiente Conjugado com *threads*, há necessidade das *threads* serem sincronizadas, toda vez que há comunicação entre os processadores, por trocas de mensagens. Com base no Gradiente Conjugado paralelo descrito na seção

5.2.4, as sincronizações das *threads* necessárias são nos passos 6, 8 e 14 devido ao produto escalar entre vetores e nos passos 4, 8 e 11, para a multiplicação de matriz esparsa por vetor. No capítulo 6, a experiência de utilização de *threads* é analisada a partir das medidas de desempenho obtidas.

O DECK permite a sincronização de todos os processos através da função *deck_barrier*. Para a sincronização de *threads*, há funções de manipulação de semáforos, como criação, bloqueio, liberação e destruição de semáforos. Para a sincronização de todas as *threads*, há a função *deck_thread_barrier*, simplificando o trabalho, por não ser necessário o uso de semáforos pelo usuário, nesse caso.

5.2.6 Algoritmo Paralelo do Gradiente Conjugado Pré-Condicionado

No capítulo 3, seção 3.4.2, foi descrito o que é pré-condicionamento e caracterizados os pré-condicionadores Diagonal e Polinomial. Nessa seção, são descritas a geração do pré-condicionador. A técnica de pré-condicionamento consiste em transformar o sistema linear em um outro sistema com a mesma solução, mas com propriedades mais favoráveis à convergência.

Como entrada, no algoritmo paralelo Gradiente Conjugado Pré-condicionado (PCG), tem-se a matriz e vetores do sistema linear e a matriz pré-condicionadora. Esse algoritmo inclui, pelo menos, duas multiplicações matriz por vetor a mais que o Gradiente Conjugado sem pré-condicionador. Uma delas está no passo 5 do algoritmo da seção 5.2.4, onde o vetor *d* recebe um vetor resultante do produto da matriz pré-condicionadora pelo vetor *r*. A outra multiplicação está nas iterações, no passo 14 do mesmo algoritmo. A operação $\delta_{novo} = r^T r$, do Gradiente Conjugado sem pré-condicionador, é substituída por $\delta_{novo} = r^T s$, onde s é o vetor resultante da multiplicação do pré-condicionador e o vetor *r*.

5.2.6.1 Pré-Condicionador Diagonal

O pré-condicionador Diagonal é gerado a partir da diagonal principal da matriz dos coeficientes do sistema linear (seção 3.4.2.1). A matriz pré-condicionadora, portanto, possui apenas a diagonal principal, sendo armazenada em um vetor. A operação de multiplicação dessa matriz pelo vetor torna-se simplificada, bastando multiplicar cada elemento correspondente dos vetores (vetor que armazena a matriz e vetor que está sendo multiplicado), para ser calculado o elemento do vetor resultante dessa operação.

Dessa forma, na versão do Gradiente Conjugado Pré-Condicionado com o pré-condicionador Diagonal, nenhuma operação de comunicação entre os processadores é adicionada. No capítulo 6, onde os experimentos são relatados saber-se-á se realmente esse pré-condicionador proporciona ganho de desempenho.

5.2.6.2 Pré-Condicionador Polinomial

O pré-condicionador Polinomial é gerado a partir de Séries Truncadas de Neumann (seção 3.4.2.2). A estrutura da matriz pré-condicionadora não é mantida, ou seja, essa matriz não permanece tridiagonal, para graus de polinômio maior que 1. Esse

grau determina o número de multiplicações matriz por matriz, influenciando diretamente no número de diagonais que a matriz resultante possuirá.

Como o número de diagonais do pré-condicionador varia, a utilização de uma estrutura de armazenamento otimizada é dificultada. O pré-condicionador polinomial é gerado pela equação 3.21, na seção 3.4.2.2, que é transcrita a seguir:

$$C^{-1} = \left(\sum_{i=0}^{m} \left(I - P^{-1}A \right)^{i} \right) P^{-1}, \qquad \text{onde } P^{-1} = \left(diag(A) \right)^{-1} \text{ e } m \text{ é o grau do polinômio}$$

Para o polinômio com grau *zero*, C^{-1} é uma matriz diagonal, equivalente ao pré-condicionador Diagonal. Para grau *um*, C^{-1} é uma matriz tridiagonal, pois a matriz *A* da equação é tridiagonal. Para grau *dois*, C^{-1} é uma matriz pentadiagonal, porque multiplicando duas matrizes tridiagonais, a resultante é uma com cinco diagonais. Para grau *três*, C^{-1} é uma matriz com sete diagonais e assim sucessivamente.

Dessa forma, para matrizes tridiagonais, sendo *gp* o grau do polinômio e *nd* o número de diagonais da matriz, o número de diagonais da matriz pré-condicionadora é dado por:

$$nd = (gp * 2) + 1 \qquad (5.5)$$

Com isso pode-se utilizar um formato otimizado para armazenar o pré-condicionador. Gerar o pré-condicionador utilizando matrizes densas seria inviável pelo grande porte dos sistemas lineares e também pelas multiplicações de matrizes existentes no algoritmo do pré-condicionador polinomial.

No algoritmo do Gradiente Conjugado Pré-Condicionado, a operação de multiplicação matriz esparsa por vetor sofre diretamente a influência do grau do polinômio. Facilitada estaria a multiplicação se fosse possível armazenar a matriz no formato denso. Porém, isso não é possível e a multiplicação matriz esparsa por vetor, descrita em 5.2.3.4 genericamente, foi utilizada também no algoritmo paralelo.

No algoritmo paralelo de multiplicação matriz esparsa (genérica) por vetor, que é usado no Gradiente Conjugado Pré-Condicionado, como não é necessário o vetor inteiro em cada nodo, somente as partes necessárias são envolvidas na comunicação. A comunicação par-ímpar atualiza o vetor que multiplicará a matriz, em cada nodo, e o número de elementos a serem enviados é definido pelo número de diagonais da matriz pré-condicionadora, como caracterizado na seção 5.2.2.5.

5.3 Considerações Finais

Nesse capítulo foi tratada a paralelização dos algoritmos dos métodos de resolução de sistemas lineares. O método iterativo do Gradiente Conjugado mostrou-se muito propício a ser paralelizado, sendo aplicado diretamente paralelismo de dados em cada operação envolvida no mesmo. Já o método do Algoritmo de Thomas não possibilitou a mesma abordagem, sendo buscada uma forma alternativa.

O método do Gradiente Conjugado Pré-condicionado permitiu que fossem investigadas formas de generalizar o processo de criação do pré-condicionador polinomial, no sentido do armazenamento da matriz e na multiplicação matriz esparsa vetor, fazendo com que fosse possível variações no grau do polinômio. Essas variações permitirão uma melhor observação dos resultados da utilização de pré-condicionadores.

Assim como no desenvolvimento de algoritmos paralelos com trocas de mensagens, alguns aspectos devem ser observados no desenvolvimento de algoritmos com *threads*, como por exemplo, o tipo de aplicação com a qual se está trabalhando, a arquitetura da máquina e a ferramenta que será utilizada para programação paralela. Questões como dependência de dados, sincronizações e o uso ou não de memória compartilhada, influenciam diretamente na obtenção de desempenho pela utilização de *threads*.

Para avaliar o desempenho dos algoritmos paralelos do Gradiente Conjugado, Gradiente Conjugado Pré-Condicionado, Gradiente Conjugado com *threads* e Algoritmo de Thomas no *cluster,* várias medidas foram realizadas. No próximo capítulo, são apresentados os dados coletados com essas medidas e os principais resultados podendo-se, também, observar o desempenho na utilização do *cluster* e do DECK.

6 Avaliação da Paralelização

A computação paralela tem emergido como uma ferramenta indispensável para alcançar alto desempenho na resolução de problemas em muitas áreas científicas, nos últimos 20 anos. Somente paralelizar aplicações é insuficiente, é necessário avaliá-las para identificar os benefícios obtidos, perceber pontos críticos e quais procedimentos podem ser adotados para um melhor desempenho ser alcançado.

A avaliação do desempenho de sistemas paralelos tem por objetivo verificar o desempenho de um sistema ou uma aplicação. Também permite investigar a relação entre as exigências da aplicação e as características da arquitetura da máquina paralela, identificar aspectos que influenciam no tempo de execução da aplicação, prever o desempenho e avaliar formas distintas de paralelização de aplicações paralelas [CRE99].

Como forma de avaliar a paralelização dos métodos de resolução são considerados a complexidade dos algoritmos e medidas empíricas. A avaliação sob aspectos teóricos diz respeito ao processo de determinação de quão bom é um algoritmo, em termos de número de operações executadas pelo mesmo. As medidas empíricas resumem o comportamento da aplicação quando executada em um sistema real ou simulado. Essas últimas também descrevem o ganho ou perda de desempenho em relação ao número de recursos alocados.

6.1 Aspectos Teóricos

Através da complexidade de tempo de um algoritmo pode-se determinar como será a execução desse algoritmo e de que forma ele fará uso dos recursos disponíveis, ou seja, quão eficiente ele será. A complexidade de tempo (ou complexidade) é uma função que associa a um tamanho de entrada do algoritmo uma medida do número de passos que são executados pelo algoritmo [TER90].

Ao ser buscado um algoritmo para determinado problema, prefere-se aquele de menor ordem de complexidade. Por exemplo, um algoritmo O(log n) é preferível a um O(n), que por sua vez é preferível a um O(n log n), que é preferível a um O(n^2).

Para algoritmos seqüenciais, basta considerar o número de operações executadas. Já para algoritmos paralelos, adicionalmente ao número de operações, é necessário considerar as operações de comunicação existentes e o número de processadores envolvidos na aplicação.

Ao avaliar a complexidade de um algoritmo, é convencionado que n é o tamanho da entrada do algoritmo, ou seja, a ordem do sistema linear e p o número de nodos usados. Nessa avaliação são observados os seguintes aspectos, conforme Akl [AKL89]:

a) Número de Passos: número de operações básicas lógicas ou aritméticas executadas pelo algoritmo. Esse número é descrito como uma função do tamanho da entrada (n) do algoritmo. O número de passos é identificado por $t_{operação}$.

b) Número de Passos na Comunicação: número de operações de comunicação executadas pelo algoritmo. É identificado por $t_{comunicação}$. No *cluster* cada nodo tem comunicação direta com todos os demais nodos. Dessa forma, considera-se que a comunicação de um escalar ocorra em uma unidade de tempo. O número de operações poderá ser uma função do tamanho da entrada ou, dependendo da situação, uma função do número de processadores participantes da aplicação.

c) Número Total de Operações: t_{total} é a soma do $t_{operação}$ e do $t_{comunicação}$.

d) Número de Processadores: número de nodos utilizados na aplicação (p). Geralmente esse número independe do tamanho da entrada do algoritmo.

e) Custo: é o produto do Número Total de Operações e o Número de Processadores utilizados. O custo (c) é o número de passos executados coletivamente por todos os processadores na solução de um determinado problema. Um algoritmo paralelo é ótimo se o seu custo (que identifica sua complexidade) é menor ou igual que a complexidade do algoritmo seqüencial [AKL89].

Deve-se observar que a soma de complexidades, no pior caso, tem como resultado a complexidade de maior ordem. A complexidade de maior ordem absorve a complexidade de ordem menor, significando que a operação de maior custo computacional influencia diretamente a complexidade final do algoritmo.

Como o Gradiente Conjugado é um método iterativo, a complexidade, tanto do algoritmo seqüencial como do paralelo é observada na iteração [WES98] e não no algoritmo como um todo, uma vez que o número de iterações é dependente da convergência do sistema linear e não do tamanho da entrada. No Algoritmo de Thomas, que é um método direto, a complexidade é do algoritmo como um todo, pois o método executa um número pré-definido de passos.

6.1.1 Algoritmo de Thomas

O método do Algoritmo de Thomas seqüencial tem complexidade de tempo $O(n)$ [FLE88, WES68]. Esse método, por pertencer a classe dos diretos, executa um número pré-definido de repetições, que corresponde à ordem do sistema linear que está sendo resolvido.

No algoritmo distribuído são necessárias comunicações. Aproximadamente $2(p - 1)$ trocas de mensagens são feitas, sendo que 50% dessas são de apenas um escalar e a outra metade das mensagens possui dois escalares. Assim, o tempo total de comunicação é $3(p - 1)$, implicando em $t_{comunicação} = O(p - 1)$.

Cada nodo executa n/p operações. Porém, essa execução não é simultânea entre os nodos, então o $t_{operação}$ não é $O(n/p)$, mas sim $t_{operação} = O(n)$ já que $n/p*p = n$. Isso ocorre pelas próprias características do algoritmo, especialmente pela existência de dependências de dados. Se fosse considerada a execução de vários sistemas lineares de forma *pipeline* haveria sobreposição de operações e, consequentemente, ganho de desempenho.

O número total de operações é $t_{total} = O(n) + O(p - 1)$. Como $p < n$, a complexidade da comunicação é menor que a complexidade das operações, assim $t_{total} =$

O(n). O custo é obtido pela multiplicação do número de processadores e do número total de operações, resultando em um custo O($n\ p$).

O custo comparado com a complexidade do algoritmo seqüencial demostra que o algoritmo não é ótimo, pois a complexidade do algoritmo seqüencial é menor que a do custo: O(n) < O($n\ p$). Pode-se antecipar que nesse algoritmo quanto maior for o número de nodos utilizados, maior será o tempo de execução.

6.1.2 Gradiente Conjugado

Como o algoritmo do método do Gradiente Conjugado é formado basicamente por operações de álgebra linear, essas operações são analisadas separadamente. Após, o algoritmo como um todo é avaliado.

a) Soma / Subtração de Vetores e Multiplicação de Escalar por Vetor: essas operações não exigem comunicação ($t_{comunicação}$ = O(1)). Com os dados dos vetores nos respectivos nodos, as somas/subtrações são feitas seqüencialmente em cada nodo. Portanto, com p processadores, cada um executa n/p operações ($t_{operação}$ = O(n/p)), logo, t_{total} = O(n/p). O custo dessa operação é c_{soma} = O(n) pois $n/p * p = n$, sendo 100% paralelizável e com complexidade igual à complexidade do seu algoritmo seqüencial.

b) Produto Escalar: o produto escalar seqüencial tem complexidade O(n). No produto escalar paralelo, cada nodo executa n/p operações ($t_{operação}$ = O(n/p)). Com relação à comunicação, é necessária uma operação de redução que envolve (p – 1) comunicações. Cada comunicação envia um único escalar. Há também um *broadcast* que envolve (p – 1) comunicações, também de um escalar ($t_{comunicação}$ = O(p –1)). Assim, o número total de operações nesse algoritmo é t_{total} = O(n/p) + O(p – 1). Como $p < n$, o primeiro termo da expressão anterior tem complexidade maior, logo: t_{total} = O(n/p), $c_{produto\ escalar}$ = O(n) resultando em uma operação paralela ótima.

c) Multiplicação Matriz Esparsa por Vetor: na multiplicação seqüencial matriz densa por vetor, o algoritmo trivial tem complexidade O(n^2) [AKL89]. Se a matriz é esparsa, a complexidade dependerá da estrutura da matriz do sistema linear. Dessa forma, a complexidade será o número de elementos não nulos da matriz [SHE94]. A complexidade da multiplicação de matriz esparsa seqüencial, onde os elementos não nulos se encontram em algumas diagonais, sendo n a ordem da matriz e nd o número de diagonais, é O($n\ nd$).

   Na multiplicação paralela, cada nodo executará $n*nd/p$ operações, sendo $t_{operação}$ = O($n\ nd/p$). Entretanto, deve-se considerar as operações de comunicação. A comunicação Par-Impar é necessária antes da multiplicação para que seja atualizado o vetor que está multiplicando. Nessa operação, o número de elementos a serem enviados em cada troca de mensagem é constante, sendo (nd –1)/2. Como são realizadas (p – 1)*2 comunicações, o $t_{comunicação}$ = O((nd –1) (p – 1)). O número total de operações do algoritmo é t_{total} = O($n\ nd/p$) + O((nd –1) (p – 1)). Com $p < n$ e $nd < n$, a complexidade predominante é o primeiro termo da expressão anterior, resultando em t_{total} = O($n\ nd/p$). O custo dessa operação é $c_{matriz\ vetor}$ = O($n\ nd$), igual à complexidade da multiplicação seqüencial, tornando essa operação paralela vantajosa.

Para ser obtida a complexidade de cada iteração do algoritmo paralelo do Gradiente Conjugado, devem ser somadas todas as complexidades paralelas das operações que compõem o método, descritas anteriormente. Assim:

$$c_{CG} = c_{soma} + c_{subração} + c_{escalar\ vetor} + c_{produto\ escalar} + c_{matriz\ vetor}$$

onde c_{soma}, $c_{subração}$, $c_{escalar\ vetor}$, $c_{produto\ escalar}$ e $c_{matriz\ vetor}$ são respectivamente as complexidades das operações de soma, subtração e multiplicação escalar por vetor, produto escalar de dois vetores e a multiplicação matriz por vetor.

$$c_{CG} = O(n) + O(n) + O(n) + O(n) + O(n\ nd)$$

$$c_{CG} = O(n\ nd)$$

A operação predominante nesse algoritmo é a multiplicação paralela matriz esparsa por vetor, que determina a complexidade do algoritmo. Comparados os custos do algoritmo paralelo e a complexidade do algoritmo seqüencial conclui-se que o algoritmo paralelo do Gradiente Conjugado é um algoritmo ótimo, pois tem a mesma complexidade do algoritmo seqüencial.

Pode-se ainda concluir que se o número de diagonais for muito menor que a ordem da matriz ($nd << n$) a complexidade do Gradiente Conjugado paralelo é $O(n)$. Por exemplo, para sistemas tridiagonais, $nd = 3$ e $c_{CG} = 3n = O(n)$.

6.2 Aspectos Empíricos

A avaliação da paralelização dos métodos de resolução de sistemas lineares esparsos também é feita sob os aspectos empíricos. As medidas são baseadas no tempo de execução que tem sido a medida mais utilizada para avaliar o desempenho de aplicações [CRE99].

O tempo de execução é o tempo total de processamento. Para a tomada dessa medida, o tempo inicial é o tempo em que o primeiro nodo iniciou o processamento e o tempo final é o tempo em que o último nodo terminou o processamento.

O *speedup* (Sp) é muito utilizado pois fornece a medida de ganho de desempenho da aplicação paralela em relação à aplicação seqüencial. Segundo Cremonesi [CRE99], essa medida consegue capturar todos os efeitos de fatores característicos da computação paralela, como por exemplo *overhead* e gargalos do sistema.

O *speedup* é obtido através da razão do tempo de execução seqüencial da aplicação ou o tempo de execução em um único nodo pelo tempo de execução paralelo. O *speedup* de um sistema ideal é igual ao número de nodos utilizados [CRE99, KUM94, JAJ92].

$$Sp = \frac{T_{(1)}}{T_{(Nnodes)}} \tag{6.1}$$

onde Sp é o *speedup*, $T_{(1)}$ é o tempo de execução do algoritmo paralelo em um único nodo, $T_{(Nnodes)}$ é o tempo de execução paralelo e *Nnodes* é o número de nodos usados na aplicação.

A eficiência representa a fração de tempo em que os processadores são empregados na resolução do problema [CRE99, KUM94, JAJ92]. Essa medida equivale

ao desempenho em cada nodo e é obtida pela razão do *speedup* e o número de nodos. A eficiência ideal é igual a 1.

$$Ef = \frac{Sp}{Nnodes} \tag{6.2}$$

onde Ef é a eficiência, Sp é o *speedup* e $Nnodes$ o número de nodos usados na aplicação.

Toda essa avaliação é realizada com variações no tamanho da entrada do algoritmo, que é a ordem do sistema linear, e no número de nodos utilizados. Poder-se-á observar onde haverá ganho de desempenho, a partir desses parâmetros.

Na execução com até quatro nodos, o *cluster* é homogêneo, ou seja, todos os nodos possuem a mesma configuração. Com um número maior de nodos, tem-se um *cluster* heterogêneo, o que influenciará no desempenho dos algoritmos.

Para serem calculadas as medidas descritas anteriormente, especialmente para o *speedup*, utilizando até quatro nodos, tomou-se como tempo de execução seqüencial a execução do algoritmo paralelo em um único nodo Dual Pentium, que é a mesma configuração dos demais nodos. Para o *cluster* heterogêneo, onde os nodos variam até nove, as medidas foram calculadas conforme sugere Donaldson [DON94]: o tempo de execução seqüencial ou tempo de execução em um único nodo é medido no processador mais rápido disponível. Se esse tempo fosse tomado em um nodo Dual Pentium, as medidas apontariam um ganho muito maior que o real, uma vez que essas máquinas não são as mais rápidas do *cluster* e ter-se-ia um ponto de referência que maximiza o desempenho.

Se o *speedup* obtido é maior que o número de nodos, ou seja, ultrapassou o *speedup* ideal, a esse *speedup* denomina-se *speedup* superlinear. Ao ser aumentado o número de nodos, acima de um determinado número, o desempenho da aplicação tende a cair, devido ao *overhead* do sistema. Nessa situação tem-se o chamado *speed down* [FOS94].

6.2.1 Algoritmo de Thomas

As medidas do tempo de execução do método de resolução de sistemas lineares esparsos Algoritmo de Thomas foram realizadas na rede *Fast-Ethernet*, utilizando-se de um a nove nodos do *cluster* e com variações no tamanho da entrada do sistema linear. Essas variações ocorreram a partir de uma ordem do sistema de 5.000 até 600.000.

O gráfico da figura 6.1 mostra o tempo de execução desse algoritmo, para os vários tamanhos de entrada e nodos. Como o Algoritmo de Thomas é um algoritmo distribuído onde o paralelismo é inexistente, a execução no *cluster* não é vantajosa pois o tempo de execução não sofre redução.

$$pep_j = \sum_{i=1}^{ne} b_i * c_i$$

FIGURA 6.1 – Tempo de Execução do Algoritmo de Thomas

A comunicação necessária a esse método é constante (independe da ordem do sistema linear) e pequena. Por esse fato e pela alta velocidade da rede de interconexão não houve aumento significativo do tempo de execução, quando aumentado o número de nodos. Por outro lado, não houve ganho de desempenho, pois em nenhum caso o *speedup* obtido foi maior que *um*, demonstrando a influência direta do tempo ocioso dos processadores.

O *speedup* do Algoritmo de Thomas pode ser observado nos gráficos das figuras 6.2 e 6.3. Esses resultados são para o *cluster* homogêneo, ou seja, sendo utilizados no máximo os quatro nodos Dual Pentium.

$$c_i = k * b_i$$

FIGURA 6.2 – *Speedup* do Algoritmo de Thomas - Ordem do sistema 5.000 a 70.000

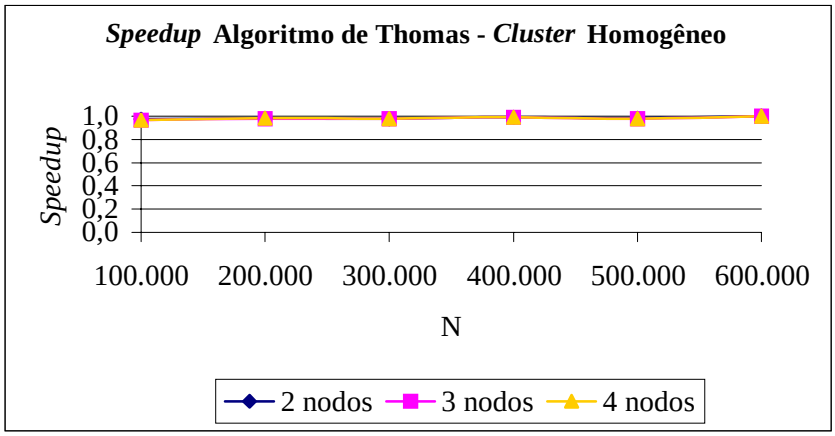


FIGURA 6.3 – *Speedup* do Algoritmo de Thomas
Ordem do sistema 100.000 a 600.000

Com relação à eficiência, quanto maior o número de nodos alocados, maior é a ineficiência do método. Os maiores valores de eficiência obtidos foram próximos a 0,5 para sistemas lineares com ordem superior a 20.000 e dois nodos. Os gráficos das figuras 6.4 e 6.5 apresentam os resultados do cálculo da eficiência, para as mesmas ordens de sistemas lineares do *speedup*.

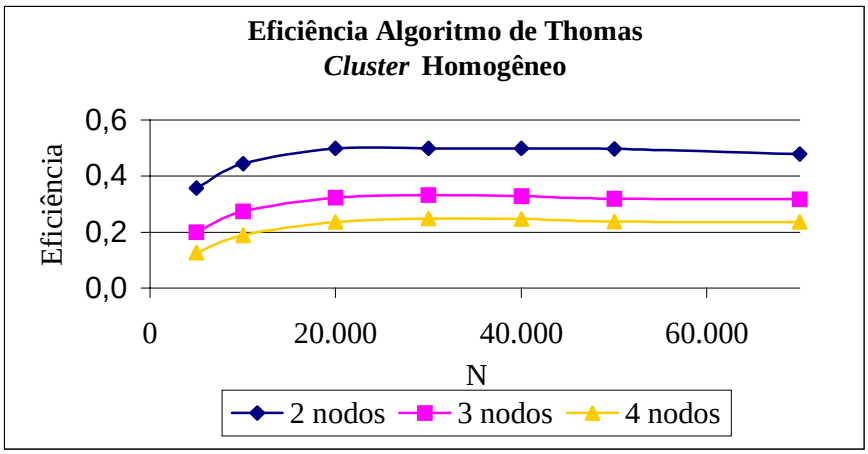


FIGURA 6.4 – Eficiência do Algoritmo de Thomas - Ordem do sistema 5.000 a 70.000

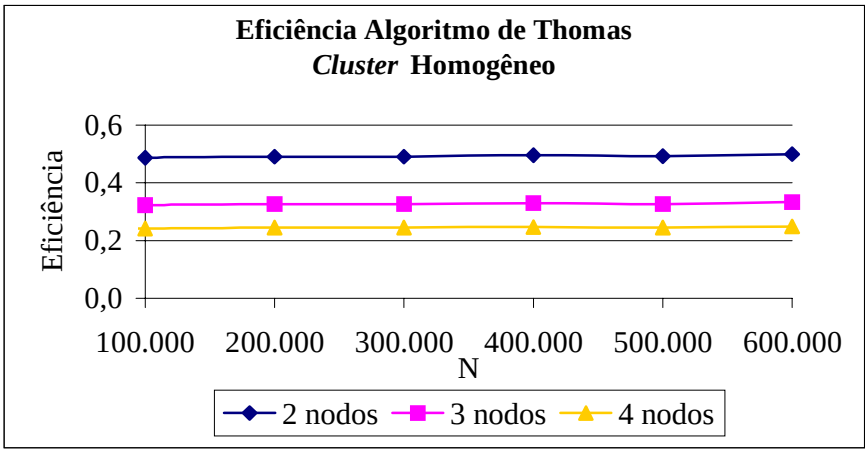


FIGURA 6.5 – Eficiência do Algoritmo de Thomas
Ordem do sistema 100.000 a 600.000

São apresentados somente os gráficos dos resultados no *cluster* homogêneo, uma vez que não houve desempenho em nenhum dos casos. Para o *cluster* heterogêneo (sendo utilizados até 9 nodos), essa situação se repete, pois para esse caso, como será visto no método do Gradiente Conjugado, o desempenho é um pouco menor que no *cluster* homogêneo.

6.2.2 Gradiente Conjugado

O método do Gradiente Conjugado paralelo foi avaliado de forma semelhante ao Algoritmo de Thomas. As medidas do tempo de execução foram realizadas com diferentes ordens dos sistemas lineares (tamanho da entrada) e grupos distintos de nodos, formados por até nove nodos.

O gráfico da figura 6.6 ilustra o tempo de execução, em segundos, do método do Gradiente Conjugado para sistemas lineares de ordem 50.000, 100.000, 300.000 e 600.000. Em geral, há diminuição do tempo de execução em paralelo, conforme é aumentado o número de nodos utilizados.

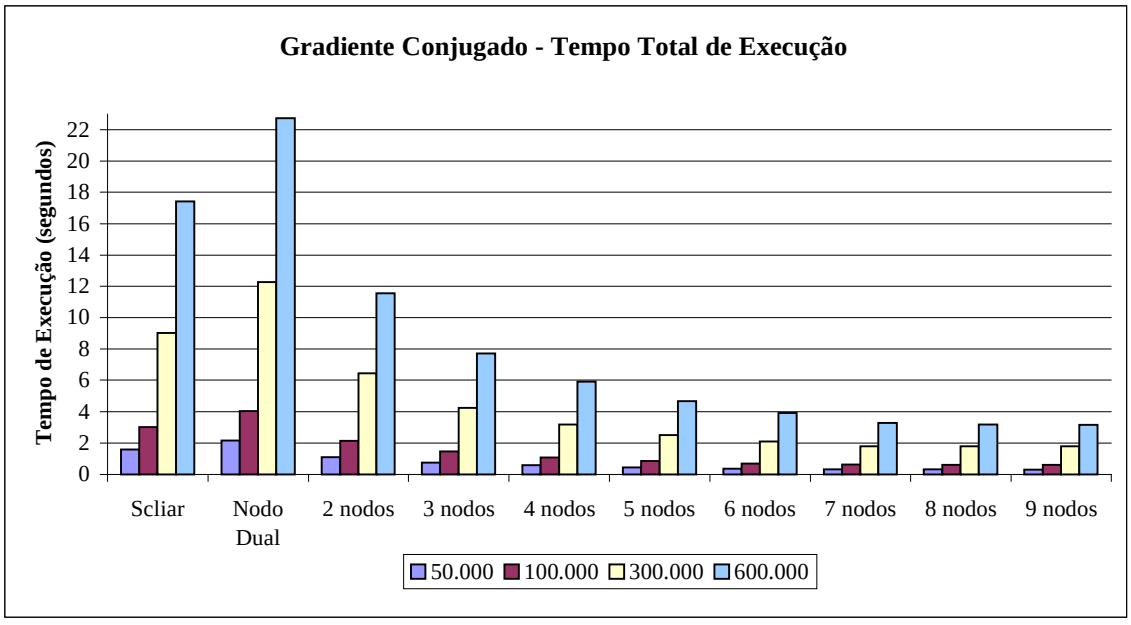


FIGURA 6.6 – Tempo de Execução do Gradiente Conjugado

Observa-se em cada grupo de nodos que o tempo de execução paralela é muito próximo do resultado da divisão do tempo de execução seqüencial pelo número de nodos usados, nos grupos de até 5 nodos. Entre 7, 8 e 9 nodos alocados, a diferença entre os tempos de execução paralela diminui porque nos grupos de 8 e 9 nodos estão incluídas as duas máquinas de menor capacidade de processamento do *cluster*.

A partir do gráfico anterior percebe-se que houve ganho de desempenho com a paralelização do Gradiente Conjugado. Para ser conhecido quanto é o ganho da paralelização em relação à execução seqüencial calcula-se o *speedup*. Os gráficos das figuras 6.7 e 6.8 representam o *speedup* para o *cluster* homogêneo.

Considera-se *cluster* homogêneo quando estão sendo utilizados até quatro nodos Dual Pentium. Logo, por *cluster* heterogêneo entende-se a utilização de até nove nodos, com configurações distintas, na execução das aplicações.

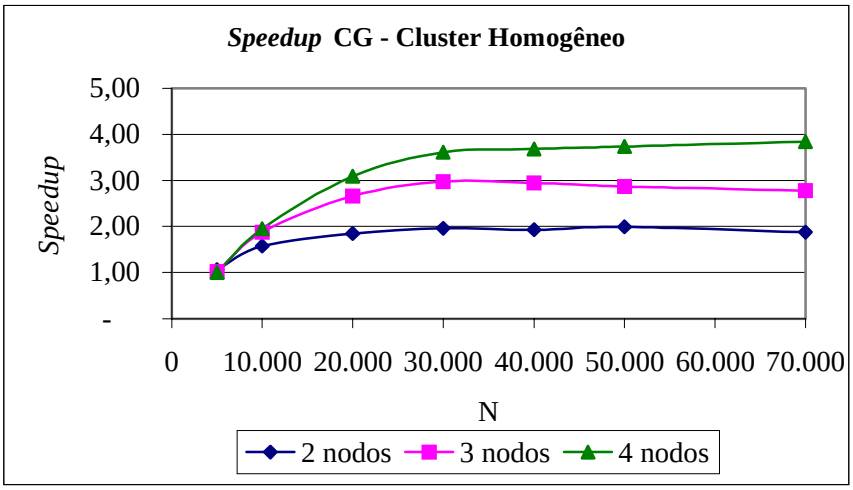


FIGURA 6.7 – *Speedup* do CG no *Cluster* Homogêneo
Ordem do Sistema Linear 5.000 a 70.000

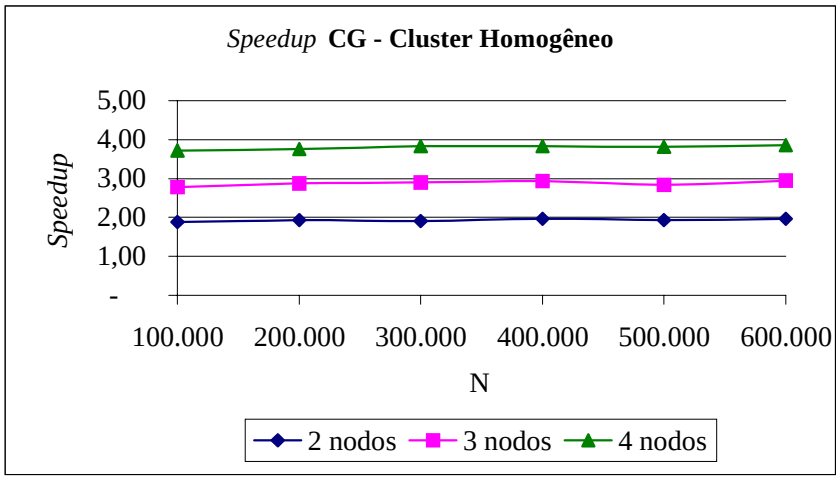


FIGURA 6.8 - *Speedup* do CG no *Cluster* Homogêneo
Ordem do Sistema Linear 100.000 a 600.000

A partir de uma entrada de 5.000 (ordem do sistema linear) tem-se ganho de desempenho para qualquer grupo de nodos do *cluster* homogêneo, pois o *speedup* é maior que *um*. Com dois nodos, o *speedup* atingido por sistemas lineares de ordem superior a 25.000 é muito próximo ou igual ao *speedup* ideal (igual ao número de nodos). Para três nodos, a partir de sistemas lineares de ordem superior a 30.000 o *speedup* é igual ou próximo ao *speedup* ideal. Para quatro nodos, devido ao aumento de comunicação, a partir de sistemas lineares com ordem superior a 70.000, o *speedup* é próximo ao *speedup* ideal.

A eficiência mostra como estão sendo utilizados os nodos na execução paralela. Os gráficos de eficiência para o *cluster* homogêneo estão nas figura 6.9 e 6.10.

A eficiência ideal é igual a *um*, ou seja, os recursos disponíveis são 100% utilizados. Para sistemas lineares com ordem igual ou superior a 30.000, para todos os grupos de nodos utilizados, a eficiência obtida é muito próxima da ideal (pelo menos 0,9). Com entrada a partir de 10.000, a menor eficiência obtida foi 0,5 para quatro

nodos. Entradas de tamanho inferior mostram ter execução paralela ineficiente, considerando-se o aproveitamento dos recursos computacionais.

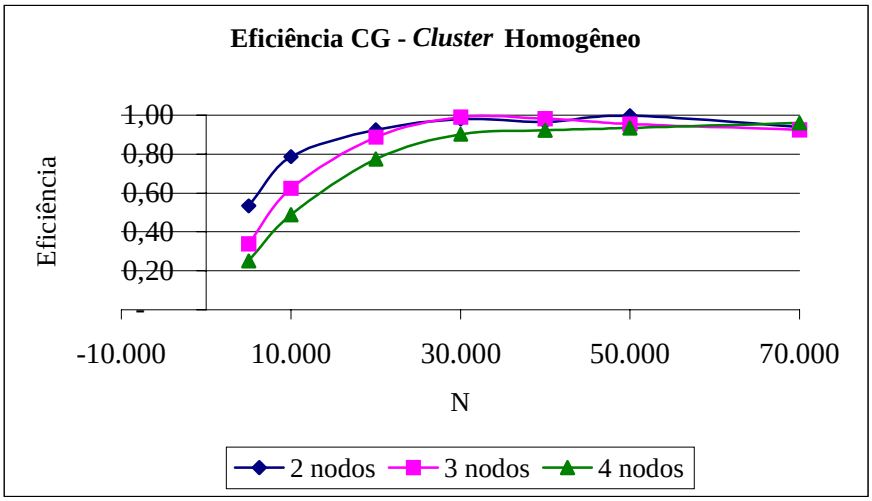


FIGURA 6.9 – Eficiência do CG no *Cluster* Homogêneo
Ordem do Sistema Linear 5.000 a 70.000

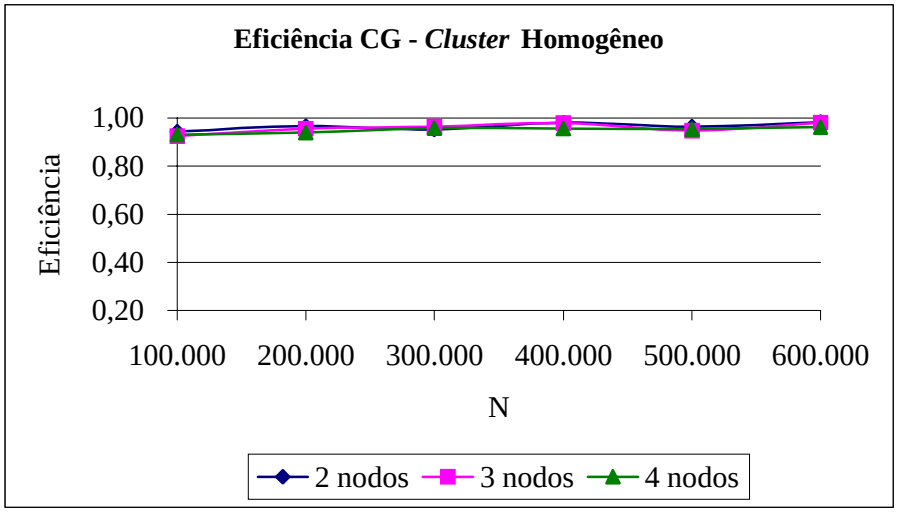


FIGURA 6.10 – Eficiência do CG no *Cluster* Homogêneo
Ordem do Sistema Linear 100.000 a 600.000

Para o cálculo das medidas no *cluster* heterogêneo, considerou-se como tempo de execução do algoritmo em um único nodo, a execução na máquina *Scliar*. Essas medidas demonstrarão que o desempenho obtido, em relação ao *cluster* homogêneo, é um pouco menor.

No *cluster* heterogêneo, a partir de um sistema linear com ordem próxima a 10.000 se ganha desempenho, pois o *speedup* é maior que *um*. Isso pode ser observado na figura 6.11.

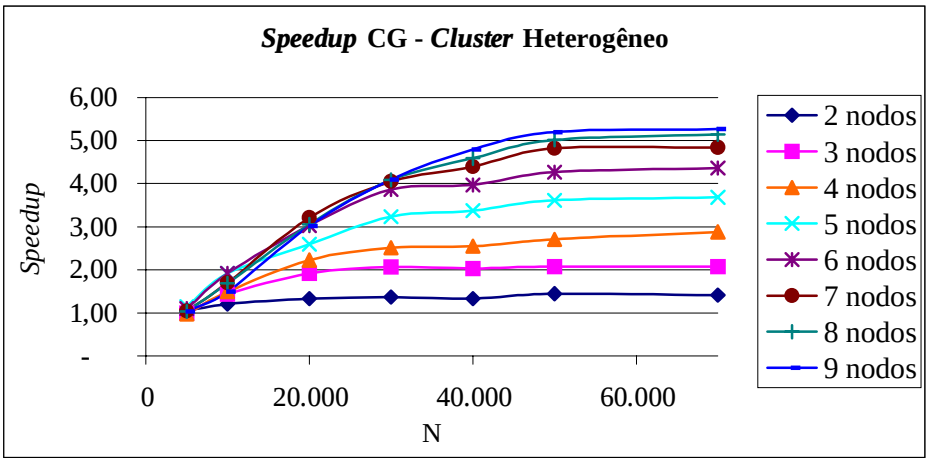


FIGURA 6.11 – *Speedup* do CG no *Cluster* Heterogêneo
Ordem do Sistema Linear 5.000 a 70.000

Na figura 6.12 está o gráfico do *speedup* do Gradiente Conjugado para entradas maiores que 100.000. Em termos de ganho de desempenho, comparado aos casos do *cluster* homogêneo, esse é menor, pois conforme aumentam o número de nodos no grupo, mais distante é o *speedup* obtido do *speedup* ideal.

Nos grupos de 8 e 9 nodos o desempenho diminui porque nesses são alocadas as máquinas mais lentas do cluster. Isso acontece pela distribuição dos dados entre os nodos ser homogênea, ou seja, independente da velocidade de processamento de cada nodo, e como o algoritmo paralelo do Gradiente Conjugado é síncrono durante a execução acontece um pequeno tempo de espera pelo processador mais lento.

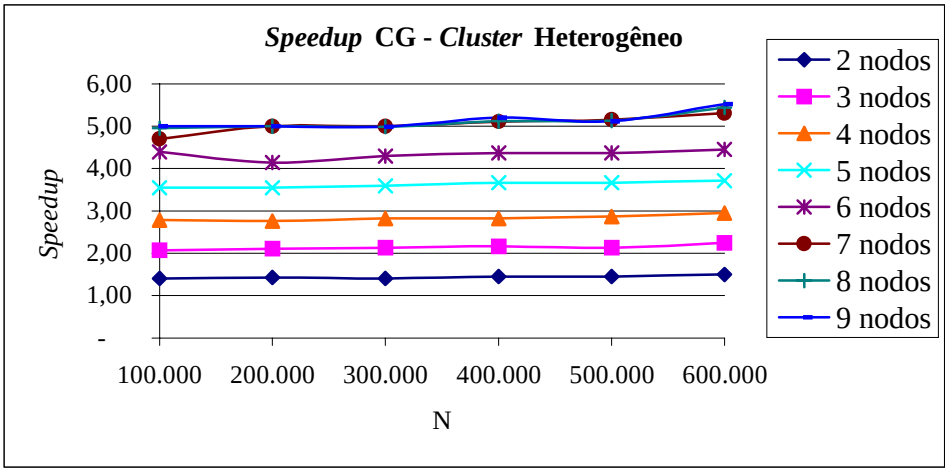


FIGURA 6.12 – *Speedup* do CG no *Cluster* Heterogêneo
Ordem do Sistema Linear 100.000 a 600.000

A eficiência permite melhor observar o aproveitamento dos recursos do *cluster* heterogêneo. As figuras 6.13 e 6.14 trazem os gráficos de eficiência. O valor máximo de eficiência obtido foi próximo a 0,8, com o grupo de 7 nodos e um sistema linear de ordem 600.000.

Com sistemas lineares de ordem superior ou igual a 40.000, para qualquer grupo de nodos, tem-se eficiência de pelo menos 0,5. Aumentando o tamanho da entrada (até 100.000), a eficiência aumenta, pois há uma quantidade maior de processamento do que de operações de comunicação.

Depois de determinada ordem do sistema linear, a eficiência é mantida. Isso ocorre por ter-se alcançado o máximo de utilização do sistema. Mesmo não sendo a eficiência ideal, esse é o máximo de aproveitamento dos recursos, limitado muitas vezes pelo número de nodos, tamanho da entrada, tempos de comunicação e/ou tempos de espera do sistema.

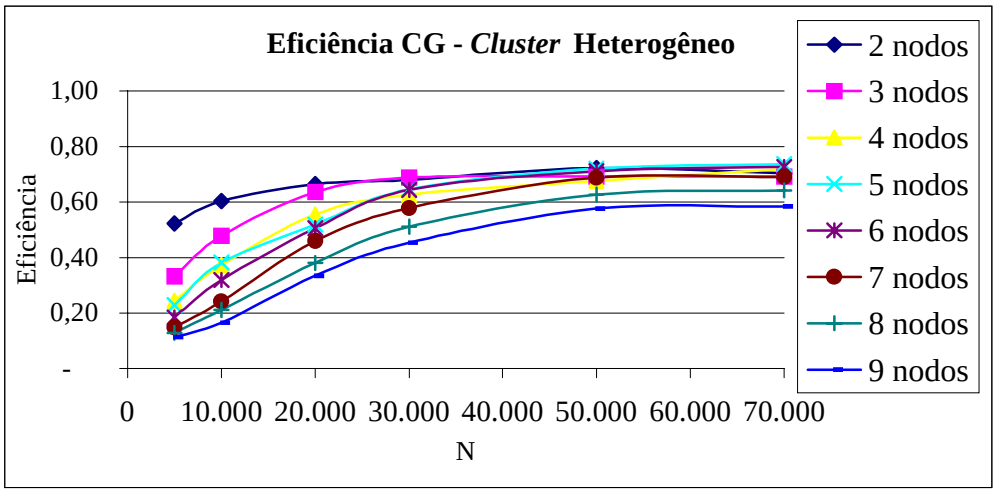


FIGURA 6.13 – Eficiência do CG no *Cluster* Heterogêneo
Ordem do Sistema Linear 5.000 a 70.000

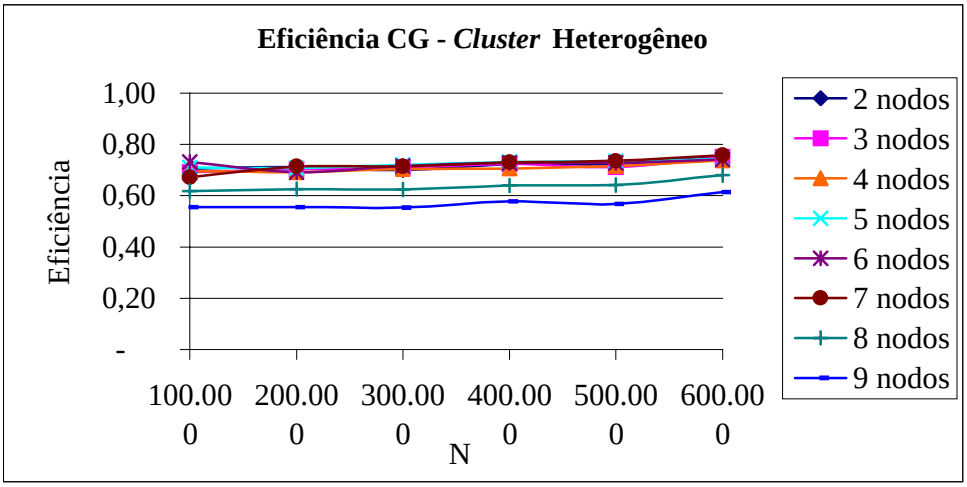


FIGURA 6.14 – Eficiência do CG no *Cluster* Heterogêneo
Ordem do Sistema Linear 100.000 a 600.000

Outro aspecto a ser observado é o número de nodos. Quanto maior esse número, mais comunicação é envolvida no algoritmo, além de, no *cluster* heterogêneo, máquinas mais lentas estarem envolvidas no processamento, levando a ter ganhos menores de desempenho.

A paralelização do Gradiente Conjugado, em termos gerais, é vantajosa. Para todos os casos houve ganho, mas alguns chegaram a resultados muito próximos do ideal: *speedup* igual ao número de nodos utilizados e eficiência próxima a *um*. Em algumas situações, o que ocorre é os recursos computacionais serem melhor explorados do que em outras.

Outro ponto observado é que o desempenho é mantido quando é aumentado o tamanho da entrada e a quantidade de nodos. Esses casos mostram que o algoritmo possui escalabilidade. Kumar [KUM94] define escalabilidade como a capacidade do sistema utilizar de forma eficiente os recursos crescentes, ou seja, aumentando o número de nodos e o tamanho da entrada, o sistema mantém o desempenho.

6.2.3 Alguns Resultados na *Myrinet*

Os resultados apresentados da paralelização dos métodos, até então, foram realizados na rede *Fast-Ethernet.* A *Myrinet* é uma rede de interconexão com alta velocidade de transmissão de dados, também disponível no *cluster*, e nessa seção, são apresentados alguns resultados obtidos pela execução dos métodos nessa rede.

O DECK possibilita a execução dos programas paralelos em ambas redes. Para o trabalho na *Fast-Ethernet*, o DECK proporciona ganho de desempenho, como visto no método do Gradiente Conjugado. Na *Myrinet*, o DECK-BIP (*Basic Interface for Parallelism*) ainda não atingiu o mesmo estágio de desenvolvimento do DECK para a *Fast-Ethernet.*

Devido a problemas de controle de fluxo, duas versões DECK-BIP foram usadas na realização das medidas de desempenho: DECK-BIP-1.3 e DECK-BIP-1.3.2. O protocolo de controle de fluxo no DECK é implementado com uma *thread* que é executada em cada nodo e é responsável por verificar a disponibilidade de *buffers* de recepção de grandes mensagens. No envio de mensagens pequenas, essa vai diretamente à *mail box* destino (BIP armazena a mensagem em seu *buffer* interno). Com grandes mensagens é necessário o envio para a *thread* de comunicação no nodo remoto pedindo para verificar a existência de um *buffer* de recepção para a mensagem e se não houver, o nodo que está enviando a mensagem tenta outra solicitação [BAR2000].

A versão DECK-BIP-1.3 possibilita desempenho melhor na *Myrinet* que na *Fast-Ethernet*. Porém a execução da aplicação fica completamente instável: para ser alcançado um tempo de execução menor que na *Fast-Ethernet* é necessário executar o código dezenas de vezes.

Na versão DECK-BIP-1.3.2 os procedimentos adotados para contornar o problema do controle de fluxo, como descrito anteriormente, consomem muita CPU. Isso faz com que as aplicações na *Myrinet* sejam executadas em um tempo maior que na *Fast-Ethernet*. Nesse contexto, alguns testes na *Myrinet*, com o método do Gradiente Conjugado, sem pré-condicionador, foram realizados com o DECK-BIP-1.3.

Buscou-se identificar os melhores casos de execução, apesar da instabilidade encontrada na rede. Os gráficos das figura 6.15 e 6.16 trazem respectivamente a comparação do *speedup* e da eficiência para as duas redes, com sistemas lineares de ordem 10.000 e 20.000.

Como pode ser observado a *Myrinet* proporciona a comunicação entre os nodos com uma velocidade mais elevada que a *Fast-Ethernet.* Na execução paralela com 3 e 4 nodos alcançou-se *speedup* superlinear, ou seja, maior que o *speedup* ideal, que é igual ao número de nodos usados. A utilização de uma rede de maior velocidade, evidencia a alta velocidade da memória *cache*, o que é explícito no *speedup* superlinear.

Com um número maior de nodos na execução, o tamanho do problema a ser resolvido em cada nodo é menor e, conseqüentemente, nesse exemplo, (quase) todos os dados manipulados pelo algoritmo do Gradiente Conjugado são suportados na memória

cache. Por exemplo, com um *n* = 10.000, tendo-se presente as estruturas de dados envolvidas no algoritmo do Gradiente Conjugado, e com um número de nodos acima de 3, todos os dados para execução podem ser armazenados na memória *cache* (256 Kb). Assim, nesse exemplo, com a diminuição do custo de comunicação através da *Myrinet*, a velocidade da memória *cache* é fator primário na obtenção do *speedup* superlinear.

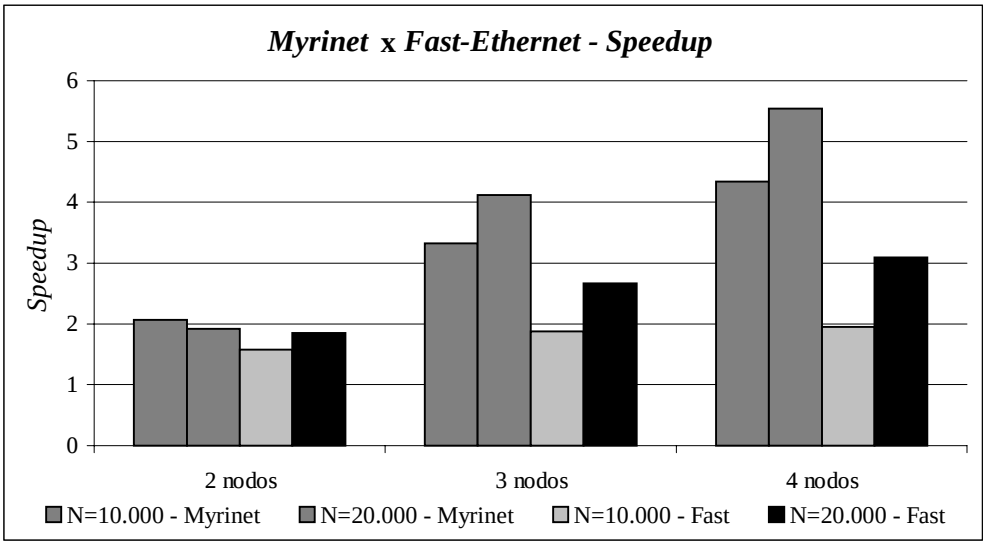


FIGURA 6.15 – *Myrinet x Fast-Ethernet – Speedup* do Gradiente Conjugado

No gráfico da Eficiência, na *Myrinet* com dois nodos, a eficiência é ideal, ou seja, igual a *um*. Com um número maior de nodos, a eficiência ultrapassa o ideal, conforme os resultados obtidos no *speedup*.

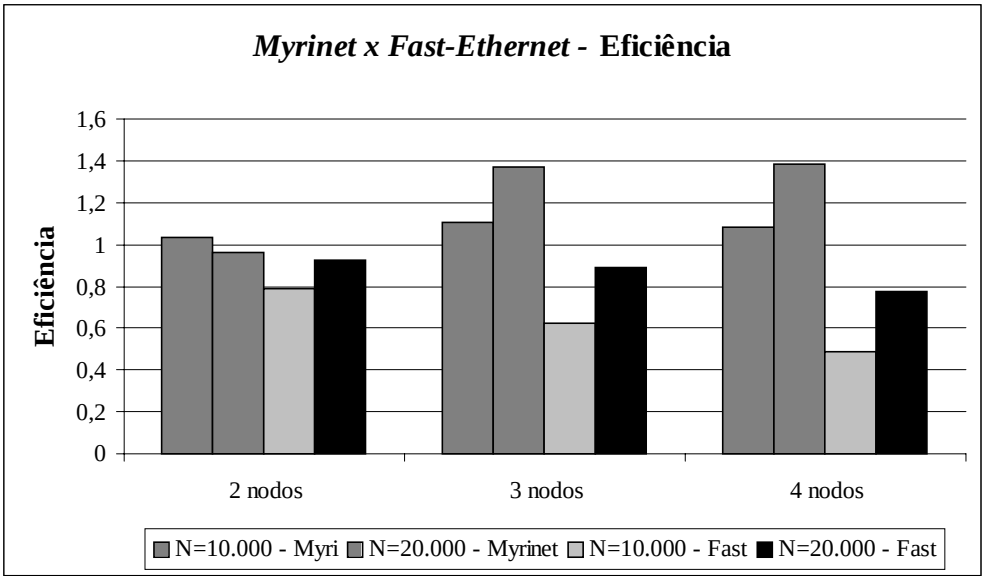


FIGURA 6.16 – *Myrinet x Fast-Ethernet* – Eficiência do Gradiente Conjugado

6.2.4 *Threads*

No DECK as primitivas relacionadas com multiprogramação e sincronização são baseadas no pacote *Pthreads* que disponibiliza recursos para implementar as *threads* e os semáforos do DECK. Em uma versão do método do Gradiente Conjugado paralelo

foram utilizadas *threads.* Nos resultados anteriores, o paralelismo é somente com trocas de mensagens.

Com a inclusão de *threads*, o algoritmo do Gradiente Conjugado necessitou algumas sincronizações para que a iteração do método fosse executada consistentemente. A utilização de *threads* nesse método de resolução foi descrita no capítulo 5, seção 5.2.5.

Alguns questionamentos quanto à utilização de *threads* surgiram logo no início, como por exemplo, qual é o número ideal de *threads* a serem usadas e realmente nos nodos Dual Pentium, os dois processadores são utilizados? Para buscar respostas às questões e melhor avaliar o desempenho de algoritmos com *threads* do DECK, optou-se por utilizar um exemplo simples, em que não houvessem trocas de mensagens, como a soma paralela de vetores.

Nesse exemplo é utilizado um único nodo Dual Pentium e realizadas variações na ordem do vetor (tamanho da entrada) e no número de *threads* utilizadas. Foram utilizadas uma, duas, quatro e oito *threads* em vetores com ordens 5.000, 30.000, 100.000 e 1.000.000.

Como pode ser observado na tabela 6.1, que traz o tempo de execução para as várias *threads* utilizadas, quanto maior o número de *threads*, maior é esse tempo. A primeira pergunta, então, está respondida: quanto mais *threads*, maior a concorrência para a execução. Para essa aplicação, onde deseja-se usufruir da capacidade dos dois processadores disponíveis em alguns nodos, sendo utilizado somente o paralelismo de dados e não o de instruções, o ideal é ter tantas *threads* quantos são o número de processadores, como mostram os dados da tabela abaixo.

TABELA 6.1 – Tempo de Execução - *Threads*

DECK *Threads*				
Ordem do Sistema	**1 *Thread***	**2 *Threads***	**4 *Threads***	**8 *Threads***
5.000	1953 µs	2142 µs	2864 µs	3401 µs
30.000	7724 µs	6862 µs	7063 µs	8741 µs
100.000	25540 µs	19892 µs	20560 µs	21174 µs
1.000.000	273826 µs	188978 µs	192158 µs	197635 µs

O fato do tempo de execução ter sido reduzido quando utilizadas duas *threads*, em relação à utilização de somente uma *thread,* indica que os dois processadores do nodo Dual Pentium estão sendo usados. Idealmente, executando uma aplicação em tais condições, o tempo de execução deveria ser reduzido pelo número de processadores alocados que compartilham a memória. Porém, na prática isso nem sempre acontece.

Para o exemplo da soma de vetores, os gráficos das figuras 6.17 e 6.18 mostram o *speedup* e a eficiência. O *speedup* ideal é igual a *dois* (número de processadores utilizados), porém o maior obtido foi 1,44 para uma entrada de 1.000.000. A eficiência ideal é igual a *um* (equivale a 100% da utilização dos recursos disponíveis) e a maior eficiência obtida foi 0,72.

Observa-se que na soma de vetores, onde não há trocas de mensagens, o uso de *threads* proporcionou ganho de desempenho menor que nos algoritmos com troca de mensagens. Foram realizados os mesmos testes da soma de vetores com a biblioteca *Pthreads* e os resultados foram semelhantes, confirmando que não há problemas no DECK.

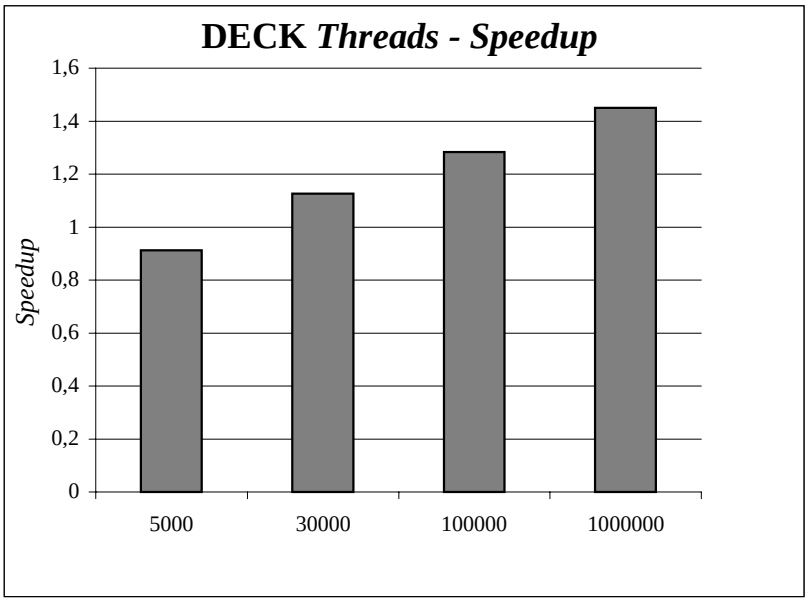


FIGURA 6.17 – Soma de Vetores com *Threads* – *Speedup*

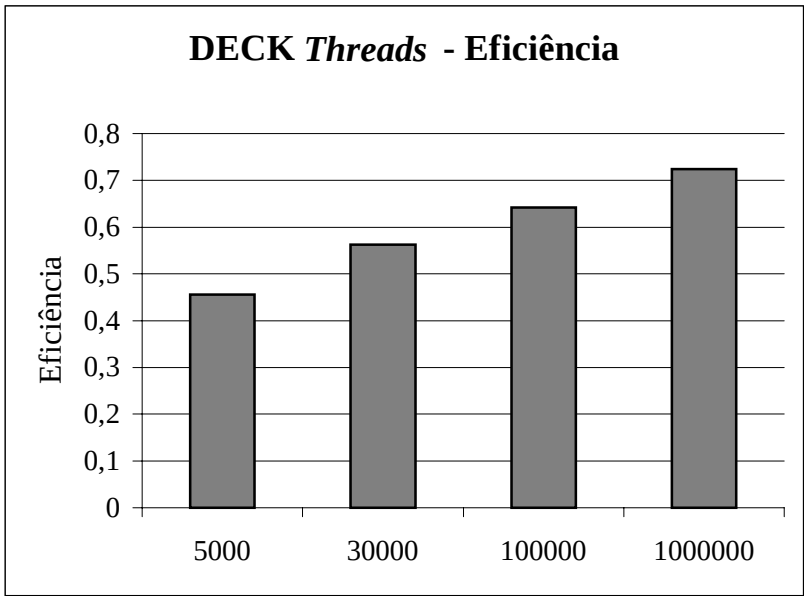


FIGURA 6.18 – Soma de Vetores com *Threads* - Eficiência

Alguns fatores contribuem para o pouco ganho de desempenho. Com entradas menores no algoritmo, pode-se ter pouca carga de dados nas *threads* não proporcionando desempenho. No método do Gradiente Conjugado há um fator a mais que proporciona a diminuição do desempenho: as sincronizações. Outro aspecto é a existência de *threads* que não são da aplicação concorrendo aos recursos.

Se for comparado à utilização de dois nodos com trocas de mensagens, com uma entrada a partir de 30.000 para o Gradiente Conjugado, o desempenho é muito próximo do desempenho ideal, o que não ocorre no caso das *threads.* Na figura 6.18, o *speedup* do Gradiente Conjugado com *threads* executado em um único nodo Dual Pentium mostra que em nenhum caso o tempo de execução com duas *threads* foi menor que o tempo de execução com uma única *thread.*

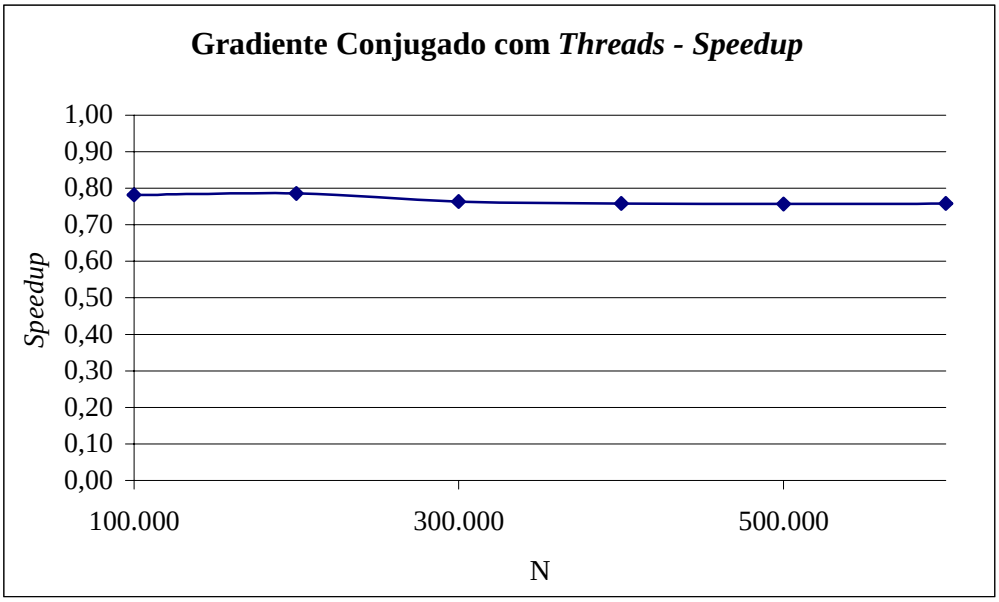


FIGURA 6.19 – CG com *Threads* - *Speedup*

Como reflexo do que mostra o *speedup*, a eficiência máxima alcançada foi 0,4 (figura 6.19), enquanto que a eficiência ideal é igual a *um*. O pouco desempenho que as *threads* proporcionam à execução paralela, como mostrado com a soma paralela de vetores, aliado às sincronizações de *threads* necessárias ao Gradiente Conjugado levam a esses resultados.

Combinar a programação paralela com memória distribuída (trocas de mensagens) e memória compartilhada (*threads*) é o desafio. Com a tendência de utilização de *multiclusters* esse tipo de programação torna-se evidenciado.

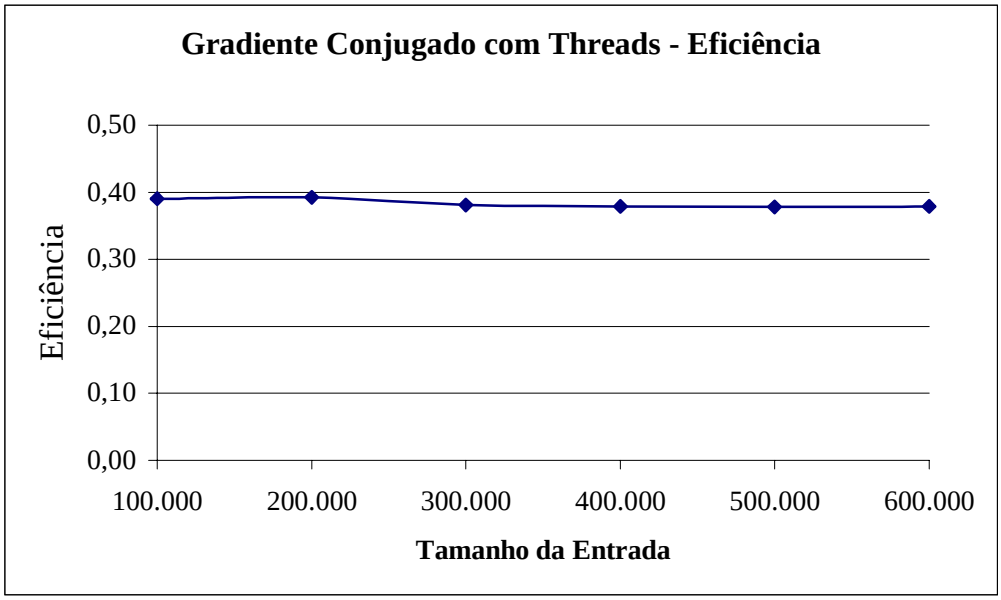


FIGURA 6.20 – CG com *Threads* - Eficiência

6.2.5 Pré-Condicionadores

No capítulo 3, seção 3.4.2, o pré-condicionamento do Gradiente Conjugado foi descrito e no capítulo 5, seção 5.2.6, a paralelização do método do Gradiente Conjugado Pré-Condicionado. O pré-condicionamento é uma técnica que transforma o sistema linear a ser solucionado em um outro sistema linear com propriedades mais favoráveis à convergência.

Os pré-condicionadores escolhidos para o método do Gradiente Conjugado Pré-Condicionado foram o Diagonal e o Polinomial. O pré-condicionador Diagonal equivale ao pré-condicionador Polinomial quando o grau do polinômio é igual a *zero*. Para o pré-condicionador Polinomial, foram experimentados polinômios de grau 1, 2, 3 e 4.

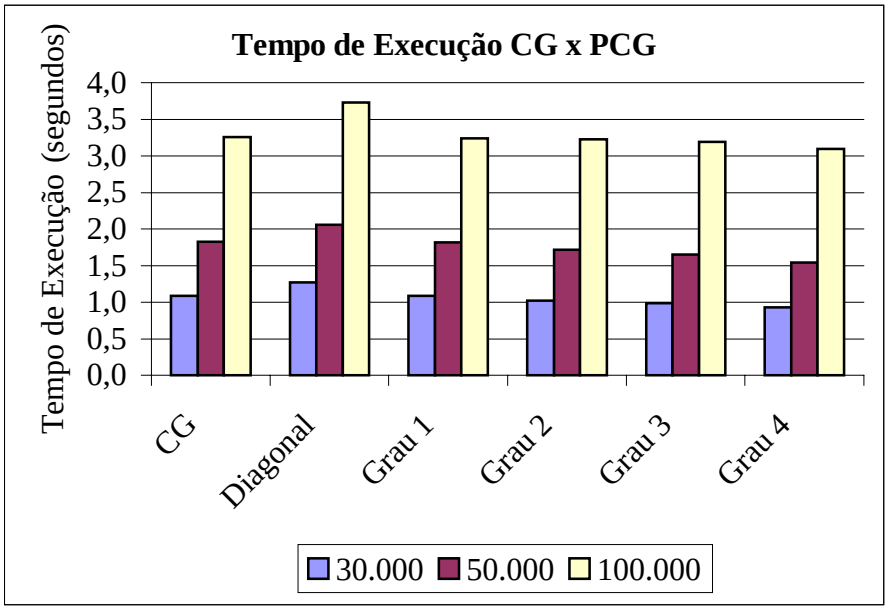


FIGURA 6.21 – Tempo de Execução CG x PCG

O gráfico da figura 6.21 mostra o tempo de execução do método do CG - Gradiente Conjugado e da versão com pré-condicionador PCG – Gradiente Conjugado Pré-Condicionado, para sistemas lineares de ordem 30.000, 50.000 e 100.000. O pré-condicionador Diagonal, para o tipo de matriz desse estudo (tridiagonal e spd), aumentou o tempo de execução, pois não houve redução do número de iterações do método, e sim um acréscimo no número de operações do algoritmo.

Com o pré-condicionador Polinomial de grau *um*, as 13 iterações necessárias à convergência dos sistemas lineares foram reduzidas a 8 iterações. Apesar do algoritmo ter sido acrescentado de multiplicação matriz tridiagonal por vetor, o tempo de execução foi menor que do Gradiente Conjugado sem pré-condicionamento.

O polinômio de grau *dois* inclui no algoritmo multiplicações matriz penta-diagonal por vetor, o que implica em maior quantidade de processamento e comunicação. Por outro lado, com esse pré-condicionador o número de iterações para resolução de tais sistemas diminuiu para 7 iterações, proporcionando uma redução no tempo de execução do Gradiente Conjugado Pré-Condicionado.

As situações anteriores ser repetem com os pré-condicionadores Polinomiais de graus 3 e 4. O de grau 3 insere multiplicações de matriz com sete diagonais por vetor e o de grau 4, multiplicações de matriz com nove diagonais por vetor. O número de iterações para esses casos, ficou 6 iterações e 5 iterações respectivamente.

Na figura 6.22 pode-se observar a comparação da paralelização do Gradiente Conjugado com e sem pré-condicionamento, para um sistema linear de ordem 50.000, no *cluster* homogêneo (até quatro nodos Dual Pentium). Há desempenho em todos os casos, mesmo aumentando a comunicação conforme é aumentado o grau do polinômio, pois o número de iterações é reduzido. Assim, conclui-se que o aumento da complexidade do algoritmo provocado pelo aumento de diagonais na matriz pré-condicionadora é compensado pela diminuição do número de iterações necessárias à convergência do sistema linear.

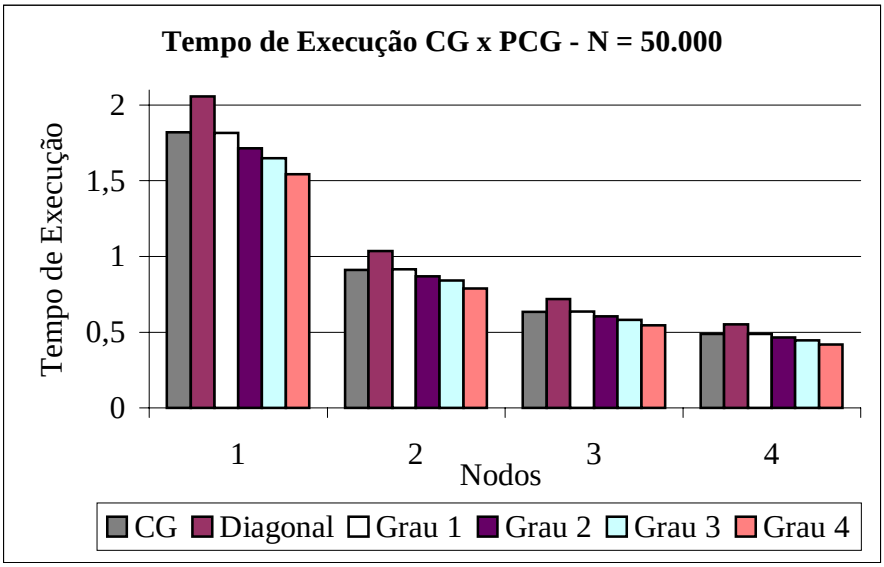


FIGURA 6.22 – Tempo de Execução CG x PCG – N = 50.000

6.3 Considerações Finais

Nesse capítulo foi realizada a avaliação da paralelização dos métodos de resolução e apresentados os principais resultados obtidos. Na avaliação de desempenho de algoritmos paralelos, a complexidade de algoritmos não depende somente da complexidade das operações, mas também da complexidade das operações que geram *overhead* no sistema, especialmente a comunicação.

Na avaliação empírica, os resultados mostraram a influência da heterogeneidade das máquinas no desempenho da aplicação. Com a utilização de *threads* no Gradiente Conjugado, não houve ganho de desempenho devido às sincronizações necessárias ao algoritmo. Através do exemplo da soma paralela de vetores, pode-se perceber que houve aproveitamento dos dois processadores nos nodos Dual Pentium, porque o tempo de execução diminuiu, mas mesmo assim, outros fatores influenciam não permitindo o melhor desempenho possível.

Cabe ressaltar que na implementação do DECK *Sockets* é utilizada uma *thread* para realizar a comunicação com o Servidor de Nomes. Essa *thread* é ativada somente quando há um *bind* ou um *fetch*, significando que o restante do tempo não ocupa CPU. No DECK-BIP, há uma *thread* na implementação responsável pelo controle de fluxo, como comentado no capítulo 4. Isso significa que *threads* da biblioteca DECK não estão concorrendo recursos com *threads* da aplicação.

Outro ponto a ser observado na avaliação do desempenho da aplicação multiprogramada é a *cache*. Desabilitar a *cache* para executar os testes, particionar os

dados de forma não contínua (por exemplo, par-ímpar) e ao mesmo tempo aumentar a carga de processamento são alternativas para verificar se realmente a memória *cache* tem influência no uso de *threads.*

Apesar do método do Algoritmo de Thomas não ser propício à paralelização, observando-se as figuras 6.1 e 6.6, o tempo de execução do Algoritmo de Thomas é menor que o do Gradiente Conjugado para um mesmo problema. Isso acontece pelo fato do Algoritmo de Thomas ser um algoritmo muito eficiente na resolução de sistemas lineares tridiagonais, o que é explicitamente demonstrado pela sua complexidade seqüencial.

As versões do método do Gradiente Conjugado mostraram resultados muito bons. Isso indica que o conjunto *cluster*, DECK e algoritmo paralelo convergiram para o mesmo ponto: ganho de desempenho. Nos casos onde houveram problemas com o algoritmo ou com o DECK, é visto que melhorias devem ser feitas, para o melhor aproveitamento dos recursos do ambiente paralelo.

7 Conclusões

O objetivo principal dessa dissertação é a paralelização e avaliação do desempenho de alguns métodos de resolução de sistemas lineares esparsos, utilizando o DECK, em um *clusters* de PCs. Várias etapas precederam a obtenção desse objetivo, como o estudo sobre sistemas de equações lineares esparsos, onde verificou-se a aplicabilidade e características desses sistemas e formas de armazenamento da matriz dos coeficientes.

Para resolução dos sistemas lineares foram investigados métodos que melhor se adequassem ao problema, como a esparsidade e a rápida solução dos sistemas, nas duas classes de métodos: diretos e iterativos. Ao término dessa etapa foram identificados os dois métodos de resolução dessa pesquisa e, para o método iterativo, os pré-condicionadores usados.

Para ser possível a paralelização, o ambiente paralelo onde foram implementados os métodos foi estudado. Esse estudo envolveu a identificação das características físicas do *cluster* e o estudo da ferramenta utilizada para programação, o DECK.

Definido o ambiente paralelo, o que engloba o paradigma de programação, foram investigadas formas de paralelizar os algoritmos dos métodos de cada classe. No método iterativo, também buscou-se utilizar multiprogramação, sendo estudada a aplicação de *threads*.

A etapa de implementação dos algoritmos, proporcionou a validação da ferramenta DECK no *cluster*. Conseguiu-se contribuir para o aprimoramento dessa ferramenta, em termos de funcionamento e de desempenho. Com relação ao *cluster*, a avaliação do desempenho realizada mostrou bons resultados na *Fast-Ethernet e* resultados promissores na utilização da *Myrinet*. Problemas foram enfrentados, mas como está se fazendo investigações e validações, isso era o esperado e proporcionou a criação de novas versões do DECK e melhorias no funcionamento do *cluster*.

A avaliação da paralelização foi realizada através da complexidade dos algoritmos paralelos e de medidas como *speedup* e eficiência, calculadas a partir do tempo de execução paralelo. Essa avaliação possibilitou a verificação do comportamento dos algoritmos paralelizados e do desempenho do DECK e do *cluster*. Os algoritmos paralelos devem ser avaliados sob parâmetros adicionais ao tempo de execução, como o número de processadores, o esquema de comunicação e sincronizações.

Ao se chegar nessa etapa do trabalho, que não é o término, pois muitos pontos relevantes precisam ser tratados, os principais resultados alcançados são: o Algoritmo de Thomas distribuído, a paralelização do método do Gradiente Conjugado, com pré-condicionadores (diagonal e polinomial com graus 1, 2, 3 e 4) e sem pré-condicionadores, tendo essa última uma versão com *threads* e a avaliação desses algoritmos.

Relacionado à paralelização, o Algoritmo de Thomas distribuído não apresentou desempenho, especialmente pelas dependências de dados no método, o que levou a inexistência de paralelismo. Se houverem vários sistemas lineares independentes para serem resolvidos, pode-se criar um *pipeline* de execução dos sistemas, sendo cada um deles resolvido conforme o Algoritmo de Thomas distribuído, aqui descrito.

O método do Gradiente Conjugado tem a paralelização muito vantajosa. A complexidade desse algoritmo paralelo já apontava para um bom desempenho. A versão com *threads* do método sofreu forte influência do *overhead* das sincronizações

necessárias entre as *threads* para a perfeita execução da iteração do Gradiente Conjugado.

A obtenção de desempenho pela utilização de *threads* é influenciada por vários fatores. Um deles é o número de *threads* utilizadas que dependerá do tamanho da entrada dos dados, da quantidade de processadores que compartilham a memória e qual tipo de paralelismo que está sendo usado. Outro fator identificado é a formulação do algoritmo: síncrono ou assíncrono e se esse apresenta dependências de dados.

No Gradiente Conjugado paralelo, quanto maior o número de nodos, menor o tempo de execução. Porém, deve-se observar que quanto menor esse tempo, nem sempre implica que os recursos computacionais estão sendo usados da melhor forma. Para isso devem ser analisados o *speedup* (para verificar o ganho da aplicação em relação à execução seqüencial, que para o menor tempo será sempre o maior *speedup*) e a eficiência do algoritmo (que mostra o ganho conforme o número de nodos utilizados).

Para o método do Gradiente Conjugado, com até quatro nodos, a partir de sistemas lineares com ordem 30.000, o desempenho obtido é muito bom (próximo ao ideal). A partir de 100.000 tanto no *cluster* homogêneo ou heterogêneo o desempenho é mantido, o que demonstra escalabilidade: aumentando a entrada e o número de nodos alocados, o desempenho é mantido.

Com relação aos pré-condicionadores, o pré-condicionador Polinomial mostrou ser eficiente. Conforme o grau do polinômio, é definida a estrutura da matriz pré-condicionadora, que influencia na multiplicação matriz por vetor e na quantidade de comunicação. Também conforme esse grau, pode-se favorecer a convergência do sistema linear. Observou-se que quanto maior o grau, mais rapidamente o sistema linear converge e em conseqüência, menor é o tempo de execução.

No *cluster* heterogêneo, pelas diferentes configurações das máquinas e o algoritmo ser síncrono, o desempenho mostrou ser um pouco menor que no *cluster* homogêneo. Acontece que nas várias sincronizações dos nodos, a máquina mais lenta faz com que todas as outras esperem por ela, já que a distribuição dos dados é homogênea. Um ponto a ser tratado é esse: realizar a distribuição dos dados conforme a capacidade de processamento de cada nodo.

Vários fatores podem introduzir ineficiência ou diminuir o ganho de desempenho nas aplicações paralelas. Dentre esses está a insuficiência de paralelismo na computação, como no caso do Algoritmo de Thomas; *overhead* decorrente de sincronizações, como no Gradiente Conjugado com *threads* e tempos de espera introduzidos, por exemplo, pela utilização de nodos heterogêneos.

7.1 Trabalhos Futuros

A área de aplicações de alto desempenho é vasta e estimula o desenvolvimento de soluções computacionais mais rápidas. A atual tendência de utilização de *clusters* é um exemplo da busca dessas soluções através de um custo menor tanto do *hardware* quanto do *software*.

Como trabalhos futuros, em um primeiro momento, tem-se o estudo detalhado sobre pré-condicionadores, especificamente o pré-condicionador polinomial, aumentando o grau do polinômio e aplicando-o a outros tipos de matrizes; a distribuição dos dados dos algoritmos conforme a capacidade de processamento de cada nodo, para verificar o desempenho, no *cluster* heterogêneo; a execução e avaliação dos algoritmos

em um *cluster* com um número maior de nodos; investigação de formas de utilização eficiente de *threads*, possibilitando a combinação de programação com memória distribuída e compartilhada; inclusão das rotinas de comunicação de grupo do DECK nos algoritmos; paralelização de outros métodos de resolução, especialmente os iterativos por serem mais apropriados.

O desenvolvimento de novas aplicações em DECK proporcionará o maior aprimoramento dessa ferramenta. A paralelização de outros métodos de resolução de sistemas lineares e até mesmo a elaboração de uma biblioteca de métodos em DECK são aplicações que podem ser desenvolvidas.

Com relação ao método do Gradiente Conjugado, outros pré-condicionadores devem ser investigados. Outro aspecto a ser dedicada atenção é a combinação da programação para memória compartilhada e memória distribuída. No caso dos métodos de resolução, são poucas bibliotecas que permitem mesclar esses dois paradigmas.

Comparações de aplicações em DECK com aplicações em MPI e/ou PVM possibilitarão um bom parâmetro para verificar o desempenho do DECK. Isso pode ser feito com outras bibliotecas de resolução desenvolvidas com essas ferramentas ou utilizando outras aplicações. Outro ponto que pode ser explorado, a partir dessa pesquisa, é a exatidão dos resultados na utilização do *cluster*.

Percebe-se perspectivas para o desenvolvimento de aplicações de alto desempenho, procurando utilizar memória comparilhada (*threads*) e distribuída (trocas de mensagens). O DECK proporciona essa combinação e tem demonstrado bom desempenho, salvo partes que estejam relacionadas ao ambiente em desenvolvimento.

Anexo 1

Algoritmos Seqüenciais dos Métodos de Resolução

Algoritmo do Método de Thomas

Algoritmo_de_Thomas(*a, b, c, d, x*)
//*a, b* e *c* são vetores com as 3 diagonais da matriz, de ordem *n*
//*d* é o vetor dos termos independentes do sistema
//*x* é o vetor das incógnitas

//Primeira Etapa
$c'_1 = c_1 / b_1$
$d'_1 = d_1 / b_1$
Para i = 2 **até** *n* **faça**
    Se i<>*n* **então** $c'_i = c_i / (b_i - a_i * c'_{i-1})$
    $d'_i = (d_i - a_i * d'_{i-1}) / (b_i - a_i * c'_{i-1})$
Fim_Para

//Segunda Etapa: Retrossubstituição
$x_n = d'_n$
Para i = *n*-1 **decrescendo até** 1 **faça**
    $x_i = d'_i - x_{i+1} * c'_i$
Fim_Para
Saída(*x***)**

Algoritmo do Gradiente Conjugado

$i = 0$
$r = b - Ax$
$d = r$
$\delta_{novo} = rr = \|r\|_2^2$
$\delta_0 = \delta_{novo}$
Enquanto $i < i_{\max}$ e $\delta_{novo} > \varepsilon^2 \delta_0$ faça
    $q = Ad$
    $\alpha = \dfrac{\delta_{novo}}{dq}$
    $x = x + \alpha q$
    Se i é divisível por 50
        $r = b - Ax$
    senão
        $r = r - \alpha q$
    $\delta_{velho} = \delta_{novo}$
    $\delta_{novo} = rr$
    $\beta = \dfrac{\delta_{novo}}{\delta_{velho}}$
    $d = r + \beta d$
    $i = i + 1$

Algoritmo do Gradiente Conjugado Pré-condicionado

$i = 0$
$r = b - Ax$
$d = C^{-1}r$
$\delta_{novo} = rd$
$\delta_0 = \delta_{novo}$
Enquanto $i < i_{\max}$ e $\delta_{novo} > \varepsilon^2 \delta_0$ faça
    $q = Ad$
    $\alpha = \dfrac{\delta_{novo}}{dq}$
    $x = x + \alpha q$
    Se i é divisível por 50
        $r = b - Ax$
    senão
        $r = r - \alpha q$
    $s = C^{-1}r$
    $\delta_{velho} = \delta_{novo}$
    $\delta_{novo} = rs$
    $\beta = \dfrac{\delta_{novo}}{\delta_{velho}}$
    $d = r + \beta d$
    $i = i + 1$

Anexo 2

Tabelas

TABELA A2.1– Tempo de Execução do Algoritmo de Thomas

Algoritmo de Thomas (Tempo de Execução em Segundos)									
N	**1 nodos**	**2 nodos**	**3 nodos**	**4 nodos**	**5 nodos**	**6 nodos**	**7 nodos**	**8 nodos**	**9 nodos**
50.000	0,057523	0,057954	0,060027	0,060509	0,060520	0,061975	0,062122	0,062652	0,065452
100.000	0,116243	0,119380	0,120170	0,120173	0,120439	0,120573	0,120864	0,121463	0,122031
300.000	0,350191	0,357507	0,357855	0,358026	0,358297	0,358478	0,358686	0,358741	0,358946
600.000	0,710784	0,711501	0,711692	0,711742	0,712327	0,712449	0,712567	0,712646	0,712797

TABELA A2.2 – Tempo de Execução do Gradiente Conjugado

Gradiente Conjugado (Tempo de Execução em Segundos)										
N	**Scliar**	**Nodo Dual**	**2 nodos**	**3 nodos**	**4 nodos**	**5 nodos**	**6 nodos**	**7 nodos**	**8 nodos**	**9 nodos**
50.000	1,573416	2,172489	1,088596	0,757891	0,581698	0,435781	0,369005	0,326198	0,313986	0,303058
100.000	3,012850	4,030363	2,136471	1,451396	1,082679	0,848092	0,686643	0,641081	0,608570	0,602470
300.000	9,020496	12,262167	6,439742	4,234997	3,199751	2,506082	2,101356	1,804099	1,806548	1,807598
600.000	17,420703	22,738685	11,569580	7,724595	5,906200	4,682963	3,915175	3,280702	3,199609	3,155195

TABELA A2.3- Myrinet x Fast-Ethernet
Speedup e Eficiência do Gradiente Conjugado

Gradiente Conjugado – *Myrinet x Fast-Ethernet*							
		Speedup			**Eficiência**		
	N	**2 nodos**	**3 nodos**	**4 nodos**	**2 nodos**	**3 nodos**	**4 nodos**
Myrinet	**10000**	2,061432	3,323857	4,339431	1,030716	1,107952	1,084858
	20000	1,920301	4,121478	5,536378	0,96015	1,373826	1,384094
Fast-Ethernet	**10000**	1,576434	1,875905	1,952605	0,788217	0,625302	0,488151
	20000	1,85024	2,660716	3,094534	0,92512	0,886905	0,773633

TABELA A2.4– Tempo de Execução Gradiente Conjugado x Gradiente Conjugado Pré-Condicionado

Gradiente Conjugado Pré-Condicionado (Tempo de Execução em Segundos)						
N	**CG**	**Diagonal**	**Grau 1**	**Grau 2**	**Grau 3**	**Grau 4**
30000	1,090289	1,27296	1,087068	1,022273	0,984421	0,928522
50000	1,821871	2,05905	1,816489	1,717639	1,649315	1,541728
100000	3,252482	3,727387	3,242874	3,225623	3,19124	3,092658

TABELA A2.5– Tempo de Execução Gradiente Conjugado (N = 50.000)

Gradiente Conjugado Pré-Condicionado – N=50.000 (Tempo de Execução em Segundos)						
Nodos	**CG**	**Diagonal**	**Grau 1**	**Grau 2**	**Grau 3**	**Grau 4**
1	1,821871	2,05905	1,816489	1,717639	1,649315	1,541728
2	0,910936	1,034698	0,91741869	0,871898	0,8414872	0,79062974
3	0,634798	0,719948	0,63736456	0,6048025	0,5827968	0,54671206
4	0,488437	0,553508	0,48961968	0,4642268	0,4469688	0,41894783

Bibliografia

[AKL89] AKL, Selim G. **The Design and Analysis of Parallel Algorithms.** Englewood Cliffs: Pretince Hall, 1989.

[AND95] ANDERSON, John D. **Computational Fluid Dynamics:** The Basics with Applications. New York: McGraw-Hill-Inc, 1995.

[ARA97] ARAÚJO, Ézio R. **Métodos Iterativos em Álgebra Linear Computacional.** Rio de Janeiro: LNCC/CNPQ, 1997.

[BAK2000] BAKER, M. **Cluster Computing White Paper.** Disponível em: <http://www.dcs.port.ac.uk/~mab/tfecc>. Acesso em: fev. 2000.

[BAR2000] BARRETO, M. E. **DECK:** Um ambiente para programação paralela em agregados de multiprocessadores. Porto Alegre: PPGC – UFRGS, 2000. Dissertação de Mestrado.

[BAR98] BARRETO, M.E.; NAVAUX, P.O.A.; RIVIÉRE, M. Deck: A new model for a distributed kernel integrating communication and multithreading for support of distributed object-orientede application with fault tolerance. In: CONGRESSO ARGENTINO DE CIENCIAS DE LA COMPUTATION, CACIC, 4., 1998, Neuquén, Argentina. **Trabajos selecionados**... Neuquén: Universidad Nacional del Conalive, 1998.

[BRO2000] BROWNE, Shirley et al. **Numerical Libraries and Tools for Scalable Parallel Cluster Computing.** Cluster Computing White Paper. Disponível em: <http://www.dcs.port.ac.uk/~mab/tfecc>. Acesso em: fev. 2000.

[BUT97] BUTENHOF, David R. **Programming with POSIX Threads.** Amsterdam: Addison-Wesley, 1997.

[BUY99] BUYYA, R.; SILVA, L. M. Parallel Programming Models and Paradigms. In: **High Performance Cluster Computing**: programming and applications. Melbourne: Pretince Hall, 1999. v.2, p.4-27.

[CAN99] CANAL, Ana Paula. **Métodos de Resolução de Sistemas Lineares e suas Aplicações em Computação de Alto Desempenho**: trabalho individual. Porto Alegre: CPGCC-UFRGS, 1999.

[CLA94] CLÁUDIO, Dalcídio M.; MARINS, Jussara M. Cálculo Numérico Computacional: teoria e prática. 2. ed. São Paulo: Atlas, 1994.

[CRE99] CREMONESI, Paolo; ROSTI, Emilia et al. Performance Evaluation of Parallel Systems. **Parallel Computing**, Netherlands, v. 25, n.13/14, p 1677-1698, Dec. 1999.

[CUN2000] CUNHA, Rudinei da; HOPKINS, Tim. **Pim 2.0 the parallel iterative methods package for systems of linear equations user's guide.** Disponível em: <http://www.cs.ukc.ac.uk/pubs/1996/54/index.html>. Acesso em: fev. 2000.

[CUN92] CUNHA, Rudinei da. **A Study on Iterative Methods for the Solution of Systems of Linear Equations on Transputers Networks.** Canterbury: University of Kent, 1992. Tese de Doutorado.

[DIV90] DIVERIO, Tiarajú A. **LEPMAC Material Didático de Apoio – Módulo SELAS.** Porto Alegre: II-UFRGS, 1990.

[DON2000] DONGARRA, Jack. **Jack Dongarra Papers.** Disponível em: <http://www.netlib.org/utk/people/JackDongarra/papers>. Acesso em: fev. 2000.

[DON94] DONALDON, V; BERMAN, F; PATURI, R. Program Speedup in a Heterogeneous Computing Network. **Journal Parallel Distributed Computing,** San Diego, v. 21. p. 316-322, 1994.

[DOR2000] DORNELES, Ricardo Vargas et al. PC Cluster Implementation of a Mass Transport Two-Dimensional Model. In: SBAC – PAD, 2000. **Proceedings...** São Carlos: UFSCar, 2000. p. 191-198 p.

[DOR99] DORNELES, Ricardo V. Modelos Atmosféricos Regionais: trabalho individual. Porto Alegre: CPGCC-UFRGS, 1999.

[DUB79] DUBOIS, P.F.; GREENBAUM, A.A.; RODRIGUE, G.H. Aproximating the inverse of a matrix for use in iterative algorithms on vector processors. **Computing**, Livermore, n. 22, p. 257-268, 1979.

[DUF89] DUFF, I.S.; ERISMAN, A. M.; REID, J.K. **Direct Methods for Sparses Matrices.** Oxford: Clarendon Press, 1989.

[DUF99] DUFF, Iain S.; VORST, Henk A. van der. Developments and trends in the parallel solution of linear systems. **Parallel Computing,** Amsterdam, v. 25, n. 13-14, p.1931-1970, Dec. 1999.

[EIJ99] EIJKHOUT, Victor. **Overview of Iterative Linear System Solver.** Disponível em: <http://www.netlib.org/utk/papers/iterative-survey>. Acesso em: nov. 1999.

[FLE88] FLETCHER, Clive A. J. **Computational Techiques for Fluid Dynamics:** fundamental and general tecniques. Berlin: Springer-Verlag, 1988. v.1.

[FLY72] FLYNN, Michael J. Some Computer Organizations and their effectiveness. **IEEE Transactions on Computers**, New York, p. 948-960, Sept.1972.

[FOS95] FOSTER, Ian T. **Designing and Building Parallel Programs Concepts and Tool for Parallel Software Engineering.** Reading: Addison-Wesley, 1995.

[GPP2000] GPPD – Group of Parallel and Distributed Processing. **MultiCluster – Support for parallel programming on Multiple Clusters**. Disponível em: <http://www-gppd.inf.ufrgs.br/projects/mclusters/index.html>. Acesso em: maio. 2000.

[HIR88] HIRSCH, Charles. **Numerical Computation of Internal and External Flows**: fundamentals of numerical discretization. Great Britain: John Wiley & Sons, 1988.

[HWA98] HWANG, K; XU, X. **Scalable Parallel Computing:** Technology, Architecture, Programming. New York: McGraw-Hill, 1998.

[IML2000] IML. **IML++.** Disponível em: <http://www.mat.mist.gov/iml++/>. Acesso em: jan. 2000.

[IOR91] IÓRIO, Valéria. **EDP:** um Curso de Graduação. Rio de Janeiro: Instituto de Matemática Pura e Aplicada – CNPq, 1991.

[JAJ92] JAJA, Joseph. **An Introduction to Parallel Algorithms.** Reading: Addison-Wesley, 1992.

[JOU2000] JOUBERT, W.D. **Parallel Distributed Iterative Solver Project PCG.** Disponível em: <http://www.cfdlab.ae.utexas.edu/pcg/index.html>. Acesso em: jan. 2000.

[KEL95] KELLEY, C.T. **Iterative Methods for Linear and NonLinear Equations.** Philadelphia: Siam, 1995.

[KRO86] KRONSJO, Lydia. **Computational Complexity of Sequential and Parallel Algorithms.** New York: John Wiley & Sons, 1986.

[KUM94] KUMAR, Vipin et al. **Introduction to Parallel Computing:** design and analysis of algorithms. California: The Benjamin/Cummings, 1994.

[LAP2000] LAPACK. **LAPACK – Linear Algebra Package.** Disponível em: <http://www.netlib.org/lapack>. Acesso em: jan. 2000.

[LIN2000] LINPACK. **LINPACK – Linear Package.** Disponível em: <http://www.netlib.org/linpack>. Acesso em: jan. 2000.

[MOL93] MOLDOVAN, D. I. **Parallel Processing – From Application to Systems.** San Mateo: Morgan Kaufmann, 1993.

[MPI2000] MPI FORUM. **MPI – Message Passing Interface Forum.** Disponível em: <http://mpi-forum.org/>. Acesso em: maio 2000.

[MYR2000] MYRICOM. **Myrinet Index Page.** Disponível em: <http://www.myri.com>. Acesso em: fev. 2000.

[PAC97] PACHECO, Peter S. **Parallel Programming with MPI.** San Francisco: Morgan Kaufmann, 1997.

[PFI98] PFISTER, Gregory F. **In Search of Clusters.** 2 nd. Upper Saddle River: Pretince Hall PTR, 1998.

[POV98a] POVITSKY, A. **Parallel Directionally Split Solver Based on Reformulation of Pipelined Thomas Algorithm.** Hampton: NASA - Institute for Computer Application in Science and Engineering, Langley Res. Center, 1998.

[POV98b] OVITSKY, A. **Parallelization of the Pipelined Thomas Algorithm.** Hampton: NASA - Institute for Computer Application in Science and Engineering, Langley Res. Center, 1998.

[PRA97] PRASAD, Shashi. **Multithreading Programming Techniques.** New York: McGraw-Hill, 1997.

[PVM2000] PVM. **PVM – Parallel Virtual Machine.** Disponível em: <http://www.epm.oml.gov/pvm/home.html>. Acesso em: maio 2000.

[QUI94] QUINN, M. J. **Parallel Computing – Theory and Practice.** New York: McGraw-Hill, 1994.

[RIG99] RIGONI, E.H et al. **Introdução à Programação em *Clusters* de Alto Desempenho.** Porto Alegre: PPGC-UFRGS, 1999.

[RIZ98] RIZZI, Rogério. **Simulação Numérica de Modelos Meteorológicos:** trabalho individual**.** Porto Alegre: CPGCC da UFRGS, 1998.

[RIZ99] RIZZI, Rogério L. DIVERIO, Tiarajú A. **Simulação Numérica de Modelos Meteorológicos:** modelos matemáticos e computação paralela: trabalho individual II. Porto Alegre: CPGCC-UFRGS, 1999.

[SAA2000] SAAD, Yousef; MALEVSKY, Andrei et al. **PSPARSLIB**: A Portable Library of Parallel Sparse Iterative Solvers. Disponível em: <http://www.cs.umn.edu/Research/arpa/p'sparslib/psp-abs.html>. Acesso em: fev. 2000.

[SAA96] SAAD, Yousef. **Iterative Methods for Sparse Linear Systems.** Boston: Pws, 1996.

[SAN2000] SANDIA NATIONAL LABORATORIES. **Inside Aztec.** Disponível em: <http://www.cs.sandia.gov/CRF/aztec1.html>. Acesso em: jan. 2000.

[SCA2000] SCALAPACK. **SCALAPACK Project.** Disponível em: <http://www.netlib.org/scalapack>. Acesso em: jan. 2000.

[SHE94] SHEWCHUK, Jonathan R. **Na Introduction to the Conjugate Gradient Method without the Agonizing Pain.** Pittsburgh: Carnegie Mellon University, 1994.

[SIL99] SILVA, Renato Simões. Utilização de *Clusters* de PCs na Resolução de Problemas de CFD. In: CNMAC, 22. 1999. **Proceedings...** Santos: CNMAC, 1999. Conferência Plenária.

[SKA2000] SKALICKY, Tomas. **LASPack Reference Manual.** Disponível em: <http://www.tu-dresden.de/miusm/skalicky/laspcak/laspack.html>. Acesso em: fev. 2000.

[SLA97] SLAVIERO, Vânia M. P. **O Método do Gradiente Conjugado com Produto Interno Geral.** Porto Alegre: DMPA – UFRGS, 1997. Dissertação de Mestrado.

[STE94] STERN, Júlio M. **Esparsidade, Estrutura, Estabilidade e Escalonamento em Álgebra Linear Computacional.** São Paulo: DCC-USP, 1994.

[STR99] STROHMAIER, E.; DONGARRA, J.J.; MEUER, H. W. The marketplace of high-performance computing. **Parallel Computing,** Amsterdam, v.25, n. 13-14, p. 1517-1594, Dec. 1999.

[TER90] TERADA, Routo. **Introdução a Complexidade de Algoritmos Paralelos.** São Paulo: IME/USP, 1990.

[TOP2000] TOP500. **TOP500 Supercomputer Sites.** Disponível em: <http://www.netlib.org/benchmark/performance.ps>. Acesso em: fev. 2000.

[VOL97] VOOLEBREGT, Edwin A. H. **Parallel Software Development Techiques for Shallow Water Models.** Delft: Technische Universiteit Delf, 1997. Ph.D. Thesis.

[WES68] WESTLAKE, Joan R. **A Handbook of Numerical Matrix Inversion and Solution of Linear Equations.** New York: John Wiley & Sons, 1968.