Universidade Federal da Paraíba
Centro de Ciências e Tecnologia
Coordenação de Pós-Graduação em Informática

Resolução de Sistemas de Equações Lineares de Grande Porte Utilizando Processamento Paralelo

Jean Gonzaga Souza de Oliveira

Campina Grande - PB
Janeiro de 1999

Jean Gonzaga Souza de Oliveira

Resolução de Sistemas de Equações Lineares de Grande Porte Utilizando Processamento Paralelo

Dissertação submetida à Coordenação de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal da Paraiba como requisito parcial para a obtenção do grau de Mestre em Ciência (M. Sc).

Área de Concentração: Ciência da Computação
Linha de pesquisa: Matemática Computacional
Orientador: Prof. João Marques de Carvalho, Ph. D
Co-orientador: Mauro Cavalcante Pequeno, Dr.

Campina Grande
Universidade Federal da Paraíba

Ficha Catalográfica

Oliveira, Jean Gonzaga Souza de

O48R

Resolução de Sistemas de Equações Lineares de Grande Porte Utilizando Processamento Paralelo - Campina Grande: CCT/COPIN da UFPB, Janeiro de 1999, 152 p.

Dissertação (mestrado) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, 1998.

Orientador: João Marques de Carvalho, Ph. D
Co-orientador: Mauro Cavalcante Pequeno, Dr.

1. Matemática Computacional
2. Sistemas de Equações Lineares
3. Processamento Paralelo
4. Eliminação de Gauss
5. Fatoração LU
6. Método iterativo de Gauss-Jacobi
7. Método iterativo dos Gradientes Conjugados
8. PVM

CDU – 519.6

RESOLUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES DE GRANDE PORTE UTILIZANDO PROCESSAMENTO PARALELO

JEAN GONZAGA SOUZA DE OLIVEIRA

DISSERTAÇÃO APROVADA EM 29.01.1999

PROF. JOÃO MAQUES DE CARVALHO, Ph.D
Orientador

PROF. MAURO CAVALCANTE PEQUENO, D.Sc
Co-Orientador

PROF. BRUNO CORREIA DA N. QUEIROZ, M.Sc
Examinador

PROF. FRANCISCO DE ASSIS FERREIRA TEJO, D.Sc
Examinador

CAMPINA GRANDE - PB

À minha querida mãe, *Maria Zuleide de Souza de Oliveira* (*In memorian*)

"Bem aventurado aquele que teme ao Senhor e anda nos teus caminhos. Pois comerás do trabalho das tuas mãos; feliz serás e te irá bem."

Salmo 128

Dedicatória

À Deus, fonte de toda sabedoria do mundo.

À minha família (meu pai: Sr. Manoel Gonzaga de Oliveira, meus irmãos: Janet Gonzaga, Gesman Gonzaga, Janeide Gonzaga, Renato Gonzaga, Luciana Gonzaga e ao meu sobrinho Yan Gonzaga).

À minha querida esposa: Valdecina Araújo Barbosa de Oliveira pelo carinho, amor, compreensão e paciência.

À minha avó Emilia, tia Lindalva, tia Raimunda (in memorian), tios, tias e primos.

Ao meu avô Luiz Gonzaga

Ao Sr. João Demétrio

Ao Prof. Mário Toyotaro Hattori (In Memorian), pelo apoio, incentivo, confiança e sobretudo pelos seus ensinamentos.

Ao Prof. Raimundo Gomes de Oliveira, diretor do Colégio Acreano, onde estudei e tive a oportunidade de aprender.

Agradecimentos

À Universidade Federal do Acre, através da Pró-Reitoria de Pós-Graduação, a todos que fazem a Coordenadoria de Apoio à Pós-Graduação e aos colegas do Departamento de Matemática e Estatística pela colaboração e apoio.

Aos professores do curso de Pós-Graduação em Informática, em especial: Marcelo Alves de Barros, José Antão Beltrão Moura, Marcus Sampaio, Edilson Ferneda e Pedro Sérgio Nicolletti.

Aos professores, M.Sc Bruno Correia da Nóbrega Queiroz e Dr. Francisco de Assis Ferreira Tejo - Universidade Federal da Paraíba, ambos, membros da comissão examinadora, que se dispuseram a colaborar com suas avaliações.

Aos professores, Dr. Mauro Cavalcante Pequeno - Universidade Federal do Ceará e Dr. João Marques de Carvalho - Universidade Federal da Paraíba, que me orientaram durante todo o trabalho.

Aos funcionários da MiniBlio: Arnaldo, Manuela e Zeneide pela amizade.

Aos colegas de mestrado, em especial: Adeilton F. Costa, Claudionor Alencar do Nascimento, Salete Maria Chalub Bandeira, Rute Freitas, Alexandre Viana, Flávia Augusta Matias Dantas, Juliano, Danielle Cristine, Marcus Vinicios Wagner, Adriana Agra e Liliane Cirne.

À Adalberto F. Monteiro Filho pelo apoio e confiança. Ao pessoal do LABCOM: Alberto, Fábio e Lilian.

À Vera Lúcia de Oliveira, Ana Lúcia Guimarães e José Marcelo Rodrigues.

Ao CENAPAD-NE - Centro Nacional de Processamento de Alto Desempenho no Nordeste pela permissão de uso do PVM lá instalado e suporte técnico.

À CAPES pelo apoio financeiro e a todos os amigos que fiz durante toda essa caminhada.

Sumário

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

O problema de resolução de um conjunto de equações lineares é um dos problemas centrais da Matemática Computacional e Ciência da Computação. Excelentes algoritmos para a resolução desses problemas em sistemas com processador único foram desenvolvidos. Por outro lado, algoritmos para resolução de sistemas lineares em computadores paralelos estão em estágio inicial.

A proposta desse trabalho é resolver sistemas lineares de grande porte, usando processamento paralelo. Usaremos um software para desenvolvimento de programas paralelos executáveis em um rede UNIX™ de computadores. A ferramenta é chamada PVM™ (*Parallel Virtual Machine*).

O trabalho apresenta um estudo dos métodos diretos: Eliminação de Gauss e Fatoração LU; e dos métodos iterativos: de Gauss-Jacobi e dos Gradientes Conjugados. Em seguida, são implementadas as rotinas para resolução de sistemas lineares, usando processamento paralelo.

A primeira abordagem utilizada neste trabalho para a implementação da comunicação entre as tarefas cooperantes, não procurou minimizar a passagem de mensagens, resultando em elevado tempo de processamento, devido ao *overhead*. Em uma segunda abordagem, a passagem de mensagens foi otimizada, minimizando o *overhead* e reduzindo consideralvemente o tempo de processamento. Resultados muito melhores aos obtidos em um processamento serial para sistemas lineares de grande porte, foram conseguidos com esta segunda abordagem.

Finalmente, são apresentados os resultados comparativos entre o tempo de execução dos algoritmos implementados para o ambiente serial e o tempo de execução para o ambiente paralelo.

Abstract

The problem of solving a set of linear algebraic equations is one of the central problems in computational mathematics and computer science. Excellent algorithms for this class of problems on single processor systems have been developed. On the other hand, algorithms for solving linear equations on parallel computers are still in its initial stage of development.

The purpose of this work is to solve large size linear systems by using parallel processing. A software tool for developing parallel programs, executable on networked UNIX computers has been emplooyed for this purpose. This tool is known as Parallel Virtual Machine (PVM™).

This work shows a study of direct method, Gaussian Elimination and LU decomposition, as well as of iterative method, Gauss-Jacobi and Conjugate Gradient. The programs for resolution of Linear Systems, using parallel processing, with these algorithms, have been implemented and tested.

The first approach followed in this work to implement communication between cooperating tasks did not try to minimize message passing, during parallel execution of the algorithms. This resulted in high overhead and, consequently, very high processing times. For a second approach, message passing was optimized, minimizing the overhead and reducing, considerably, the processing times. This second approach produced much better times for large size systens, than those yielded by serial processing.

Finally, the comparative results between the running times of sequential and parallel algorithms, are shown.

Capítulo 1

Introdução

1.1 - Introdução

Nos últimos anos temos testemunhado uma crescente aceitação e adoção de processamento paralelo, para computação científica de alto desempenho e aplicações de propósito geral. Esta aceitação foi facilitada por dois desenvolvimentos: Processadores Paralelos (PP) e o uso difundido de computação distribuída.

PP são, agora, os mais poderosos computadores no mundo. Estas máquinas combinam, de algumas centenas a poucos milhares de CPU, em um único gabinete, ligado a centenas de gigabytes de memória. Oferecem enorme poder computacional e são usados para resolver problemas computacionais de grande porte, tais como modelagem de climas e projeto de remédios. À medida que a simulação se torna mais realística, o poder computacional requerido para produzí-la deve crescer, proporcionalmente.

O segundo maior desenvolvimento, atingindo a resolução de problemas científicos, é a computação distribuída. Computação distribuída é um processo, por meio do qual computadores, conectados através de uma rede, são usados, coletivamente, para resolver um único grande problema.

Uma das coisas comuns entre computação distribuída e PP é a noção de passagem de mensagens. Em todo processamento paralelo, dados precisam ser trocados entre tarefas cooperantes [Sunderam 94].

A programação paralela consiste, basicamente, em dividir um programa em vários módulos, a serem executados em diferentes estações, paralelamente, visando a solução do problema. Cada módulo, comumente chamado de "tarefa", trabalha sobre a parte de dados que lhe é conferido e, de acordo com a estrutura do programa, troca os resultados com outras tarefas, ou com uma Tarefa-Mestra. Essa troca de resultados é feita através do que se conhece por

message passing, o que pode se dar entre as várias CPU de uma mesma máquina ou até mesmo entre máquinas remotas.

Ao criar um programa paralelo, é fundamental ter em mente quais as partes do código que são paralelizáveis e como dividir as tarefas entre as CPU. De uma maneira geral, as partes paralelizáveis do código, consistem de funções do programa que não dependem umas das outras e que podem ser executadas simultaneamente. Um exemplo típico de código paralelizável, seria o problema de calcular a integral de uma função, pelo método de somatório simples. A forma mais direta de resolver este problema, em paralelo, seria dividir a função em partições e passá-las para as diferentes CPU, de modo que cada CPU calcule a integral na respectiva região. Ao final do cálculo, cada CPU passaria o seu resultado para uma outra CPU, que se encarregaria de somar os valores parciais, obtendo, assim, a integral.

Ao desenvolver um algoritmo paralelo, devemos ter sempre em mente o que se quer paralelizar, o custo de processamento de cada tarefa (isto é, quanta carga cada tarefa vai impor à CPU) e qual a arquitetura de máquinas que possuímos. Assim, ao dividir um problema temos que notar a importância de balancear a divisão dos processos, de modo que uma CPU mais poderosa não fique sub-utilizada, ou que uma com menor poder de processamento não seja sobrecarregada.

Sem dúvida, a comunicação é o maior gargalo da execução de programas paralelos. Assim, devemos procurar minimizá-lo, afim de evitarmos uma queda de desempenho na execução. A maneira mais óbvia de minimizar a comunicação, é diminuir o número de processadores envolvidos na solução do problema, ou minimizar a passagem de parâmetros entre os processos. Um conceito muito importante em paralelismo é o de granularidade, que consiste no "grau de divisão" dos dados de um problema entre as CPU que participarão da solução. Assim, a granularidade está diretamente relacionada com o tempo de processamento total (geralmente quanto maior ela for, menor será o tempo de processamento) e com o tempo de comunicação entre as CPU (uma vez que quanto mais CPU participarem da tarefa, maior será o tempo de comunicação).

Uma consideração importante é a sincronização entre as tarefas. Uma vez que as CPU podem, geralmente possuir carga e poder de processamento diversos, há a possibilidade de que uma CPU termine a sua tarefa antes de outra e tenha que passar dados para aquela que ainda está trabalhando. Assim, é importante que o programador tenha uma boa noção da ordem do tempo de execução dos processos, para que não deixe um processo esperando por muito tempo a resposta de outra CPU, o que certamente atrasará a execução.

Dessa forma, vemos que a programação paralela não envolve somente o desenvolvimento de programas em si, mas também várias considerações importantes, no sentido de otimizar o código, com a finalidade de obter o maior desempenho possível.

Com o avanço tecnológico e o barateamento dos custos de hardware, *clusters* de *workstations* se tornaram opções viáveis para o desenvolvimento de aplicações.

A programação distribuída, entretanto, envolve um novo conjunto de aspectos que não são cobertos pela programação convencional. Com o objetivo de facilitar o trabalho dos programadores de aplicações paralelas, foi criado o PVM™ (*Parallel Virtual Machine*), um pacote de software para desenvolvimento de programas paralelos, executável em redes UNIX™ ou LINUX™ de computadores. O PVM™ permite que uma coleção heterogênea de estações de trabalho e supercomputadores funcione como uma única máquina paralela de alto desempenho.

O PVM™ é um projeto de pesquisa em andamento. O software é distribuído livremente e está sendo usado em muitas aplicações computacionais ao redor do mundo. Ele é portável e executa em uma grande variedade de plataformas modernas. Ele foi bem aceito pela comunidade mundial de computação e é usado com sucesso para a resolução de problemas de grande porte na ciência, indústria e negócios [Sunderam 94].

Para medir o custo da execução de um programa, costumeiramente definimos uma função complexidade (custo) O, onde $O(n)$ é a medida do tempo requerido para executar o algoritmo em um problema de tamanho n.

Em geral, o custo de obtenção da solução aumenta com o tamanho n do problema. Se o valor de n é suficientemente pequeno, mesmo um algoritmo ineficiente não terá muito trabalho para executá-lo, de modo que a escolha de um algoritmo para um pequeno problema não é crítica. Em muitos casos, entretanto, quando n cresce, surge uma situação onde o algoritmo já não pode ser executado em um período de tempo aceitável. Dizemos que a complexidade de um algoritmo é $O(\log n)$ (leia-se "ordem log n"), se o processamento, pelo algoritmo, de um problema de tamanho n, leva um tempo proporcional a $\log n$ [Kronsjö 87]. É equivalente dizer que:

a) a complexidade do algoritmo é da ordem $\log n$; isto pode também ser escrito como "$O(\log n)$";

b) o algoritmo é executado em tempo $O(\log n)$;

c) a quantidade de trabalho exigido pelo algoritmo é proporcional a $\log n$, ou é de $O(\log n)$.

Os algoritmos que operam sobre vetores, requerem $O(n)$ operações, onde n é o número de componentes dos vetores; os algoritmos que operam sobre matrizes quadradas e vetores, têm

complexidade $O(n^2)$, onde n é o número de colunas ou de linhas da matriz; e, finalmente, os algoritmos que operam com matrizes, como por exemplo algoritmos para multiplicação de matrizes, têm complexidade $O(n^3)$.

1.2 - Objetivo da Dissertação

O problema de resolução de um sistema de equações lineares é um dos problemas centrais da Matemática Computacional e Ciência da Computação. Muitos métodos para a solução desse problema envolvem a triangularização de uma matriz, seguida por uma substituição regressiva.

O objetivo desse trabalho é estudar a resolução de sistemas lineares de grande porte usando processamento paralelo, através de uma analise comparativa de desempenho entre este processamento e o processamento serial convencional.

Nossa investigação de métodos para resolução de sistemas de equações lineares em uma rede UNIX™ de computadores utilizou os seguintes métodos:

1. Eliminação de Gauss
2. Fatoração LU
3. Método iterativo de Gauss-Jacobi
4. Método iterativo dos Gradientes Conjugados.

Os métodos iterativos foram populares na década de 50 e uma das classes desses métodos foi amplamente analisada por David Young, cujo livro ([Young 71]) é um tratado sobre o assunto. Young e seus colaboradores continuam trabalhando ainda em métodos iterativos, dentre os quais destacam-se o método iterativo de Gauss-Jacobi, o de Gauss-Seidel e o método iterativo dos Gradientes Conjugados.

A compreensão dos métodos diretos foi possível graças aos trabalhos de J. Wilkinson [Wilkinson 63] no início da década de 60. A Eliminação de Gauss está definitivamente consagrada, graças aos trabalhos de Wilkinson e ao sucesso do método dos elementos finitos, que a utiliza para resolver sistemas lineares contendo milhares de equações.

O princípio utilizado na resolução numérica dos métodos diretos é o da divisão e conquista. Em geral a matriz dos coeficientes A é representada como o produto de duas matrizes, reduzindo, deste modo, o tamanho do problema a ser resolvido.

Os métodos diretos, tais como Eliminação de Gauss e similares, podem gerar elementos não nulos em posições da matriz que originalmente continham zeros. Essa geração, conhecida como *fill-in* (preenchimento) prejudica a eficiência dos algoritmos se a matriz dos coeficientes for esparsa, isto é, se a porcentagem dos elementos nulos for maior que a dos elementos não nulos. A

exploração da estrutura esparsa se faz armazenando, somente, os elementos não nulos, usando para tal uma estrutura de dados que permita identificar cada elemento da matriz.

Faremos modificações nos algoritmos desses métodos, caso sejam necessárias, para adequá-los ao ambiente PVM™. Implementaremos os algoritmos adaptados e efetuaremos a avaliação do desempenho dos algoritmos paralelos, comparando o desempenho dos mesmos com o dos algoritmos implementados em arquitetura seqüencial.

Algoritmos para computação paralela devem ser concebidos para cada ambiente (hardware + software). Usamos o ambiente PVM™ que está disponível no Centro Nacional de Processamento de Alto Desempenho no Nordeste (CENAPAD-NE), localizado na Universidade Federal do Ceará, em Fortaleza. O CENAPAD-NE é originário de um projeto da Secretaria de Ciência e Tecnologia do Estado do Ceará (SECITECE), da Financiadora de Estudos e Projetos (FINEP) e do Ministério da Ciência e Tecnologia (MCT). O endereço do CENAPAD-NE na Internet é http://www.cenapadne.br/

1.3 - Organização da Dissertação

A Dissertação contém 6 capítulos e 2 apêndices. O primeiro capítulo é uma introdução e os cinco restantes seu desenvolvimento.

O capítulo 2 apresenta os conceitos básicos de paralelismo e a Taxinomia de Flynn.

O capítulo 3 apresenta o software PVM™ (*Parallel Virtual Machine*) e enfoca, também, o paradigma geral para programação de aplicações PVM™.

O capítulo 4 apresenta uma revisão dos métodos implementados (Eliminação de Gauss, Fatoração LU, método de Gauss-Jacobi e método dos Gradientes Conjugados) com seus respectivos algoritmos. Também é feito um estudo dos erros na obtenção da solução de sistemas de equações lineares e um estudo sobre pivoteamento.

O capítulo 5 apresenta uma descrição das matrizes dos coeficientes dos sistemas lineares usados nos testes, a metodologia de testes utilizada, os algoritmos paralelos e os resultados dos testes realizados e uma avaliação comparativa do desempenho dos algoritmos paralelos.

O capítulo 6 apresenta conclusões e sugestões para trabalhos futuros.

O apêndice A mostra como executar e escrever aplicações PVM™, usando as linguagens de programação C e FORTRAN. Em seguida, é feita uma descrição das principais rotinas do PVM™ usadas na implementação dos programas paralelos.

O apêndice B mostra uma listagem dos algoritmos paralelos, para os métodos de resolução de sistemas lineares estudados.

Capítulo 2

Paralelismo

2.1 - Introdução

Problemas de física nuclear, previsão das condições climáticas, projeto de aeronaves, termodinâmica, sismologia e outros, sempre necessitaram de grande capacidade de processamento de dados. Esforços foram desenvolvidos na obtenção de circuitos mais rápidos, que permitiram acelerar os computadores e, portanto, melhorar sua capacidade de processamento. No entanto, estas pesquisas resultaram em pequenas evoluções, enquanto que as necessidades de processamento cresceram de maneira bem mais rápida.

Para superar estas limitações, os pesquisadores em Arquitetura de Computadores desenvolveram, ao longo dos anos, técnicas de projeto chamadas de "operações concorrentes", que permitem aumentar, de forma apreciável, a capacidade de processamento das máquinas, para uma dada tecnologia. O princípio geral destas técnicas é a obtenção de paralelismo no tratamento dos dados, de onde surgiu o nome geral de "Processamento Paralelo", para designar estas técnicas [Navaux 90].

2.2 - Conceitos Básicos de Paralelismo

- **Definição**: Vários processadores cooperando para executar, de forma simultânea, partes de um mesmo programa.
- **Finalidade**: Estratégia para executar mais rapidamente, atividades complexas. Um algoritmo grande e complexo pode ser dividido em pequenas tarefas, executadas simultaneamente por n processadores que se comunicam entre si.
- **Paralelismo de dados**: Cada processador executa as mesmas instruções sobre dados diferentes;
- **Paralelismo funcional**: Cada processador executa instruções diferentes, que podem ou não operar sobre o mesmo conjunto de dados.

- **Multiprogramação**: Um processador executando concorrentemente vários segmentos de programas (sistema de tempo compartilhado).
- **Multiprocessamento**: Vários processadores executando processos diferentes utilizando, para tanto, memória compartilhada ou memória distribuída.
- **Memória Compartilhada** (*Shared Memory*): Vários processadores compartilhando uma única memória. A memória é acessada por somente um processador de cada vez. Veja a Figura 2.1.

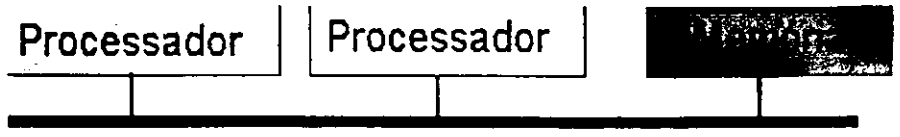

Figura 2.1 - Memória Compartilhada

Vantagens:

1. Aumento da velocidade;
2. Os processadores trabalham independentemente.

Desvantagens:

1. Largura de faixa limitada;
2. Limite no número de processadores;
3. São necessárias técnicas de sincronização para leitura e gravação.

Exemplo:

Cray Y-MP (Centro Nacional de Supercomputação da UFRGS).

- **Memória Distribuída** (*Distributed Memory*): Vários processadores, que operam independentemente, onde cada um possui sua própria memória. Os dados são compartilhados através da rede, usando o mecanismo de *message passing*. Veja a Figura 2.2.

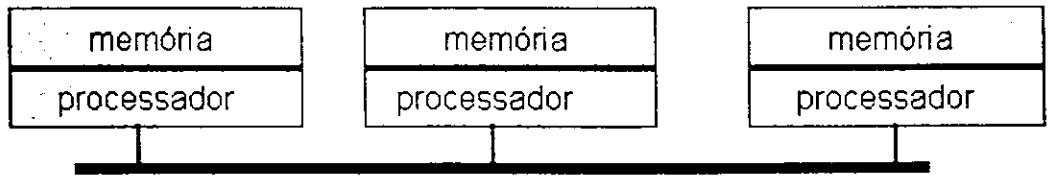

Figura 2.2 - Memória Distribuída

Vantagens:

1. Velocidade no acesso à memória, sem interferência;
2. Não existe limite para o número de processadores.

Desvantagens:

1. O usuário é responsável pelo sincronismo e recebimento de dados;
2. Elevado *overhead* devido à comunicação.

Exemplo:

IBM Risc/6000 SP2 (CENAPAD-NE)

- **Passagem de mensagem** (*message passing*): Método de comunicação entre vários processadores que possuem memória própria. Se baseia na transmissão dos dados via uma rede de interconexão, segundo as regras do protocolo da rede (exemplo: IP).

2.3 - Taxinomia de Flynn

Qualquer computador, serial ou paralelo, opera executando instruções sobre dados. Um fluxo de instruções (o algoritmo) diz ao computador o que fazer em cada caso. Um fluxo é uma seqüência de objetos tais como dados, ou de ações, tais como instruções. Um fluxo de dados (a entrada para o algoritmo) é afetado por essas instruções [Akl 89].

Existem diversas classificações propostas para as arquiteturas de máquinas como a de Feng [Feng 72], que é baseada no processamento serial × paralelo, e a de Handler [Handler 77] que determina o grau de paralelismo e *pipeline* em vários níveis. Entretanto, a classificação melhor aceita até a presente data é a de Flynn [Flynn 72], em especial devido a sua grande simplicidade.

A classificação de Flynn baseia-se na multiplicidade de fluxo de dados e instruções, e propõe quatro categorias de máquinas. Segundo ela, o ponto principal do processo de um computador é a execução de um conjunto de instruções sobre um conjunto de dados. Fluxo é uma seqüência de instruções ou de dados executados num único processador. Portanto, fluxo de instruções é uma seqüência de instruções executadas por uma máquina enquanto que fluxo de dados é uma seqüência de dados, inclusive de entrada, resultados parciais ou intermediários, utilizada pelo fluxo de instruções. As quatro categorias de arquiteturas propostas por Flynn (usando o modelo baseado nos diferentes fluxos usados em processos computacionais) são:

SISD: "Single Instruction stream, Single Data stream"

SIMD: "Single Instruction stream, Multiple Data stream"

MISD: "Multiple Instruction stream, Single Data stream"

MIMD: "Multiple Instruction stream, Multiple Data stream"

2.3.1 - Computadores SISD

Um computador nesta classe consiste de um único Elemento de Processamento, recebendo um único fluxo de instruções, que opera sobre um único fluxo de dados. Em cada passo durante a computação, a unidade de controle emite uma instrução que opera em dados obtidos da unidade de memória. Tal instrução pode dizer ao processador, para executar, por exemplo alguma

operação lógica ou aritmética em alguns dados e então colocar de volta o resultado na memória. Veja a Figura 2.3.

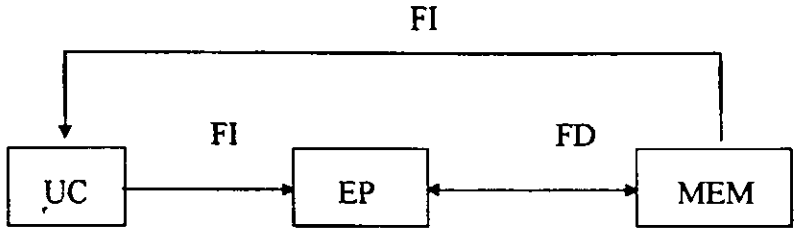

Figura 2.3 - Computador do tipo SISD.

Na Figura 2.3 foi adotada a seguinte legenda:

UC - Unidade de Controle

EP - Elemento Processador

MEM - Memória

FI - Fluxo de Instruções

FD - Fluxo de Dados

A grande maioria dos computadores de hoje segue este modelo inventado por John Von Neumann e seus colaboradores, no final dos anos 40. Um algoritmo para um computador desta classe é dito ser seqüencial ou serial [Akl 89].

2.3.2 - Computadores SIMD

Nesta categoria, um computador paralelo consiste de *n* processadores idênticos. Cada um dos *n* processadores possui sua própria memória local, onde ele pode armazenar programas e dados. Todos os processadores operam sob o controle de uma única instrução, emitida por uma unidade central de controle. Equivalentemente, os *n* processadores têm cópias idênticas de um único programa, cada cópia sendo armazenada em sua memória local. Há *n* fluxos de dados, um por processador. Este modelo corresponde aos processadores matriciais (*array processors*). Veja a Figura 2.4.

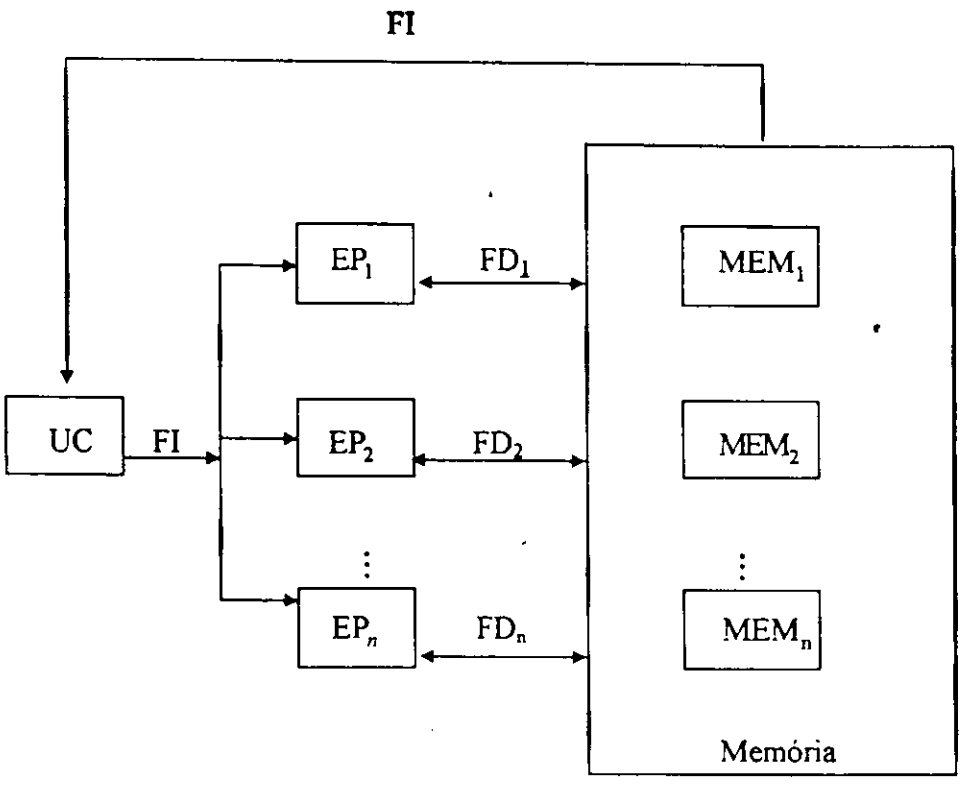

Figura 2.4 - Computadores do tipo SIMD.

Os processadores operam sincronamente. Em cada passo, eles executam a mesma instrução, cada um sobre um dado diferente. A instrução pode ser simples (como somar dois números) ou complexa (como juntar duas listas de números). Similarmente, os dados podem ser simples (um número) ou complexo (vários números). Algumas vezes, pode ser necessário ter somente um subconjunto dos processadores executando uma instrução. Esta informação pode ser codificada na própria instrução, desse modo informando ao processador se estará ativo (e executar a instrução) ou inativo. Processadores que estão inativos durante uma instrução ou aqueles que completaram a instrução antes dos outros podem, ficar ociosos até que a próxima instrução seja emitida. O tempo de execução entre duas instruções pode ser fixo ou depender da instrução sendo executada.

Em muitos problemas que são resolvidos em um computador SIMD, é desejável que os processadores sejam capazes de se comunicar entre si durante a computação, para trocar dados ou resultados intermediários. Isto pode ser realizado de duas maneiras, resultando em duas subclasses: computadores SIMD onde a comunicação entre os processadores é feita através de memória compartilhada (*Shared Memory*) e aqueles onde a comunicação é feita via uma rede de interconexão [Akl 89].

2.3.3 - Computadores MISD

Neste caso, *n* processadores, cada um dos quais com sua própria unidade de controle, compartilham uma unidade de memória comum, onde residem os dados, como mostra a Figura 2.5 . Há *n* fluxos de instruções e um único fluxo de dados. Em cada passo, um dado recebido da memória é operado por todos os processadores, simultaneamente, de acordo com as instruções recebidas da sua unidade de controle. Assim, o paralelismo é realizado, fazendo com que os processadores realizem tarefas diferentes, ao mesmo tempo, sobre o mesmo dado. Esta classe, serve naturalmente para aquelas computações que requerem uma única entrada, submetida a várias operações [Akl 89].

Embora seja fácil imaginar e projetar computadores MISD, tem havido pouco interesse neste tipo de arquitetura paralela. Não há máquina construída usando este modelo.

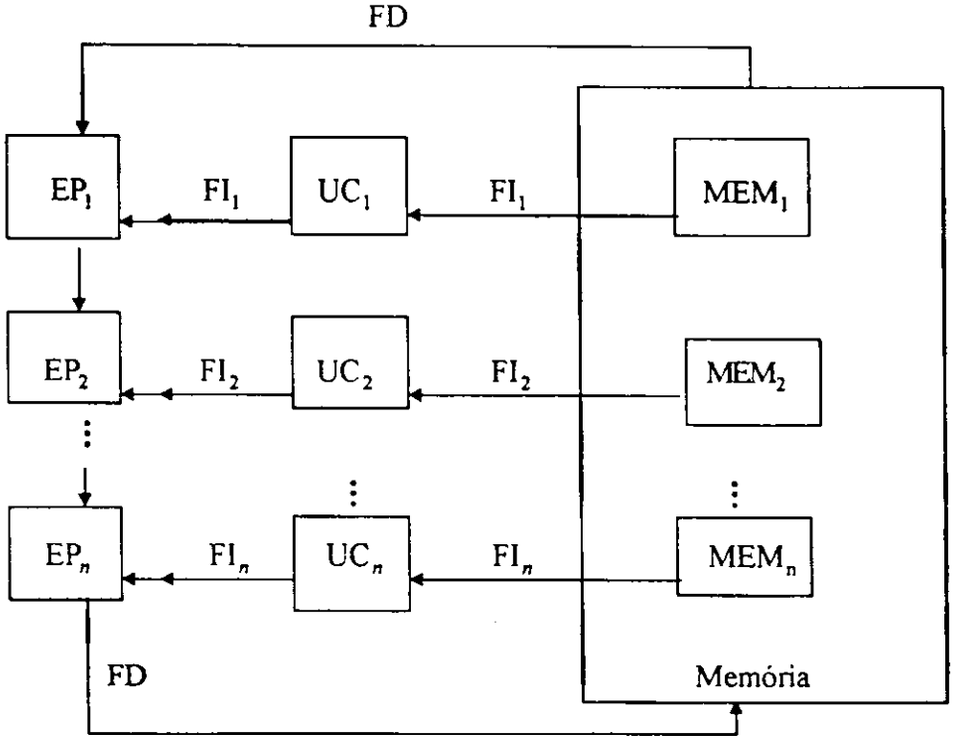

Figura 2.5 - Computadores do tipo MISD

2.3.4 - Computadores MIMD

Esta classe de computadores é mais geral e mais poderosa em nosso paradigma de computação paralela, o qual classifica computadores paralelos conforme o fluxo de instrução e/ou o fluxo de dados é duplicado. Nesta classe se enquadram os multiprocessadores.

Aqui temos *n* processadores, *n* fluxos de instruções e *n* fluxos de dados, como mostra a Figura 2.6. Os processadores, aqui, são do tipo usado em computadores MISD, no sentido de que cada um possui sua própria unidade de controle, sua memória local e sua própria unidade de

aritmética e lógica. Isso faz esses processadores mais poderosos que aqueles usados por computadores SIMD.

Cada processador opera sob o controle de um fluxo de instrução, emitido por sua unidade de controle. Assim os processadores estão, potencialmente, executando diferentes programas em diferentes dados, enquanto resolvem subproblemas de um único problema. Isto significa que os processadores operam, tipicamente, assincronamente. O que mantém a interação entre os processadores é a memória global, ou sistema de mensagens, gerenciados por um sistema operacional único.

Como nos computadores SIMD, a comunicação entre processadores é executada através de memória compartilhada ou por meio de redes de interconexão. Computadores MIMD que compartilham uma memória comum, são freqüentemente referidos como **Multiprocessadores** (ou máquinas fortemente acopladas), enquanto que aqueles com redes de interconexão, são conhecidos como **Multicomputadores** (ou máquinas fracamente acopladas).

Multicomputadores são, algumas vezes, referidos como sistemas distribuídos. A distância é usualmente baseada na distância física separando os processadores e é, portanto, freqüentemente subjetiva. A regra do dedo polegar é a seguinte: se todos os processadores estão nas proximidades uns dos outros (na mesma sala, digamos), então eles formam um **Multicomputador**; de outra forma (em várias cidades, digamos) formam um **Sistema Distribuído** [Akl 89]. Um exemplo de Sistema Distribuído é o mostrado na Figura 2.7 formado por um Supercomputador SP2 e duas estações de trabalho IBM RISC6000.

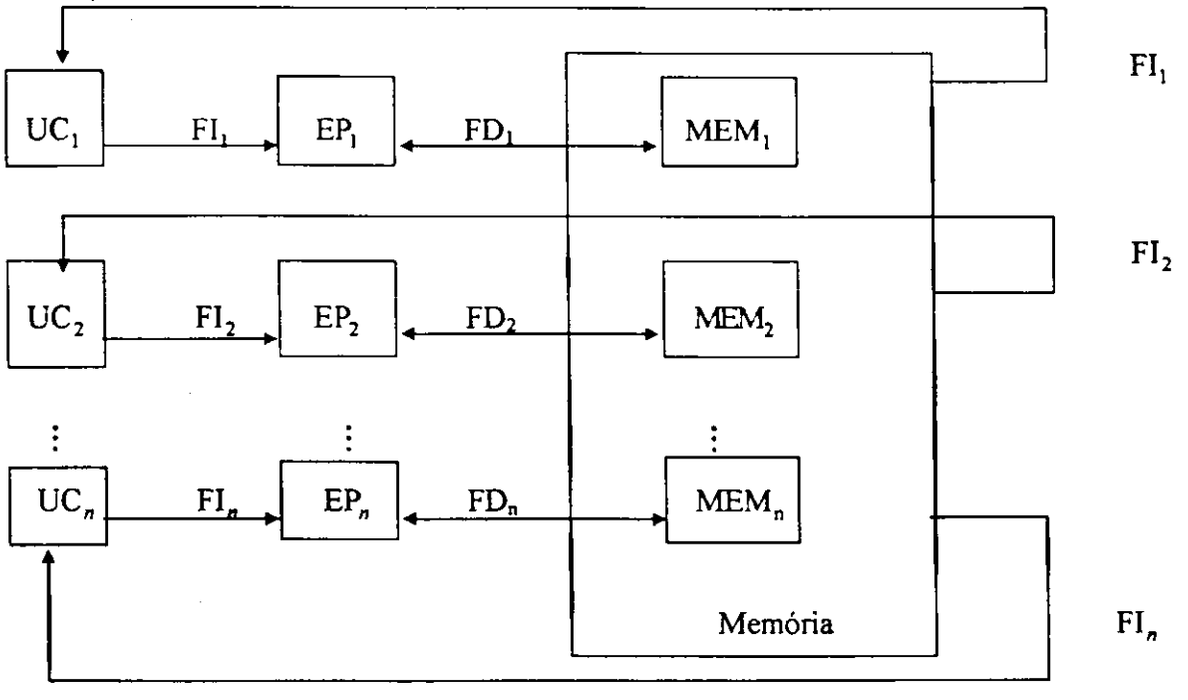

Figura 2.6 - Computadores do tipo MIMD.

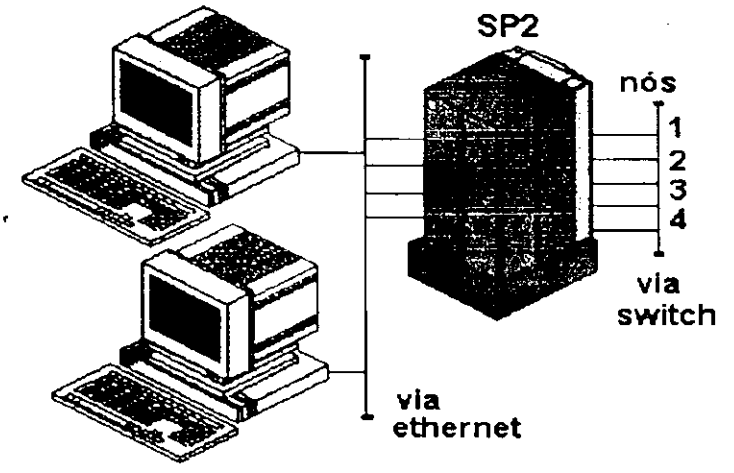

Figura 2.7 – Sistema Distribuído – CENAPAD-NE

2.4 - Resumo

As inúmeras possibilidades de organização de um grande número de processadores têm produzido uma grande variedade de computadores paralelos, com o objetivo de atingir alto desempenho em classes de problemas tradicionalmente dominadas pelos supercomputadores, a um custo relativamente baixo. O primeiro método para classificar arquiteturas de computadores, amplamente disseminado, foi proposto por Flynn em 1966. O método se baseia nas possibilidades de combinação entre uma ou mais seqüências de instruções, atuando sobre uma ou mais seqüências de dados.

O próximo capítulo trata de uma descrição de um software para computação paralela chamado PVM™ (*Parallel Virtual Machine*), o qual permite que uma rede de computadores UNIX™ funcione como uma única máquina paralela de alto desempenho, chamada *Máquina Virtual Paralela*.

Capítulo 3

O PVM

3.1 - Introdução

Este capítulo apresenta um pacote de software para desenvolvimento de programas paralelos, executáveis em uma rede UNIX™ de computadores. A ferramenta chamada *Parallel Virtual Machine* (PVM™), permite que uma coleção heterogênea de estações de trabalho e supercomputadores funcione como uma única máquina paralela de alto desempenho, chamada *Máquina Virtual Paralela*.

O projeto do sistema começou no verão de 1989, no Laboratório Nacional de Oak Ridge. O sistema protótipo foi construído por Vaidy Sunderam e Al Geist. Esta versão do sistema foi usada internamente no laboratório e não foi liberada para uso público. A versão 2.0 do sistema foi escrita na Universidade do Tennessee e liberada para divulgação, em março de 1991. Durante o ano seguinte, o PVM™ começou a ser usado em muitas aplicações científicas. Depois de várias mudanças sugeridas por usuários, uma revisão completa foi empreendida, e a versão 3.0 foi completada em fevereiro de 1993. A versão que usamos nos testes é a 3.3 (a qual chamaremos, simplesmente, de PVM™). O software é distribuído livremente e está sendo usado hoje em muitas aplicações computacionais.

O software é o sustentáculo do projeto de pesquisa em computação de redes heterogêneas, uma iniciativa colaborativa entre o Laboratório Nacional de Oak Ridge, a Universidade do Tennessee, a Universidade Emory e a Universidade Carnegie Mellon, todas nos Estados Unidos da América.

O sistema evoluiu, nos últimos anos, para uma tecnologia viável em processamento paralelo distribuído. O PVM™ suporta um modelo simples de passagem de mensagens.

O software foi projetado para reunir recursos computacionais e fornecer aos usuários uma plataforma paralela para execução de suas aplicações, sem levar em conta o número de computadores diferentes que eles usam e onde estão localizados. Quando o PVM™ está

corretamente instalado, ele é capaz de controlar os recursos combinados de uma plataforma de computação de redes heterogêneas e entregar altos níveis de desempenho e funcionalidade.

3.2 - O Sistema PVM

O software é um pacote que permite a um programador criar e acessar um sistema de computação concorrente, composto por uma rede de processadores fracamente acoplados. O hardware reunido na máquina virtual do usuário pode ser composto de uma *workstation*, várias máquinas vetoriais ou supercomputadores paralelos. Os elemento individuais podem ser de um mesmo tipo (homogêneo) ou de tipos diferentes (heterogêneo) ou uma mistura deles, contanto que todas as máquinas usadas estejam conectadas através de uma ou mais redes. Estas redes podem ser tão pequenas como uma LAN, ou grandes como a Internet. Esta habilidade de reunir diversas arquiteturas sob um controle central, permite ao usuário PVM™ dividir o problema em subtarefas e determinar que cada uma delas seja executada pelo processador mais indicado, o que possibilitará aos usuários um grande ganho no desempenho de seus programas, se desenvolvidos em paralelo. Esse software atua interligando várias máquinas da rede, formando um ambiente pré-estabelecido pelo usuário. Esse ambiente passa a ser visto de uma forma transparente, como se fosse um único nó do sistema. Sob o PVM™ uma coleção de computadores seriais, paralelos e vetoriais, aparece como um único computador de memória distribuída. Ele fornece as ferramentas para lançar tarefas na máquina virtual e permite que estas se comuniquem durante o processamento, a fim de solucionar o problema.

O PVM™ é um conjunto integrado de ferramentas de software e bibliotecas que emulam uma plataforma de computação concorrente heterogênea, flexível, de propósito geral, de computadores interconectados. O principal objetivo do PVM™ é possibilitar que uma coleção de computadores seja usada, cooperativamente, para computação paralela e concorrente.

O software usa o modelo de passagem de mensagens, para permitir a programadores explorar a computação distribuída, através de uma grande variedade de tipos de computadores, incluindo os Supercomputadores. O conceito chave no PVM™ é que ele faz com que uma coleção de computadores pareça como uma grande máquina *virtual*, daí o seu nome. O software fornece uma plataforma uniforme, na qual programas podem ser desenvolvidos de modo eficiente, usando o hardware disponível. Ele transparentemente manipula todo o roteamento de mensagens, conversão de dados e escalonamento através da rede de computadores de arquiteturas variadas.

O usuário escreve uma aplicação, como uma coleção de tarefas cooperantes. Estas tarefas acessam recursos do PVM™ através de uma biblioteca de rotinas. Estas rotinas, por sua vez permitem o início e o fim de tarefas na rede, bem como comunicação e sincronização. As

primitivas de passagem de mensagens PVM™ são orientadas a operações heterogêneas, envolvendo construções fortemente tipificada para armazenamento e transmissão. Construção de comunicações incluem aquelas para emissão e recepção de estruturas de dados, bem como primitivas de alto nível, tais como espalhamento, barreira de sincronização e soma global.

Em qualquer estágio da execução de uma aplicação concorrente, qualquer tarefa pode iniciar ou parar outras tarefas, adicionar ou remover computadores da máquina virtual. Qualquer processo pode se comunicar e/ou sincronizar com qualquer outra tarefa. Os princípios sobre os quais o PVM™ está baseado incluem:

- Máquina virtual configurável pelo usuário: as tarefas computacionais de uma aplicação são executadas em um conjunto de máquinas, que são selecionadas pelo usuário, para um dado programa PVM™ em execução. Máquinas com um único processador e multiprocessadores (incluindo computadores com memória compartilhada e distribuída) podem fazer parte da máquina virtual. A máquina virtual pode ser alterada através da adição e remoção de máquinas, durante a operação (uma importante característica para tolerância a falhas).
- Acesso seletivo ao hardware: o programa paralelo pode "ver" a máquina virtual como uma coleção de CPU sem características especiais, ou pode aproveitar as vantagens da arquitetura de cada CPU da máquina virtual.
- Computação baseada em processos: a unidade de paralelismo é a tarefa (freqüentemente mas nem sempre um processo UNIX™), a linha de controle alterna entre comunicação e computação. Nenhum mapeamento processo-processador é incluído ou imposto pelo PVM™; em particular múltiplas tarefas podem ser executadas em um único processador.
- Modelo de *message passing* explícito: as tarefas que formam o programa paralelo dividem o trabalho através, da troca de mensagens determinada pelo usuário, a ser realizada pelas mesmas. O tamanho das mensagens é limitado somente pela quantidade de memória disponível. As mensagens são passadas, entre tarefas, pela rede de conexão. A conversão de dados é feita automaticamente entre as máquinas que não compartilham a mesma representação de dados
- Suporte à heterogeneidade: o sistema suporta heterogeneidade em termos de máquinas, redes e aplicações. Com respeito a passagem de mensagens, ele permite que mensagens contendo mais de um tipo de dados sejam trocadas entre máquinas, tendo diferentes modos de representação de dados.

- Suporte a multiprocessador: o sistema usa recursos próprios de passagem de mensagens nos multiprocessadores para tirar vantagem do hardware básico.

O sistema PVM™ é composto de duas partes:

PVM Daemon A primeira parte é um *daemon*, chamado *pvmd3* (algumas vezes usaremos o nome abreviado *pvmd*), que reside em todos os computadores que compõem a máquina virtual (um exemplo de programa *daemon* é o programa de correio eletrônico que executa em *background* e manipula todos os *email* que estão chegando e saindo em um dado computador). O *pvmd3* foi projetado de modo que qualquer usuário com um *login* válido possa instalar este *daemon* em uma máquina.

Quando um usuário deseja carregar uma aplicação, ele primeiro cria uma máquina virtual disparando o PVM™. A aplicação pode então ser iniciada em qualquer das máquinas. Múltiplos usuários podem configurar uma máquina virtual sobreposta, e cada usuário pode executar várias aplicações, simultaneamente. Os *pvmd* pertencentes a um usuário não interagem com aqueles pertencentes a outros usuários, para reduzir o risco com a segurança, e minimizar a colisão de um usuário com outro. O *pvmd* serve como um roteador e controlador de mensagens. Ele fornece um ponto de contato, autenticação, controle de processos e detecção de falhas. Um *pvmd* inativo, ocasionalmente verifica se os outros iguais a ele ainda estão em execução. Mesmo se um programa aplicação falhar, os *pvmd* continuam em execução.

Cada *pvmd* mantém uma lista de todas as tarefas sob seu gerenciamento. Muitas tarefas estão também em uma segunda lista, classificada pelo identificador de processos. Uma tabela de máquinas descreve a configuração de uma máquina virtual. Ela lista o nome, endereço e estado de comunicação de cada máquina. Um *pvmd* encerra suas atividades, quando ele é removido da máquina virtual, morre, perde contato com o *pvmd* mestre, ou quando falha (erro no barramento).

Biblioteca de Programação A segunda parte do sistema PVM™ é uma biblioteca de rotinas de interface chamada **libpvm**. Ela contém um repertório completo de primitivas que são necessárias para a cooperação entre as tarefas de uma aplicação. Esta biblioteca contém rotinas que podem ser chamadas pelo usuário para passagem de mensagens, distribuição de processos, coordenação de tarefas e modificação da máquina virtual. Libpvm é escrita em C e suporta diretamente aplicações C e C++. Este conjunto de linguagens foi incluído, baseado na observação de que a grande maioria das aplicações são escritas em C e FORTRAN, com uma tendência emergente em experimentos com linguagens e metodologias orientadas a objeto. A biblioteca FORTRAN, `libfpvm3.a` (também

escrita em C), é um conjunto de funções *wrapper* que executam as chamadas FORTRAN. A biblioteca libpvm permite a uma tarefa interagir com o *pvmd* e outras tarefas. Ela contém funções para empacotamento (composição) e desempacotamento de mensagens, funções para executar chamadas ao sistema PVM™, através do uso de mensagens para enviar serviços de requisição ao *pvmd*.

Libpvm fornece funções para empacotar todos os tipos de dados primitivos em uma mensagem, em um dos vários formatos de codificação. Há 5 conjuntos de codificadores e decodificadores. Cada *buffer* de mensagens tem um conjunto associado a ele. Quando uma mensagem é criada, o conjunto codificador é determinado através do parâmetro de formato da função `pvm_mkbuf()`. Quando uma mensagem é recebida, os decodificadores são determinados pelo campo codificador do cabeçalho da mensagem. Os dois mais usados empacotam dados nos formatos nativo (máquina nativa) e padrão (XDR).

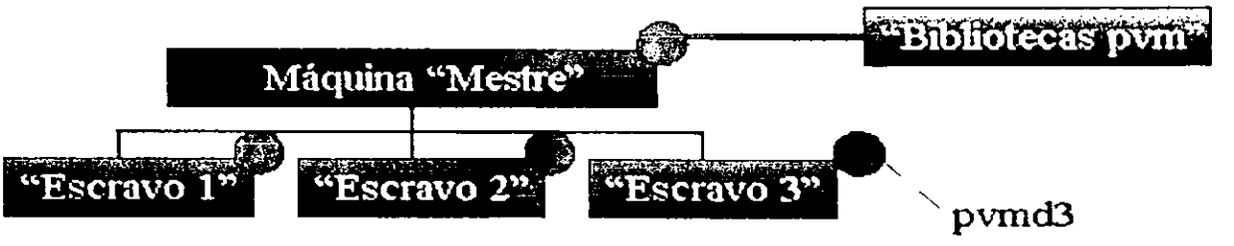

Figura 3.1 - Componentes do PVM™

O modelo de computação PVM™ é baseado na noção de que uma aplicação consiste de várias **tarefas**. Cada **tarefa** é responsável por uma parte da aplicação que está sendo executada. Algumas vezes uma aplicação é paralelizada ao longo de suas funções; isto é, cada tarefa executa uma função diferente, por exemplo, entrada, inicialização, solução, saída e exibição. Este processo é, freqüentemente chamado **paralelismo funcional**. Um método muito comum de paralelizar uma aplicação é chamado **paralelismo de dados**. Neste método, todas as tarefas são as mesmas, mas cada uma delas conhece e resolve somente uma pequena parte dos dados. Este método também é chamado de modelo de computação SPMD (*Single-Program Multiple-Data*). O sistema suporta um ou o outro modelo, ou uma mistura destes. Dependendo de suas funções, as tarefas podem ser executadas em paralelo de maneira sincronizada, embora este não seja sempre o caso.

As ligações das linguagens C e C++ para a biblioteca de interface de usuário PVM™, são implementadas como funções, seguindo a convenção geral usada pela maioria dos sistemas C, incluindo sistemas operacionais como UNIX™. Programas de aplicação escritos em C e C++

acessam a biblioteca de funções PVM™ através da conexão a uma biblioteca chamada `libpvm3.a` que é parte da distribuição padrão.

As ligações da linguagem FORTRAN são implementadas como sub-rotinas, ao invés de funções. Um argumento adicional foi introduzido em cada chamada à biblioteca PVM™, para o resultado de *status* a ser retornado ao programa chamador. As rotinas para colocação e recuperação de dados colocados em *buffer* de mensagens são unificadas, com um parâmetro adicional indicando o tipo do dado.

Todas as tarefas PVM™ são identificadas por um inteiro *identificador de tarefa* (TID). Mensagens são enviadas e recebidas de tarefas, com seus respectivos TID. Uma vez que esses identificadores precisam ser únicos em toda a máquina virtual, eles são fornecidos pelo *pvmd* local e não são definidos pelo usuário. Embora o PVM™ codifique a informação de cada TID, os usuários devem considerar os TID como identificadores inteiros transparentes. O sistema PVM™ contém várias rotinas que retornam valores TID para que as aplicações de usuários possam identificar outras tarefas no sistema.

Daemons PVM™ e tarefas podem compor e enviar mensagens de tamanhos arbitrários contendo dados. Os dados podem ser convertidos usando o sistema de conversão de dados XDR (eXtended Data Representation standard) [Sun 87], quando estiverem passando entre máquinas com formatos de dados incompatíveis.

O emissor de uma mensagem não espera por uma confirmação do receptor, mas continua assim que a mensagem tenha sido mandada pela rede e o *buffer* de mensagens possa ser removido com segurança ou reusado. O PVM™ entrega mensagens com segurança, desde que fornecidas as destinações existentes.

Primitivas de recebimento, bloqueado e não-bloqueado, são fornecidas. Desse modo, uma tarefa pode esperar por uma mensagem sem (necessariamente) consumir tempo de processamento através de *polling*. Também é fornecido um recebimento com tempo de espera, que retorna depois de um certo tempo, caso nenhuma mensagem tenha chegado. Se ignorarmos a recuperação de falhas, então uma aplicação será executada até o seu término ou, se algum componente falhar, ela será encerrada.

O PVM™, usa o modelo de programação Mestre-Escravo. Sob o paradigma Mestre-Escravo, uma tarefa é usada como mestre. O único propósito da Tarefa-Mestra é criar todas as outras tarefas (Tarefas-Escravas) que vão trabalhar no problema, coordenar a entrada de dados iniciais para cada tarefa e coletar os resultados de cada tarefa. As Tarefas-Escravas realizam os

cálculos propriamente ditos. Ao processo de criação dos processos escravos dá-se o nome de *spawning*. Veja a Figura 3.2.

Figura 3.2 - Modelo de programação Mestre-Escravo

No paradigma Mestre-Escravo, a comunicação entre processos é fundamental em, pelo menos, duas situações:

- Transmissão das informações da Tarefa-Mestra às Tarefas-Escravas, a fim de realizarem a computação necessária;
- Recepção (pela Tarefa-Mestra) dos resultados das Tarefas-Escravas, a fim de concluir a fase do processamento

Em PVM™, tal comunicação subdivide-se como na Figura 3.3:

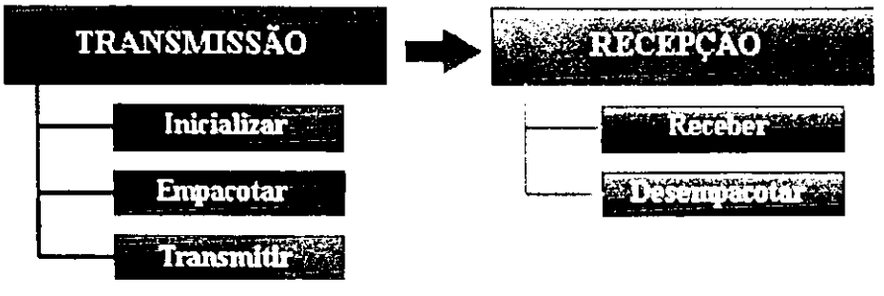

Figura 3.3 - Comunicação entre processos

O paradigma geral para programação de aplicações usando PVM™, é como segue. O usuário escreve um ou mais programas seqüênciais em C ou FORTRAN, que contenham embutidos chamadas à biblioteca PVM™. Cada programa corresponde a uma tarefa compondo a aplicação. Estes programas são compilados para cada arquitetura na máquina virtual, e os arquivos objetos resultantes são colocados em localizações acessíveis na máquina virtual.

Para executar uma aplicação, o usuário comumente inicia uma cópia de uma tarefa (usualmente a "mestra" ou a tarefa de inicialização) manualmente, de uma máquina do sistema virtual. Este processo, subseqüentemente inicia outras tarefas PVM™ (Tarefas-Escravas), eventualmente resultando em um conjunto de tarefas ativas, que então computam localmente e trocam mensagens com cada uma das outras para resolverem o problema.

```
#include "pvm3.h"
main()
{       int cc, tid, msgtag;
        char buf[100];
        printf("Eu sou t%x\n",pvm_mytid());
        cc=pvm_spawn("hello_other", (char**)0,0,"", 1, &tid);
        if (cc==1)
        {       msgtag=1;
                pvm_recv(tid, msgtag);
                pvm_upkstr(buf);
                printf("De: t%x: %s\n", tid, buf)
        }
        else
                printf("Não posso iniciar hello_other\n");
        pvm_exit();
}
```

Figura 3.4 - Programa hello.c

Vemos na Figura 3.4 o corpo do programa hello.c, um exemplo simples que ilustra os conceitos básicos da programação PVM™. Este programa foi projetado para ser chamado manualmente; depois de mostrar seu identificador de tarefa (obtido com `pvm_mytid()`), ele inicia uma cópia de outro programa chamado hello_other.c, usando a função `pvm_spawn()`. Uma distribuição com sucesso levará o programa a executar uma recepção bloqueada através da chamada à rotina `pvm_recv()`; depois do recebimento da mensagem, o programa mostra a mensagem enviada por hello_other e também seu identificador de tarefa; a mensagem é extraída do *buffer*, usando `pvm_upkstr()`. No final, a chamada à rotina `pvm_exit()` desliga o programa do PVM™.

```
#include "pvm3.h"
main()
{       int ptid, msgtag;
        char buf[100];
        ptid=pvm_parent();
        strcpy(buf,"Alo mundo, de:");
        gethostname(buf+strlen(buf),64)
        msgtag=1;
        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid,msgtag);
        pvm_exit();
}
```

Figura 3.5 - Programa PVM hello_other.c

A Figura 3.5 é a listagem do programa "escravo" ou distribuído; sua primeira ação é obter o identificador da Tarefa-Mestra, usando a chamada à rotina `pvm_parent()`. Esta tarefa, então, obtém o nome da máquina e o transmite para a Tarefa-Mestra, usando as três chamadas seguintes - `pvm_initsend()`, para inicializar o *buffer* de emissão; `pvm_pkstr()`, para colocar uma *string* em um *buffer* de emissão; e `pvm_send()` para transmiti-lo ao processo destino especificado por ptid, rotulando a mensagem com o número 1.

Para conhecer maiores detalhes de como iniciar o PVM™ e escrever aplicações PVM™ veja o Apêndice A. Para maiores detalhes sobre as rotinas da biblioteca PVM™ usadas na implementação dos programas paralelos veja o Apêndice B.

3.3 - Resumo

Este capítulo mostrou o que é o PVM™, seus componentes: *daemons* e a biblioteca Libpvm. Mostrou exemplos de programas mestre e programas escravos na linguagem C explicando o funcionamento das rotinas da biblioteca PVM™.

O software pode ser obtido por ftp anônimo de `netlib2.cs.utk.edu`. Olhe no diretório pvm3. O arquivo `index` descreve os arquivos deste diretório e seus subdiretórios. Usando uma ferramenta para navegar na Internet como o Netscape, os arquivos PVM™ podem ser recuperados usando o endereço `http://www.netlib.org/pvm3/index.html`. O software pode ser solicitado por *email*. Para receber este software envie *email* para `netlib@ornl.gov` com a mensagem: `send index from pvm3`. A documentação é distribuída em arquivos *postscript* e inclui um guia do usuário, manual de referência e um cartão de referência rápida

De acordo com especialistas, 75% dos problemas de computação numérica envolvem de alguma forma a solução de problemas de álgebra linear e a resolução de sistemas de equações lineares é um problema que surge em várias áreas do conhecimento.

No capítulo seguinte veremos alguns métodos para a resolução de sistemas de equações lineares com o número de equações igual ao número de incógnitas.

Capítulo 4

Sistemas de Equações Lineares

4.1 - Introdução

Sistemas de equações lineares ocorrem, freqüentemente, em resolução de problemas numéricos, diretamente (por exemplo, analisando circuitos pela aplicação das leis de Kirchhoff), e indiretamente (por exemplo, como resultado do uso de técnicas de diferenças finitas, para obter uma solução aproximada de uma equação diferencial parcial).

4.2 - Sistema Linear

É um conjunto de m equações a n incógnitas $x_1, x_2, x_3, \ldots, x_n$. Restringimos nossa discussão aos casos onde o número de equações e incógnitas é o mesmo. Assim, o sistema

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1,\\
&a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2,\\
&a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = b_3,\\
&\cdots\cdots\cdots\cdots\\
&a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \cdots + a_{in}x_n = b_i,\\
&\cdots\cdots\cdots\cdots\\
&a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n,
\end{aligned} \tag{4.1}$$

é linear.

O sistema linear (4.1) pode ser escrito na forma matricial

$$A\mathbf{x} = \mathbf{b}, \tag{4.2}$$

em que A é matriz $n \times n$ dos coeficientes representados por a_{ij} com $i, j = 1, 2, 3, \ldots, n$, $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ é o vetor das incógnitas e $\mathbf{b} = (b_1, b_2, \ldots, b_n)^T$ é o vetor dos termos independentes.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix} \quad \text{e} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix}. \qquad (4.3)$$

A matriz A é chamada de *matriz incompleta* do sistema. As entradas de A e $\mathbf{b}$ podem ser complexas, com $\mathbf{x}$ podendo ser complexo. Contudo, se todas as entradas de A e $\mathbf{b}$ são reais então os elementos do vetor solução serão também reais. Assumiremos que todos os sistemas são reais, a menos que se diga o contrário.

Sistemas da forma (4.1) podem diferir em tamanho, de um pequeno número de equações a grandes sistemas contendo centenas, ou até mesmo milhares, de equações. Há também um vasto espectro para a estrutura da matriz A: não simétricas, simétricas, bandas, triangulares e diagonais. Em geral, a grande maioria dos sistemas que são resolvidos por métodos diretos, tendem a ser esparsos (muitos elementos nulos) ou altamente estruturados (freqüentemente, os elementos não nulos estão concentrados nas bandas) ou ambos.

Devemos observar, ainda, que talvez a forma mais natural de resolver um sistema $A\mathbf{x} = \mathbf{b}$ seja escrever $\mathbf{x} = A^{-1}\mathbf{b}$, o que implica em obter a inversa de A. No entanto, é totalmente desaconselhável o cálculo de A^{-1}, uma vez que, a matriz obtida pode diferir muito da verdadeira A^{-1}. Por exemplo, considere a equação $3\mathbf{x} = 18$, que tem como solução $\mathbf{x} = \frac{18}{3} = 6$. Se escrevermos $\mathbf{x} = (3)^{-1} \times 18$, então $\mathbf{x} = (0{,}33333) \times 18 = 5{,}99994$, trabalhando com 5 casas decimais.

Podemos dizer que se a inversa, A^{-1}, da matriz dos coeficientes existe, então a solução existe e é única. Contudo (4.2) não deverá ser resolvida pela determinação de A^{-1} explicitamente, por causa do alto custo computacional e dos problemas de erros de arredondamento.

A condição que A é inversível ou não singular (isto é, A^{-1} pode ser calculada) é equivalente às seguintes afirmações:

- Nenhuma equação no sistema pode ser expressa como combinação linear das outras;
- As colunas (linhas) da matriz dos coeficientes são linearmente independentes;
- O determinante de A é diferente de zero.

Esperamos que o software numérico possa detectar e relatar uma matriz singular. O maior problema é determinar os efeitos dos erros de arredondamento na computação e isso requer uma

análise detalhada dos algoritmos. Além disso, a matriz pode ser mal condicionada, isto é, pequenas mudanças na matriz dos coeficientes ou nos elementos do vetor dos termos independentes, podem causar grandes mudanças nos elementos da solução computada.

O estudo de métodos para a resolução de sistemas lineares é necessário, pois, em geral, os problemas práticos exigem a resolução de sistemas lineares de grande porte. Felizmente, existe um método cujo tempo necessário para a solução varia polinomialmente com $a_3n^3 + a_2n^2 + a_1n + a_0$. Como n^3 é o termo dominante para $n > 1$, o tempo é proporcional a esse termo, representando esse fato com a notação $O(n^3)$. Para n = 50, tem-se n^3 = 125.000, e o tempo necessário para resolver um sistema de equações lineares será da ordem de segundos. Entre os métodos diretos, destacam-se os métodos de Eliminação, que evitam o cálculo direto da matriz inversa de A e, além disso, não apresentam problemas com o tempo de execução, como a Regra de Cramer.

Os métodos diretos são aqueles que, a menos de erros de arredondamento, fornecem a solução exata do sistema linear, caso ela exista, após um número finito de operações aritméticas.

Excelentes métodos numéricos para resolução de sistemas lineares, em um sistema de processador único, foram desenvolvidos e muitos códigos confiáveis e de alta qualidade estão disponíveis para diferentes casos de sistemas lineares (isto é, no *International Mathematical and Statistical Library* -IMSL). Por outro lado, os métodos para a resolução de sistemas de equações lineares em computadores paralelos MIMD, estão ainda no estágio conceitual, embora algumas idéias básicas já tenham surgido. O corrente estado da arte em álgebra linear numérica paralela, é bem descrito por Heller [Heller 78] e Sameh [Sameh 78].

4.3 - A solução de Sistemas Triangulares

4.3.1 - Sistema Triangular Superior

Uma matriz triangular é aquela na qual os elementos abaixo ou acima da diagonal principal são zeros. U é uma **matriz triangular superior** se $u_{ij} = 0$ para todo $i > j$. Consideremos o sistema triangular superior $U\mathbf{x} = \mathbf{y}$

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & \ddots & & \vdots \\ 0 & 0 & 0 & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & 0 & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} \qquad (4.4)$$

Vemos imediatamente que a solução da n-ésima equação é $x_n = \frac{y_n}{u_{nn}}$. A (n–1)-ésima equação tem somente uma incógnita, x_{n-1}, que é calculada por $x_{n-1} = \frac{\left(y_{n-1} - u_{n-1,n}x_n\right)}{u_{n-1,n-1}}$, e assim por diante, ou seja:

$$x_k = \frac{\left(y_k - \sum_{j=k-1}^{n} u_{kj}x_j\right)}{u_{kk}}. \qquad (4.5)$$

Supondo que a matriz dos coeficientes é armazenada em um espaço bidimensional, U, que o vetor dos termos independentes e o vetor solução são espaços unidimensionais, **y** e **x**, respectivamente, então o algoritmo 4.1 resolve o sistema triangular superior (4.4).

Algoritmo 4.1 - Substituição Regressiva - Dada uma matriz triangular superior U, de ordem n, e o vetor dos termos independentes **y**, este algoritmo determina o vetor **x**, de modo que $U\mathbf{x} = \mathbf{y}$.

1. $x_n = \frac{y_n}{u_{nn}}$
2. **for** $k = n-1$ **until** 1 **do**
3. soma = 0
4. **for** $j = k+1$ **until** n **do**
5. soma:= soma + $u_{kj}x_j$
6. **endfor**
7. $x_k = \frac{y_k - \text{soma}}{u_{kk}}$
8. **endfor** □

O somatório da linha 5 requer $(n - k)$ multiplicações. Como k varia de $(n - 1)$ a 1 tem-se

$$\sum_{k=1}^{n-1}(n-k) = 1+2+3+\cdots+n-1 = \frac{n(n-1)}{2},$$

ou seja, $O(n^2)$.

4.3.2 - Sistema Triangular Inferior

L é uma **matriz triangular inferior** se $l_{ij} = 0$ para todo $i < j$. Consideremos o sistema triangular inferior $L\mathbf{y} = \mathbf{b}$

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & \ddots & & \vdots \\ \vdots & \vdots & & l_{n-1,n-1} & 0 \\ l_{n1} & l_{n2} & \cdots & l_{n,n-1} & l_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \qquad (4.6)$$

Vemos, imediatamente, que a solução da 1ª equação é $y_1 = \frac{b_1}{l_{11}}$. A 2ª equação tem somente uma incógnita, y_2, que é calculada por $y_2 = \frac{(b_2 - l_{21}y_1)}{l_{22}}$, e assim por diante, ou seja:

$$y_k = \frac{\left(b_k - \sum_{j=1}^{k-1} l_{kj}y_j\right)}{l_{kk}} \qquad (4.7)$$

Supondo que a matriz dos coeficientes é armazenada em um espaço bidimensional, L, que o vetor dos termos independentes e o vetor solução são espaços unidimensionais, **b** e **y**, respectivamente, então o algoritmo 4.2 resolve o sistema triangular inferior (4.6).

Algoritmo 4.2 - Substituição Progressiva - Dada uma matriz triangular inferior, L, de ordem n e o vetor dos termos independentes **b**, este algoritmo determina o vetor **y**, de modo que $L\mathbf{y} = \mathbf{b}$.

1. $y_1 = \frac{b_1}{l_{11}}$
2. **for** $k = 2$ **until** n **do**
3. soma = 0
4. **for** $j = 1$ **until** $k-1$ **do**
5. soma:= soma + $l_{kj}y_j$
6. **endfor**
7. $y_k = \frac{b_k - \text{soma}}{l_{kk}}$
8. **endfor** □

O somatório na linha 5 requer $(k - 1)$ multiplicações e como a linha 2 é executada $(k - 1)$ vezes, o número total de multiplicações será

$$\sum_{k=2}^{n}(k-1) = 1+2+3+\cdots+n-1 = \frac{n(n-1)}{2},$$

ou seja, $O(n^2)$. O processo portanto, tem um custo operacional $O(n^2)$.

4.4 - Operações Elementares

Veremos a seguir alguns algoritmos para solução do sistema $A\mathbf{x} = \mathbf{b}$. A essência destes algoritmos para, os métodos diretos, são transformações sobre o sistema que deixa inalterada sua solução.

São operações elementares sobre uma matriz, $n \times n$, qualquer:

1. Multiplicar uma linha por um escalar não nulo.
2. Adicionar, a uma linha, uma outra linha da matriz.
3. Subtrair de uma linha, uma outra linha da matriz multiplicada por um escalar não nulo.

Os métodos diretos que estudaremos funcionam por ser fácil resolver um sistema $U\mathbf{x} = \mathbf{b}$ se U é triangular superior, isto é, em um tal sistema, podemos calcular por substituição regressiva, nesta ordem, as componentes $x_n, x_{n-1}, \ldots, x_1$. Tal método funciona, se $u_{ii} \neq 0, \forall\, i \leq n$. Como $\det(U) = \prod_{i=1}^{n} u_{ii}$, esta condição equivale a termos um sistema bem determinado. Estudaremos a seguir, como levar, através de operações elementares, um sistema qualquer à forma triangular.

4.5 - A Eliminação de Gauss

O método da Eliminação de Gauss triangulariza a matriz A dos coeficientes, ou seja, reduz a matriz A a uma matriz triangular superior, conforme o processo a seguir:

Consideremos um sistema de n equações lineares a n incógnitas, do tipo:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \quad\; \vdots \quad\; \cdots \quad\; \vdots \quad\; \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{4.8}$$

Podemos escrever o sistema (4.8) na forma matricial:

$$A\mathbf{x} = \mathbf{b}$$

em que,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{e} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Podemos associar ao sistema $A\mathbf{x} = \mathbf{b}$, a matriz ampliada, denotada por $[\, A \mid \mathbf{b} \,]$ onde

$$[\, A \mid \mathbf{b} \,] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{bmatrix}, \tag{4.9}$$

em que, $b_1 = a_{1(n+1)}$, $b_2 = a_{2(n+1)}$, ... , $b_n = a_{n(n+1)}$.

O primeiro passo é a eliminação de todos os elementos da primeira coluna de $[\, A \mid \mathbf{b} \,]$, exceto a_{11}. Vamos supor que o pivô $a_{11} \neq 0$. Definimos os multiplicadores por

$$m_{i1} = \frac{a_{i1}}{a_{11}}$$

para $i = 2, 3, \ldots, n$.

Multiplicamos a 1ª linha por m_{i1}, e a subtraímos da i-ésima linha, para $i = 2, 3, \ldots, n$, obtendo,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a_{22}-m_{21}a_{12} & \cdots & a_{2n}-m_{21}a_{1n} & b_2-m_{21}b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n2}-m_{n1}a_{12} & \cdots & a_{nn}-m_{n1}a_{1n} & b_n-m_{n1}b_1 \end{bmatrix}.$$

Reescrevendo a nova matriz como:

$$[A \mid \mathbf{b}]^{(1)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)} \end{bmatrix}, \tag{4.10}$$

onde o sobrescrito (1) indica que uma eliminação foi realizada com

$$a_{ij}^{(1)} = a_{ij} - m_{i1}a_{1j} \text{ para } i, j = 2, 3, \ldots, n$$

e

$$b_i^{(1)} = b_i - m_{i1}b_1 \text{ para } i = 2, 3, \ldots, n.$$

Admitamos, agora que o pivô $a_{22}^{(1)} \neq 0$. No segundo passo, eliminamos todos os elementos da 2ª coluna de $[A \mid \mathbf{b}]^{(1)}$, abaixo de $a_{22}^{(1)}$. Definimos $m_{i2} = \dfrac{a_{i2}^{(1)}}{a_{22}^{(1)}}$ para $i = 3, 4, \ldots, n$. Multiplicamos a 2ª linha por m_{i2}, e a subtraímos da i-ésima linha, para $i = 3, 4, \ldots, n$, obtendo:

$$[A \mid \mathbf{b}]^{(2)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} & b_3^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} & b_n^{(2)} \end{bmatrix},$$

em que,

$$a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i2}a_{2j}^{(1)} \text{ para } i, j = 3, 4, \ldots, n$$

e

$$b_i^{(2)} = b_i^{(1)} - m_{i2}b_2^{(1)} \text{ para } i = 3, 4, \ldots, n.$$

Repetindo este procedimento $(n-1)$ vezes, obtemos:

$$[A \mid \mathbf{b}]^{(n-1)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} & b_3^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{bmatrix} \tag{4.11}$$

contanto que no i-ésimo estágio os pivôs $a_{ii}^{(i-1)} \neq 0$, para $i = 1, 2, 3, \ldots, (n-1)$, onde:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - m_{ik} a_{kj}^{(k-1)},$$

$$b_i^{(k)} = b_i^{(k-1)} - m_{ik} b_k^{(k-1)},$$

$$m_{ik} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}},$$

para $k = 1, 2, 3, \ldots, (n-1)$ e $i, j = (k+1), (k+2), \ldots, n$.

De (4.11), temos:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}} \quad \text{se } a_{nn}^{(n-1)} \neq 0 \tag{4.12}$$

e

$$x_i = \frac{1}{a_{ii}^{(i-1)}} \left[b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j \right] \tag{4.13}$$

para $i = (n-1), (n-2), \ldots, 1$. Maiores detalhes, ver [Patel 94], [Forsythe 67] e [Albrecht 73].

O método de Eliminação de Gauss, descrito anteriormente, pode ser implementado utilizando o algoritmo 4.5.1. Ele encontra-se implementado no LINPACK [Dongarra 79].

Algoritmo 4.5.1 - Eliminação de Gauss - Este algoritmo triangulariza a matriz A do sistema linear $A\mathbf{x} = \mathbf{b}$, de ordem n, usando o método da Eliminação de Gauss. n é o número de equações e incógnitas; a_{ij} para $i, j = 1, 2, 3, \ldots, n$ são os elementos da matriz A; $b_1, b_2, \cdots, b_n$ são os elementos do vetor dos termos independentes **b**.

1. **for** $k = 1$ **until** $n - 1$ **do**
2. **for** $i = k + 1$ **until** n **do**
3. $m_{ik} := \dfrac{a_{ik}}{a_{kk}}$
4. **for** $j = k + 1$ **until** n **do**
5. $a_{ij} := a_{ij} - m_{ik} a_{kj}$
6. **endfor**
7. $b_i := b_i - m_{ik} b_k$
8. **endfor**
9. **endfor** □

Após a triangularização da matriz A em uma matriz triangular superior, obtemos o vetor **x**, a partir do sistema (4.11).

A idéia por traz da Eliminação de Gauss, é converter um dado sistema $A\mathbf{x} = \mathbf{b}$ em um sistema triangular equivalente, através de operações elementares do tipo 3. Tentaremos fazê-lo usando, nesta ordem, a primeira linha para anular os elementos abaixo da diagonal na primeira coluna, a segunda linha para anular os elementos abaixo da diagonal na segunda coluna, etc..

Dois sistemas lineares, $A\mathbf{x} = \mathbf{b}$ e $A'\mathbf{x} = \mathbf{b}'$, são equivalentes se qualquer solução de um é também solução do outro.

Em cada etapa da Eliminação, denominamos a linha que está sendo multiplicada e subtraída das demais, de **linha pivô**, seu elemento diagonal de **elemento pivô** e os fatores de multiplicação de **multiplicadores**. Cada linha é usada, por sua vez, para eliminar os coeficientes imediatamente abaixo do seu elemento diagonal. Vemos abaixo o estado do sistema, quando a k-ésima equação está prestes a ser usada no processo de eliminação

$$A^{(k)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & \cdots & \cdots & a_{2n}^{(2)} \\ & & \ddots & & & \vdots \\ & & & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}, \quad \mathbf{b}^{(k)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_k^{(k)} \\ \vdots \\ b_n^{(k)} \end{bmatrix},$$

A k-ésima linha é conhecida como linha pivô. O bloco de elementos sobrescritos k é chamado **bloco ativo**.

O objetivo do k-ésimo passo é usar múltiplos da linha pivô para eliminar os elementos $a_{k+1,k}^{(k)}, a_{k+2,k}^{(k)}, \ldots, a_{nk}^{(k)}$.

Para $k = 1, 2, \ldots, (n-1)$ resulta em um sistema triangular

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & a_{1n}^{1} \\ & a_{22}^{(2)} & \cdots & \cdots & a_{2n}^{(2)} \\ & & \ddots & & \vdots \\ & & & \ddots & \vdots \\ & & & & a_{nn}^{(n)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ \vdots \\ b_n^{(n)} \end{bmatrix}$$

equivalente ao sistema original, cuja solução pode ser determinada usando substituição regressiva. O sobrescrito na última linha é indicado por n para consistência notacional. Na verdade há somente $(n-1)$ passos de Eliminação.

4.5.1 - Pivoteamento

A Eliminação Gaussiana é computacionalmente possível a menos que em algum estágio encontremos um pivô nulo. Por exemplo, o processo aplicado a

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

irá fracassar. O sistema tem solução $\mathbf{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$, e fica claro que o pivô nulo não necessariamente significa que a solução do sistema não exista.

Assim, necessitamos modificar nosso algoritmo para resolver esta situação. Para executar a Eliminação, podemos trocar a linha que tem pivô nulo, com qualquer linha no bloco ativo que

não tem valor zero na coluna pivô. Se todos os elementos na coluna pivô do bloco ativo são zeros, então a matriz é singular, abortamos o processo e relatamos a falha ao usuário.

A mudança de linhas é conhecida como *pivoteamento* parcial, e seu uso não é restrito à situação mostrada acima. Do ponto de vista da estabilidade numérica, é desejável computar o *pivoteamento* parcial colocando como elemento pivô, em cada etapa, o elemento de maior módulo. Do contrário, pivôs de módulo menor que o maior módulo levam a multiplicadores maiores que a unidade, que por sua vez, tendem a aumentar os erros nos dados originais e os erros de arredondamento. Esta estratégia, garante que todos os multiplicadores $\left|m_{i1} = \frac{a_{i1}}{a_{11}}\right| \leq 1$ para $i = 2, 3, \ldots, n$, controlando a propagação de erros de arredondamento.

4.6 - A Fatoração LU

O objetivo desta seção é fatorar a matriz A do sistema linear

$$A\mathbf{x} = \mathbf{b}, \tag{4.14}$$

em que A é de ordem n, não singular e sem estrutura especial, num produto

$$A = LU, \tag{4.15}$$

em que L é uma matriz triangular inferior, com diagonal principal unitária e U é uma matriz triangular superior, segundo Ketter [Ketter 69] e Hopkins [Hopkins 88].

Admitindo que as matrizes L e U tenham sido encontradas na forma

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \quad \text{e } U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \tag{4.16}$$

tal que $A = LU$, o sistema linear, (4.14), se torna então

$$(LU)\mathbf{x} = \mathbf{b}. \tag{4.17}$$

Estabelecendo que

$$U\mathbf{x} = \mathbf{y} \tag{4.18}$$

o sistema (4.17) se torna

$$L\mathbf{y} = \mathbf{b}. \tag{4.19}$$

A seguir, será analisado como obter os elementos das matrizes L e U, a partir da matriz A.

Seja o sistema linear $A\mathbf{x} = \mathbf{b}$, 4×4, onde a matriz A é dada por

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 1 & 5 \\ -1 & 1 & -5 & 3 \\ 3 & 1 & 7 & -2 \end{bmatrix} \text{ e } \mathbf{b} = \begin{bmatrix} 10 \\ 31 \\ -2 \\ 18 \end{bmatrix}.$$

Eliminando x_1 das três últimas equações resulta em

$$A_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 3 \\ 0 & 2 & -4 & 4 \\ 0 & -2 & 4 & -5 \end{bmatrix}$$

Com algum esforço verificamos que para recuperar A a partir de A_1, precisamos multiplicar esta por alguma matriz e vice-versa; para se chegar de A para A_1 é necessário multiplicar A por alguma matriz. A matriz que multiplica A para transformá-la em A_1 deve anular todos os elementos abaixo da diagonal da primeira coluna de A. A matriz

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix},$$

em que

$$m_{21} = -\frac{a_{21}}{a_{11}} = -\frac{2}{1} = -2$$

$$m_{31} = -\frac{a_{31}}{a_{11}} = -\frac{-1}{1} = 1$$

$$m_{41} = -\frac{a_{41}}{a_{11}} = -\frac{3}{1} = -3$$

executa a transformação. Assim, $M_1 A = A_1$. Para desfazer a transformação, fica claro que basta calcular $A = M_1^{-1} A_1$. Resta saber se a obtenção de M_1^{-1} não é difícil. Felizmente, M_1 tem uma bela propriedade. Ela é da forma

$$M_1 = I + \mathrm{m}_1 \mathrm{e}_1^{\mathrm{T}}, \tag{4.20}$$

onde $\mathbf{m}_1 = \begin{bmatrix} 0 & -2 & 1 & -3 \end{bmatrix}^{\mathrm{T}}$ e $\mathbf{e}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$ como se pode verificar executando o produto e somando com I. A propriedade é

$$M_1^{-1} = I - \mathrm{m}_1 \mathrm{e}_1^{\mathrm{T}}, \tag{4.21}$$

pois

$$(I + \mathbf{m}_1 \mathbf{e}_1^{\mathrm{T}})\left(I - \mathbf{m}_1 \mathbf{e}_1^{\mathrm{T}}\right) = I - \mathbf{m}_1 \mathbf{e}_1^{\mathrm{T}} + \mathbf{m}_1 \mathbf{e}_1^{\mathrm{T}} - (\mathbf{m}_1 \mathbf{e}_1^{\mathrm{T}})(\mathbf{m}_1 \mathbf{e}_1^{\mathrm{T}}) = I - \mathbf{m}_1 (\mathbf{e}_1^{\mathrm{T}} \mathbf{m}_1) \mathbf{e}_1^{\mathrm{T}} = I$$

uma vez que $\mathbf{e}_1^T \mathbf{m}_1 = 0$ porque $\mathbf{e}_1^T$ só tem o primeiro componente não nulo e $\mathbf{m}_1$ tem o primeiro nulo. Isso mostra que o segundo membro de (4.21) é, de fato, a inversa de M_1. Em termos práticos, para obter M_1^{-1} basta trocar os sinais dos elementos abaixo do elemento diagonal na coluna 1 de M_1. Efetuando o produto $M_1^{-1} A_1$, recuperamos A.

Agora, tendo A_1, que corresponde ao primeiro membro do sistema, do qual eliminamos x_1 das três últimas equações, prosseguimos para obter A_2, que tem os elementos, abaixo da diagonal, nas colunas 1 e 2, nulos. Como em A_1 os elementos abaixo da diagonal são nulos na coluna 1, precisamos preservá-los e anular os elementos abaixo da diagonal, na coluna 2. Seja M_2 a matriz tal que $M_2 A_1$ fique com elementos abaixo da diagonal nulos na coluna 2 e mantenha a coluna 1 de A_1, inalterada.

Antes de construir M_2, vamos generalizar a relação (4.20), que fornece M_1, procurando encontrar uma matriz que anule os elementos abaixo da diagonal, na coluna k. Para isso, basta escolher um vetor $\mathbf{m}_k$ da forma

$$\mathbf{m}_k = \begin{bmatrix} 0 & 0 & \dots & 0 & m_{k+1} & m_{k+2} & \dots \end{bmatrix}^T$$

e multiplicar por $\mathbf{e}_k^T = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}$, que tem o k-ésimo elemento igual a 1, resultando em

$$M_k = I + \mathbf{m}_k \mathbf{e}_k^T \tag{4.22}$$

Assim, para executar a nossa operação na coluna 2 usamos

$$\mathbf{m}_2 = \begin{bmatrix} 0 & 0 & -\dfrac{a_{32'}}{a_{22'}} & -\dfrac{a_{42'}}{a_{22'}} \end{bmatrix}^T$$

onde $a_{ij'}$, denota o elemento (i, j) de A_1. Obtemos

$$\mathbf{m}_2 = \begin{bmatrix} 0 & 0 & -2 & 2 \end{bmatrix}^T$$

e

$$M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -2 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix}$$

Efetuando o produto $M_2 A_1$, resulta

$$A_2 = M_2 A_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 3 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & 2 & 1 \end{bmatrix}.$$

Finalmente, para anular o elemento abaixo da diagonal na coluna 3, encontramos

$$\mathbf{m}_3 = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T.$$

Multiplicando por $\mathbf{e}_3^T = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$ obtemos

$$M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

$$A_3 = M_3 A_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 3 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

Observamos que em cada passo k podemos desfazer a transformação para recuperar a matriz do passo anterior, isto é

$$A_k = M_k A_{k-1} \ , \qquad A_{k-1} = M_k^{-1} A_k \, .$$

Convencionando que $A_0 = A$, obtemos

$$A = A_0 = M_1^{-1} A_1 = M_1^{-1} M_2^{-1} A_2 = \cdots = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1} A_{n-1} \qquad (4.23)$$

se $A \in \Re^{n \times n}$. Observe que A_{n-1} é triangular superior, razão pela qual passamos a chamá-la por U e por L o produto das inversas de M_k

$$L = M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1}. \qquad (4.24)$$

No nosso exemplo obtemos

$$L = M_1^{-1} M_2^{-1} M_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 3 & -2 & -1 & 1 \end{bmatrix}.$$

Observamos que as colunas de L são constituídas pelas colunas de M_1^{-1} (coluna 1), M_2^{-1} (coluna 2) e M_3^{-1} (coluna 3), ou seja o produto em (4.24) preserva a coluna k de M_k^{-1}. Isso significa que, à medida que efetuamos a transformação, podemos construir L, coluna a coluna, pois a coluna k de L será igual à coluna k de M_k^{-1}, tendo a forma

$$\begin{bmatrix} 0 & \cdots & 1 & -m_{(k+1)k} & -m_{(k+2)k} & \cdots & -m_{nk} \end{bmatrix}^{\mathrm{T}}$$

e

$$-m_{(k+r)k} = \frac{a_{(k+r)k}}{a_{kk}}, \quad r = 1, 2, 3, \ldots, n-k.$$

Efetuando agora o produto de L por $A_3 = U$ recuperamos a matriz A, original, ou seja

$$A = LU.$$

Substituindo A, no sistema $A\mathbf{x} = \mathbf{b}$ por LU e chamando

$$U\mathbf{x} = \mathbf{y} \qquad (4.25a)$$

resulta

$$L\mathbf{y} = \mathbf{b}. \qquad (4.25b)$$

Resolvendo (4.25b) e a seguir (4.25a) obtém-se a solução $\mathbf{x}$ desejada. Para o exemplo considerado, temos

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 3 & -2 & -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 31 \\ -2 \\ 18 \end{bmatrix}$$

desenvolvendo o produto do primeiro membro obtém-se

$$\begin{aligned} y_1 &= 10 \\ 2y_1 + y_2 &= 31 \\ -y_1 + 2y_2 + y_3 &= -2 \\ 3y_1 - 2y_2 - y_3 + y_4 &= 18, \end{aligned}$$

cuja solução é: $y_1 = 10$, $y_2 = 11$, $y_3 = -14$, $y_4 = -4$. Agora resolvemos $U\mathbf{x} = \mathbf{y}$.

Resumindo, para se resolver o sistema $A\mathbf{x} = \mathbf{b}$

(a) fatoramos a matriz A na forma $A = LU$,

(b) resolvemos o sistema triangular inferior $L\mathbf{y} = \mathbf{b}$, chamado substituição progressiva e

(c) resolvemos o sistema triangular superior $U\mathbf{x} = \mathbf{y}$, chamado substituição regressiva.

A questão levantada da equivalência de dois sistemas, pode, agora, ser respondida de forma trivial, porque $A\mathbf{x} = (LU)\mathbf{x}$ nos diz que o sistema do primeiro membro é igual ao do segundo membro e, portanto, a solução do sistema do segundo membro deve ser também a solução do primeiro membro.

Como subproduto da Fatoração LU podemos calcular o determinante da matriz dos coeficientes A, através de

$$A = LU, \ \det A = \det(LU) = \det L \times \det U.$$

Como L é triangular inferior e os elementos de sua diagonal são todos iguais a 1, det L é dado pelo produto dos elementos da diagonal principal, det $L = 1$. Como U é triangular superior, o seu determinante será igual ao produto dos elementos da diagonal principal

$$\det U = u_{11} \times u_{22} \times u_{33} \times \cdots \times u_{nn} = \prod_{i=1}^{n} u_{ii} .$$

Como det $L = 1$, temos que det A = det U. O produto dos elementos da diagonal pode causar *overflow* ou *underflow* mesmo que det A possa ser representado internamente, se valores elevados ou pequenos de u_{ii} ocorrerem no início do produto.

Outro subproduto da Fatoração LU é a possibilidade de obter a inversa da matriz A. Para obter A^{-1}, resolvemos n sistemas lineares $A\mathbf{x} = \mathbf{b}^{(k)}$, $k = 1, 2, \ldots, n$, adotando $\mathbf{b}^{(k)} = \mathbf{e}_k = \begin{bmatrix} 0 & 0 & \ldots & 1 & \ldots & 0 \end{bmatrix}^T$, com o componente k de $\mathbf{e}_k$ igual a 1. Assim, cada sistema permite obter uma coluna de A^{-1}. Observamos, contudo, que em computação numérica, raramente precisamos de A^{-1} [Hattori 96].

Algoritmo 4.6.1 - Fatoração LU - Dada uma matriz A, $n \times n$, constrói as matrizes L e U tal que $A = LU$, armazenando U, uma matriz triangular superior como triângulo superior de A e L, uma matriz triangular inferior, no triângulo inferior de A, sem armazenar os elementos da diagonal por serem todos iguais a 1. A matriz A original será destruída.

1. **for** $k = 1$ **until** $n - 1$ **do**
2. **for** $i = k + 1$ **until** n **do**
3. $a_{ik} := \dfrac{a_{ik}}{a_{kk}}$
4. **for** $j = k + 1$ **until** n **do**
5. $a_{ij} := a_{ij} - a_{ik} a_{kj}$
6. **endfor**
7. **endfor**
8. **endfor** □

Claramente o algoritmo termina em um número finito de passos. Isso permite calcular o número de operações aritméticas necessárias para fatorar uma matriz $n \times n$. Para obter esse número observamos que

(a) a linha 1 será executada $(n - 1)$ vezes;

(b) para cada k a repetição controlada pela linha 2 será executada $(n - k)$ vezes;

(c) e para cada passo i da linha 2 a linha 1 será executada $(n - k)$ vezes e a linha 3, uma vez.

Desta observação, resulta que o número de multiplicações (ou divisões) será calculado somando-se o número de operações (c) para cada k

$$\sum_{k=1}^{n} (n-k)(n-k+1) = \sum_{k=1}^{n-1} (n-k)^2 + \sum_{k=1}^{n-1} (n-k)$$

$$= n(n-1)\frac{(2n-1)}{6} + n\frac{(n-1)}{2}$$

$$= \frac{n^3}{3} + \frac{n}{3}.$$

Para valores elevados de n, o termo dominante será o primeiro e, assim, dizemos que a complexidade da Fatoração LU é $O(n^3)$.

Tendo a Fatoração LU da matriz dos coeficientes de um sistema linear $A\mathbf{x} = \mathbf{b}$, podemos resolvê-lo resolvendo os sistemas triangulares $L\mathbf{y} = \mathbf{b}$ e $U\mathbf{x} = \mathbf{y}$.

4.6.1 - A Fatoração LU com Pivoteamento Parcial

Vimos que, na Fatoração LU, a matriz M_k, que no passo k anula os elementos abaixo da diagonal na coluna k, tem elementos m_{ik}, obtidos pela relação

$$m_{ik} = -\frac{a_{ik}}{a_{kk}}, \tag{4.26}$$

em que o elemento a_{kk} é chamado o pivô do passo k. Se, para algum k, o pivô se tornar nulo, a divisão em (4.26) se torna impossível e, em conseqüência, impede o prosseguimento da Fatoração LU. Nem sempre um pivô nulo impede o prosseguimento da fatoração, conforme veremos no exemplo abaixo, em que uma simples permutação de linhas evita um pivô nulo.

Considere o exemplo

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 2 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}.$$

Pode-se verificar que a matriz A é não singular ($\det(A) = -1$). Executando o primeiro passo da Fatoração LU, resulta

$$M_1 A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Para prosseguir precisamos permutar as linhas 2 e 3 de $M_1 A$, pré-multiplicando essa matriz por uma matriz de permutação

$$P_2 M_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

Com isso, evitamos o pivô nulo e, na realidade, obtivemos o fator U desejado. Resta verificar a influência da matriz de permutação. O fator L será

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

Efetuando o produto LU, obtemos

$$LU = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 1 & 2 \end{bmatrix} = A'$$

a matriz A, com as linhas 2 e 3 permutadas. Na realidade, $LU = PA$. Portanto, a relação entre A e os fatores L e U passa a ser

$$PA = LU \text{ ou } A = P^{-1}LU = P^{T}LU,$$

uma vez que a matriz de permutação é ortogonal, isto é, $P^{-1} = P^{T}$. [Strang 79]

Se tivermos um sistema linear $A\mathbf{x} = \mathbf{b}$, resolvemos $L\mathbf{y} = P\mathbf{b}$ e $U\mathbf{x} = \mathbf{y}$.

Vamos agora perturbar a matriz A, somando 10^{-4} a a_{22}. Tem-se:

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1{,}0001 & 2 \\ 1 & 2 & 2 \end{bmatrix}, \quad A_1 = M_1A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0{,}0001 & 1 \\ 0 & 1 & 1 \end{bmatrix},$$

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -10000 & 1 \end{bmatrix}, \; A_2 = M_2A_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0{,}0001 & 1 \\ 0 & 0 & -9999 \end{bmatrix} = U \text{ e } L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 10000 & 1 \end{bmatrix}$$

Seja $\mathbf{b} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}^{T}$, no sistema linear $A\mathbf{x} = \mathbf{b}$, com A dada. Então, a solução de

$$L\mathbf{y} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 10000 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

é $\mathbf{y} = \begin{bmatrix} 1 & 1 & -10000 \end{bmatrix}^{T}$, a solução de $U\mathbf{x} = \mathbf{y}$, fornece, com 5 dígitos $\mathbf{x} = \begin{bmatrix} 1{,}001 & -1{,}0000 & 0{,}99990 \end{bmatrix}^{T}$, enquanto que a solução correta é $\mathbf{x} = \begin{bmatrix} 1{,}0000 & -1{,}0000 & 1{,}0001 \end{bmatrix}^{T}$.

Se no segundo passo permutarmos as linhas 2 e 3 obtemos

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0{,}0001 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0{,}99990 \end{bmatrix}.$$

Então, a solução de $L\mathbf{y} = P_{23}\mathbf{b}$ é $\mathbf{y} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^{T}$ e de $U\mathbf{x} = \mathbf{y}$ é $\mathbf{x} = \begin{bmatrix} 1{,}0000 & -1{,}0000 & 1{,}0001 \end{bmatrix}^{T}$, correto com 5 dígitos. Devemos evitar pivô nulo e pivô "pequeno" [Strang 79]. Para chegar a essa conclusão calculamos, usando a fórmula de propagação de erros, o erro da expressão

$$a_{ij} := a_{ij} - l_{ik} a_{kj}$$

supondo que a_{ij}, a_{kj} e a_{kk} estão sujeitos a erro ε,

$$|\text{Erro}| \leq \varepsilon \left[1 + |a_{kj}| \frac{|a_{kk} + a_{ik}|}{a_{kk}^2} + \left| \frac{a_{ik}}{a_{kk}} \right| \right]. \qquad (4.27)$$

Verifica-se que a única maneira de minimizar o erro, é escolher o maior $|a_{kk}|$ possível.

O método que apresentamos escolhe, em cada passo k, o maior elemento, em valor absoluto, na coluna k da matriz A, a partir da diagonal para baixo. A estratégia é conhecida como *pivoteamento* parcial. Podemos utilizar também o *pivoteamento* total, em que procuramos, em cada passo k, o maior elemento em valor absoluto na submatriz obtida excluindo-se as $(k-1)$ primeiras linhas e as $(k-1)$ primeiras colunas da matriz obtida no passo anterior. A desvantagem do *pivoteamento* total é o custo da pesquisa do maior elemento em valor absoluto em cada passo: precisamos examinar $(n-k+1)(n-k+1)$ elementos em cada passo. Como a fatoração requer $(n-1)$ passos, o total de comparações é dado por

$$\sum_{k=1}^{n-1} (n-k+1) = O(n^3).$$

A vantagem do *pivoteamento* total é de assegurar a estabilidade do método. No *pivoteamento* parcial o número de comparações no passo k é $(n-k+1)$ e o total de comparações será

$$\sum_{k=1}^{n-1} (n-k+1) = O(n^2),$$

tendo como desvantagem a segurança parcial da estabilidade do método. Na prática, usa-se *pivoteamento* parcial porque é muito difícil encontrar matrizes em que o *pivoteamento* parcial seja instável [Stewart 73].

Vamos, agora, reapresentar o algoritmo 4.6.1 com *pivoteamento* parcial.

Algoritmo 4.6.2 - Fatoração LU com *pivoteamento* parcial.

1. **for** $k = 1$ **until** $n-1$ **do**
   /* encontra o pivô pesquisando a coluna k */
2. Encontre r tal que $|a_{rk}| = \max_i |a_{ik}|$, com $i = k, k+1, \ldots, n$;
3. **if** $a_{ik} = 0$ **then** a matriz A é singular, **stop**
4. Permute as linha r e k
5. **for** $i = k+1$ **until** n **do**
6. $a_{ik} := \dfrac{a_{ik}}{a_{kk}}$
7. **for** $j = k+1$ **until** n **do**
8. $a_{ij} := a_{ij} - a_{ik} a_{kj}$
9. **endfor**
10. **endfor**
11. **endfor** □

Observe que as permutações de linhas devem ser registradas, de alguma forma, porque precisamos delas ao resolver o sistema triangular $L\mathbf{y} = P\mathbf{b}$, onde P é a matriz de permutação. No algoritmo, deixamos esse detalhe omisso. Agora discutiremos a forma de guardar P para posterior utilização.

Se P é a matriz global de permutação, usada no *pivoteamento* parcial, obtemos a fatoração de PA e não de A, de modo que $PA = LU$. Para resolver $A\mathbf{x} = \mathbf{b}$, devemos não só substituir a matriz A pelos fatores L e U, para economizar espaço de memória, como escriturar as permutações de linhas ditada pelo *pivoteamento*. Por questões de eficiência, devemos também evitar a operação de permutação de linhas, porque cada permutação envolve $3(n - k)$ operações de atribuição no passo k ($k = 1, 2, \ldots, n - 1$). Essencialmente, a linha 4 do algoritmo 4.6.2 é o ponto sutil da implementação.

A idéia básica da implementação é de usar um vetor auxiliar **v**, cujos componentes, inteiros, fornecem a seqüência das linhas da matriz A que vão ser processadas. Inicialmente v_1, $v_2, \ldots, v_n$ rotulam as linhas 1, 2, ... , n da matriz A, ou seja, $v_i = i$. No passo k da fatoração, se houver permutação das linhas k e r, o componente v_k terá valor r e o v_r, o valor k. Assim, ao final da fatoração, $v_1, v_2, \ldots, v_n$ fornecerá a seqüência das linhas que foram usadas como pivô. As operações de transformação serão executadas nas colunas $(k + 1), \ldots, n$ e nas linhas $v_{k+1}, \ldots, v_n$. O algoritmo 4.6.2, então toma a forma

Algoritmo 4.6.2a [Forsythe 67] - Fatoração LU com *pivoteamento* parcial.

1. **for** $i = 1$ **until** n **do** $v_i := i$
2. **for** $k = 1$ **until** $n - 1$ **do**
   /* encontra o pivô pesquisando a coluna k */
3. Encontre r tal que $|a_{rk}| = \max_i |a_{ik}|$, com $i = v_k, v_{k+1}, \ldots, v_n$;
4. **if** $a_{rk} = 0$ **then** a matriz A é singular; **stop**
5. Permute as linhas v_k e v_r e defina $p = v_k$
6. **for** $i = v_{k+1}$ **until** v_n **do**
7. $a_{ik} := \dfrac{a_{ik}}{a_{pk}}$
8. **for** $j = k + 1$ **until** n **do**
9. $a_{ij} := a_{ij} - a_{ik}a_{pj}$
10. **endfor**
11. **endfor**
12. **endfor** □

4.7 - Métodos Iterativos

O termo método iterativo refere-se a uma variedade de técnicas que usam sucessivas aproximações para obter soluções, a cada passo mais precisas para um sistema linear. Nesta seção, vamos abordar dois tipos de métodos iterativos: estacionários e não estacionários. Métodos estacionários são antigos, simples de compreender e implementar, mas geralmente não eficazes. Os métodos iterativos estacionários são aqueles que podem ser expressos na forma

$$\mathbf{x}^{(k)} = B\,\mathbf{x}^{(k-1)} + c$$

(onde nem B nem c dependem da quantidade de iterações k). Os principais métodos iterativos estacionários são: o método iterativo de Gauss-Jacobi, o método iterativo de Gauss-Seidel, o método iterativo de Sobre-Relaxação Sucessiva (SOR) e o método iterativo de Sobre-Relaxação Sucessiva Simétrica (SSOR).

Os métodos não-estacionários foram desenvolvidos recentemente; sua análise é geralmente difícil, mas são altamente eficazes. Os métodos não-estacionários são baseados na idéia de seqüências de vetores ortogonais. A diferença entre os métodos não-estacionários e os métodos estacionários, é que a computação envolve informações que mudam a cada iteração. Tipicamente, constantes são computadas através de produtos internos de resíduos, ou outros vetores que surgem do método iterativo. Entre os métodos não-estacionários, encontra-se o método iterativo dos Gradientes Conjugados [Barrett 94].

No caso de sistemas esparsos de grande porte, usualmente são utilizados métodos iterativos que não destroem a estrutura da matriz dos coeficientes, ao contrário dos métodos diretos que introduzem elementos não nulos nas posições que inicialmente continham elementos nulos. Dizemos que um sistema linear é esparso quando a matriz dos coeficientes possui uma grande percentagem de elementos nulos. Para tais sistemas, a resolução através do método da Eliminação de Gauss não é aconselhável, por que este método não preserva a esparsidade, ou seja, durante o processo de eliminação muitos elementos nulos poderão se tornar não nulos.

Métodos iterativos são mais adequados que os métodos diretos, na solução de sistemas, quando se procura minimizar o tempo de processamento e o espaço de armazenamento requerido no computador [Varga 62], devido à simplicidade e uniformidade das operações realizadas nas iterações. Contudo, deve-se chamar a atenção para o fato de que não há regras rígidas para escolher entre um método direto e um iterativo [Ralston 65]. Na verdade, existem controvérsias quanto a essas regras.

Existem alguns métodos que fazem uso apenas dos elementos da matriz A original. Estes métodos consistem de algoritmos simples para converter qualquer vetor $\mathbf{x}^{(k)}$ em outro , $\mathbf{x}^{(k+1)}$,

que depende de $\mathbf{x}^{(k)}$, A e $\mathbf{b}$, e preservam a esparsidade de A, pois os elementos desta matriz não são alterados. Eles pertencem à classe dos métodos iterativos para resolver sistemas lineares.

Definição 4.7.1 - *Diz-se que o método é iterativo quando partindo de uma aproximação inicial, é possível se chegar a aproximações mais precisas que dependem, sempre, de valores anteriormente calculados.*□

Os métodos iterativos gozam, ainda de, uma outra vantagem sobre o método da Eliminação de Gauss, que é o fato de apresentarem uma relativa insensibilidade ao crescimento dos erros de arredondamento.

A idéia central dos métodos iterativos é generalizar o método iterativo linear, utilizado na busca de raízes de uma equação.

Seja o sistema linear $A\mathbf{x} = \mathbf{b}$, onde:

A : matriz dos coeficientes, $n \times n$;

$\mathbf{x}$: vetor das variáveis, $n \times 1$;

$\mathbf{b}$: vetor dos termos constantes, $n \times 1$.

Este sistema é convertido, de alguma forma, num sistema do tipo $\mathbf{x} = C\mathbf{x} + \mathbf{g} = \rho(\mathbf{x})$ onde C é uma matriz $n \times n$ e $\mathbf{g}$ um vetor $n \times 1$. Observamos que $\rho(\mathbf{x}) = C\mathbf{x} + \mathbf{g}$ é uma função de iteração, dada na forma matricial.

É, então, proposto o esquema iterativo:

Partimos de $\mathbf{x}^{(0)}$ (vetor aproximação inicial) e então construímos, consecutivamente, os vetores

$\mathbf{x}^{(1)} = C\,\mathbf{x}^{(0)} + \mathbf{g} = \rho(\mathbf{x}^{(0)})$ (primeira aproximação)

$\mathbf{x}^{(2)} = C\,\mathbf{x}^{(1)} + \mathbf{g} = \rho(\mathbf{x}^{(1)})$ (segunda aproximação) etc.

De um modo geral, a aproximação de ordem (k+1) é calculada da fórmula $\mathbf{x}^{(k+1)} = C\,\mathbf{x}^{(k)} + \mathbf{g}$, ou seja,

$\mathbf{x}^{(k+1)} = \rho(\mathbf{x}^{(k)})$, k = 1, 2, 3, ...

É importante observar que, se a seqüência de aproximações $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(k)}, \ldots$ é tal que, $\lim_{k \to \infty} \mathbf{x}^{(k)} = \alpha$ então, $\alpha = C\alpha + \mathbf{g}$, ou seja, α é a solução do sistema linear $A\mathbf{x} = \mathbf{b}$.

4.7.1 - Teste de Parada

Um método iterativo produz uma seqüência $\left\{\mathbf{x}^{(i)}\right\}$ de vetores convergindo para o vetor $\mathbf{x}$, satisfazendo um sistema $A\mathbf{x} = \mathbf{b}$, $n \times n$. Para ser eficaz, um método iterativo precisa decidir quando parar. Um bom critério deve:

1. identificar quando o erro $\mathbf{e}^{(i)} \equiv \mathbf{x}^{(i)} - \mathbf{x}$ é pequeno o bastante para parar,
2. parar, se o erro não mais decresce ou decresce lentamente, e
3. limitar o número máximo de iterações.

O usuário precisa fornecer as quantidades *maxit* e *stop_tol*. O inteiro *maxit* é o número máximo de iterações que o algoritmo permitirá executar. O número real *stop_tol* mede quão pequeno o usuário quer o resíduo, $\mathbf{r}^{(i)} = A\mathbf{x}^{(i)} - \mathbf{b}$, da última solução $\mathbf{x}^{(i)}$. O usuário deverá escolher *stop_tol* menor que 1 e maior que o ε da máquina (em uma máquina com o padrão de aritmética de ponto flutuante IEEE, $\varepsilon = 2^{-24} \approx 10^{-7}$ em precisão simples, e $\varepsilon = 2^{-53} \approx 10^{-16}$ em precisão dupla)

O processo iterativo é repetido, até que o vetor $\mathbf{x}^{(k)}$ esteja suficientemente próximo do vetor $\mathbf{x}^{(k-1)}$.

Medimos a distância entre $\mathbf{x}^{(k)}$ e $\mathbf{x}^{(k-1)}$, por $M^{(k)} = \max_{1 \le i \le n} \left|\mathbf{x}_i^{(k)} - \mathbf{x}_i^{(k-1)}\right|$.

Assim, dada uma precisão ε, o vetor $\mathbf{x}^{(k)}$ será escolhido como $\mathbf{x}$, solução aproximada da solução exata, se $M^{(k)} < \varepsilon$.

Da mesma maneira que no teste de parada dos métodos iterativos para zeros de funções, podemos efetuar, aqui, o teste do erro relativo:

$$M_R^{(k)} = \frac{M^{(k)}}{\max_{1 \le i \le n} \left|\mathbf{x}_i^{(k)}\right|}. \qquad (4.7.1)$$

Computacionalmente, usamos também como teste de parada o número máximo de iterações.

4.7.2 - Método Iterativo de Gauss-Jacobi

A forma como o método de Gauss-Jacobi transforma o sistema linear $A\mathbf{x} = \mathbf{b}$ em $\mathbf{x} = C\mathbf{x} + \mathbf{g}$ é a seguinte:

Tomemos o sistema linear

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{4.7.2}$$

e supondo $a_{ii} \neq 0$, $i = 1, 2, \ldots, n$, isolamos o vetor $\mathbf{x}$ mediante a separação pela diagonal, assim:

$$\left[\begin{array}{l} x_1 = \dfrac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\ x_2 = \dfrac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \\ \qquad\qquad \vdots \\ x_n = \dfrac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1}) \end{array}\right. \tag{4.7.3}$$

Desta forma temos $\mathbf{x} = C\mathbf{x} + \mathbf{g}$ onde

$$C = \begin{bmatrix} 0 & -\dfrac{a_{12}}{a_{11}} & -\dfrac{a_{13}}{a_{11}} & \cdots & -\dfrac{a_{1n}}{a_{11}} \\ -\dfrac{a_{21}}{a_{22}} & 0 & -\dfrac{a_{23}}{a_{22}} & \cdots & -\dfrac{a_{2n}}{a_{22}} \\ -\dfrac{a_{31}}{a_{33}} & -\dfrac{a_{32}}{a_{33}} & 0 & \cdots & -\dfrac{a_{3n}}{a_{33}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\dfrac{a_{n1}}{a_{nn}} & -\dfrac{a_{n2}}{a_{nn}} & -\dfrac{a_{n3}}{a_{nn}} & \cdots & 0 \end{bmatrix}, \qquad \mathrm{g} = \begin{bmatrix} \dfrac{b_1}{a_{11}} \\ \dfrac{b_2}{a_{22}} \\ \vdots \\ \dfrac{b_n}{a_{nn}} \end{bmatrix} \text{ e } \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

O método de Gauss-Jacobi consiste em, dado $\mathbf{x}^{(0)}$, um vetor aproximação inicial, obter $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(k)}, \ldots$ através da relação recursiva $\mathbf{x}^{(k+1)} = C\mathbf{x}^{(k)} + \mathbf{g}$:

$$\left[\begin{array}{l} x_1^{(k+1)} = \dfrac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)}) \\ x_2^{(k+1)} = \dfrac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}) \\ \qquad\qquad \vdots \\ x_n^{(k+1)} = \dfrac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2^{(k)} - \cdots - a_{n,n-1}x_{n-1}^{(k)}) \end{array}\right. \tag{4.7.4}$$

$$x_i^{(k+1)} = \frac{1}{a_{11}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)}\right) \quad \text{para } i = 1, 2, 3, \ldots, n \quad \text{e } k = 0, 1, 2, \ldots. \tag{4.7.5}$$

Definido este método iterativo básico, pode-se concluir, pela construção da matriz C, que uma condição necessária para que o método seja aplicável, é que todos os elementos da diagonal principal da matriz dos coeficientes A sejam diferentes de zero.

O valor de $\mathbf{x}^{(0)}$ é arbitrário pois veremos mais adiante, que a convergência ou não de um método iterativo, para solução de um sistema linear de equações, é independente da aproximação inicial escolhida.

Daremos aqui um teorema que estabelece uma condição suficiente para a convergência do método iterativo de Gauss-Jacobi.

Teorema 4.7.1: Seja o sistema $A\mathbf{x} = \mathbf{b}$ e

$$\alpha_k = \frac{(\sum_{j=1, j\neq k}^{n} |a_{kj}|)}{a_{kk}}.$$

Se $\alpha = \max_{1\leq k\leq n} \alpha_k < 1$ então o método de Gauss-Jacobi gera uma seqüência $\{\mathbf{x}^{(k)}\}$ convergente para a solução do sistema dado, independentemente da escolha da aproximação inicial, $\mathbf{x}^0$. □

A demonstração deste teorema pode ser encontrada na referencia [Demidovich 73].

Corolário 4.7.1 (Critério das linhas): É condição suficiente para que a iteração definida em (4.7.5) convirja, que

$$|a_{ii}| > \sum_{j=1, j\neq i} |a_{ij}|, \qquad \text{para } i = 1, 2, 3, \ldots, n. \square \tag{4.7.6}$$

Observação: A matriz que satisfaz as hipóteses do corolário 4.7.1 é chamada *estritamente diagonal dominante*.

Na prática, é usado o critério de suficiência de convergência expresso no corolário 4.7.1, para o método de Gauss-Jacobi. Basta que o sistema satisfaça o critério para se ter a convergência garantida, independentemente da escolha do vetor aproximação inicial. Para maiores detalhes veja o livro de David Young [Young 71].

Algoritmo 4.7.1 Este algoritmo resolve $A\mathbf{x} = \mathbf{b}$ usando o método de Gauss-Jacobi dado pela equação 4.7.5. n é número de equações e incógnitas; a_{ij} para i , j = 1, 2, 3, ... , n são os elementos da matriz A; b_1, b_2, $\cdots$, b_n são elementos do vetor dos termos independentes **b**; x_1, x_2, ... , x_n são os elementos do vetor aproximação inicial; TOL é a tolerância e MAXIT é o número máximo de iterações.

1. **Begin**
2. ite : = 0
3. **while** ite ≤ MAXIT **do**
4. ite : = ite + 1
5. **for** i = 1 **until** n **do**
6. $xold_i = x_i$
7. **endfor**
8. **for** i = 1 **until** n **do**
9. sum : = 0
10. **for** j = 1 **until** n **do**
11. **if** $j \neq i$ **then**
12. sum : = sum + $a_{ij} \times xold_j$
13. **endif**
14. **endfor**
15. $x_i = \dfrac{(b_i - \text{sum})}{a_{ii}}$
16. **endfor**
17. maior : = $|x_1 - xold_1|$
18. **for** i = 2 **until** n **do**
19. aux : = $|x_i - xold_i|$
20. **if** aux ≤ maior **then**
21. maior : = aux
22. **endif**
23. **endfor**
24. **if** maior ≤ TOL **then**
25. **exit**
26. **endif**
27. **endwhile**
28. **if** maior ≤ TOL **then**
29. **print** ite, x
30. **stop**
31. **else**
32. **print** Algoritmo falha, solução não converge
33. **stop**
34. **endif**
35. **end**□

4.7.3 - Método iterativo dos Gradientes Conjugados

No início da década de 50, Hestenes e Stiefel [Hestenes 52] apresentaram um novo método iterativo para resolução de sistemas de equações lineares. Este novo método ficou conhecido como método dos Gradientes Conjugados. Embora seja um método iterativo, converge para a solução verdadeira, na ausência de erros de arredondamento, em um número finito de iterações [Young 81]. Por causa desta e de muitas outras propriedades interessantes, o método

dos Gradientes Conjugados atraiu atenção considerável da comunidade de análise numérica quando, foi apresentado pela primeira vez.

É um método eficaz para sistemas de equações lineares com a matriz dos coeficientes simétrica positiva definida. É o mais antigo e mais conhecido dos métodos não-estacionários. O método gera uma seqüência de vetores de aproximação (isto é, sucessivas aproximações da solução), os resíduos correspondentes a esses vetores e vetores de direção, usados na atualização dos vetores de aproximação e respectivos resíduos. Deriva o seu nome do fato dele gerar uma seqüência de vetores conjugados (ou ortogonais). Estes vetores são os resíduos dos vetores de aproximação. Eles são também os gradientes de um funcional, a minimização do qual é equivalente a resolver o sistema linear.

Em cada iteração do método, dois produtos internos são executados, a fim de computar os escalares atualizadores, que são definidos para fazer com que as seqüências satisfaçam certas condições de ortogonalidade. Em um sistema linear com matriz dos coeficientes simétrica positiva definida, estas condições implicam que a distância para a solução verdadeira é minimizada em alguma norma.

Consideremos o problema de minimizar o funcional Q(**x**), definido por

$$Q(\mathbf{x})=\frac{1}{2}\mathbf{x}^T A\,\mathbf{x}-\mathbf{b}^T\mathbf{x},$$

onde $\mathbf{b} \in \Re^n$ e $A \in \Re^{n\times n}$ é simétrica positiva definida, ou seja, A é simétrica, com $\mathbf{x}^T A\,\mathbf{x} > 0$, para qualquer vetor $\mathbf{x} \neq 0$, de ordem $n\times 1$. O valor mínimo de Q é $-\frac{1}{2}\mathbf{b}^T A^{-1}\mathbf{b}$, obtido atribuindo $\mathbf{x} = A^{-1}\mathbf{b}$. Assim, a minimização de Q e a resolução de $A\mathbf{x} = \mathbf{b}$ são problemas equivalentes.

O problema consiste em determinar um vetor $\mathbf{x} \in \Re^n$ que seja o ponto de mínimo da função Q(**x**). Muitos Algoritmos foram sugeridos para resolução deste problema. Basicamente o procedimento deste algoritmo é de seguir iterativamente uma direção descendente de Q(**x**), até a determinação do ponto de mínimo com a precisão exigida [Pequeno 83]. Uma das estratégias mais simples para minimizar Q é o método *Steepest Descent* [Golub 89]. A idéia básica deste método consiste de um processo iterativo, no qual cada iteração apresenta dois estágios: no primeiro, a direção descendente $\mathbf{p}^{(k)}$ é determinada, e no segundo, o parâmetro $\alpha^{(k)}$ que define o tamanho do passo ao longo da direção descendente $\mathbf{p}^{(k)}$ é determinado. O resultado da iteração é dado por:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\ \mathbf{p}^{(k)}$$

o processo iterativo termina quando o ponto $\mathbf{x}^{(k)}$ satisfaz um critério de convergência, pré-determinado. A escolha de $\mathbf{p}^{(k)}$ deverá ser tal que $\mathbf{p}^{(k)}$ seja uma direção de declive de Q(**x**) [Pequeno 83].

Do cálculo, sabemos que se Q(**x**) tem um mínimo em algum ponto **x**, então as derivadas parciais de Q nesse ponto são nulos. Calculando Q_{x_i}, derivada parcial de Q em relação a x_i, e fazendo $Q_{x_i} = 0$, resulta no sistema linear $A\mathbf{x} - \mathbf{b} = 0$ ou $A\mathbf{x} = \mathbf{b}$. Daí vemos que é correto ver $A\mathbf{x} = \mathbf{b}$ como o gradiente de Q e escrever $\nabla Q(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$. Sabemos que em um ponto $\mathbf{x}^{(i)}$, a função Q decresce mais rapidamente na direção do gradiente negativo $-\nabla Q(\mathbf{x}^{(i)}) = \mathbf{b} - A\mathbf{x}^{(i)}$.

Definimos

$$\mathbf{p}^{(i)} = \mathbf{r}^{(i)} = \mathbf{b} - A\mathbf{x}^{(i)}$$

como sendo o resíduo de $\mathbf{x}^{(i)}$. Se $\mathbf{x}^{(i)}$ é alguma aproximação de **x**, então no método *Steepest Descent*, obtemos uma aproximação melhorada, $\mathbf{x}^{(i+1)}$, movendo-se, na direção de $\mathbf{p}^{(i)} = -\nabla Q(\mathbf{x}^{(i)})$, para o ponto onde $Q(\mathbf{x}^{(i+1)})$ é mínimo, isto é,

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \alpha_i \mathbf{p}^{(i)}.$$

Agora resta descobrir um valor para α_i (tamanho do passo ao longo do vetor de direção $\mathbf{p}^{(i)}$) que minimize $F(\alpha) = Q(\mathbf{x}^{(i)} + \alpha\, \mathbf{p}^{(i)})$. Calculando $F'(\alpha)$ pela regra da cadeia,

$$F'(\alpha) = \nabla Q(\mathbf{x}^{(i)} + \alpha\, \mathbf{p}^{(i)})\mathbf{p}^{(i)} = [A(\mathbf{x}^{(i)} + \alpha\, \mathbf{p}^{(i)}) - \mathbf{b})]\, \mathbf{p}^{(i)} =$$

$$= [A\mathbf{x}^{(i)} + \alpha A\mathbf{p}^{(i)} - \mathbf{b}]\mathbf{p}^{(i)} = [\mathbf{p}^{(i)} - \alpha A\mathbf{p}^{(i)}]\mathbf{p}^{(i)}.$$

Assim, fazendo $F'(\alpha) = 0$, vemos que α_i para a computação de $\mathbf{x}^{(i+1)}$, é dado por

$$\alpha_i = \frac{\mathbf{p}^{(i)}\mathbf{p}^{(i)}}{\mathbf{p}^{(i)} A\, \mathbf{p}^{(i)}}.$$

Algoritmo 4.7.2 [Wolfe 78] - Este algoritmo resolve o sistema linear $A\mathbf{x} = \mathbf{b}$, usando o método *Steepest Descent*.

1. $\mathbf{x}^0$, aproximação inicial.
2. **for** $i = 1$ **until** *maxit* **do**
3. $\quad \mathbf{p}^{(i-1)} = \mathbf{b} - A\mathbf{x}^{(i-1)}$
4. $\quad \alpha_i = \dfrac{(\mathbf{p}^{(i-1)})^{\mathrm{T}}\ \mathbf{p}^{(i-1)}}{(\mathbf{p}^{(i-1)})^{\mathrm{T}} A\, \mathbf{p}^{(i-1)}}$
5. $\quad \mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha_i\ \mathbf{p}^{(i-1)}$
6. $\quad$ **if** (o critério de convergência for satisfeito) **then**
7. $\quad\quad$ go to 10
8. $\quad$ **endif**
9. **endfor**
10. **print** Solução = $\mathbf{x}^i$
11. **print** Resíduo = $\mathbf{r}^{(i)}$

O método *Steepest Descent* é fácil de implementar, mas freqüentemente, para matrizes mal-condicionadas, a taxa de convergência é bastante lenta. O algoritmo não é considerado um bom método. O algoritmo é relativamente eficiente durante as iterações iniciais, mas quando o $\mathbf{x}^{(k)}$ se aproxima da solução $\mathbf{x}$, o algoritmo perde eficiência. Seu valor está no fato de ser o precursor de todos os métodos que utilizam o gradiente e ser um método de fácil compreensão [Pequeno 83]. A principal razão de sua convergência lenta é que, em cada iteração, nós buscamos o mínimo de Q em uma única direção. Contudo, escolhendo os nossos vetores direção diferentemente, obtemos o método dos Gradientes Conjugados, que dá a solução em não mais de n iterações, na ausência de erros de arredondamento.

Os vetores $\mathbf{x}^{(i)}$ são atualizados, em cada iteração, por um múltiplo (α_i) do vetor direção $\mathbf{p}^{(i)}$:

$$\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha_i\ \mathbf{p}^{(i)}.$$

Correspondentemente os resíduos $\mathbf{r}^{(i)} = \mathbf{b} - A\,\mathbf{x}^{(i)}$ são atualizados como

$$\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha\ \mathbf{q}^{(i)} \quad \text{onde} \quad \mathbf{q}^{(i)} = A\,\mathbf{p}^{(i)}. \tag{4.7.7}$$

A escolha de $\alpha = \alpha_i = \dfrac{(\mathbf{r}^{(i-1)})^{\mathrm{T}}\ \mathbf{r}^{(i-1)}}{(\mathbf{p}^{(i)})^{\mathrm{T}} A\, \mathbf{p}^{(i)}}$ minimiza $(\mathbf{r}^{(i)})^{\mathrm{T}} A^{-1}\, \mathbf{r}^{(i)}$ sobre todas a possíveis escolhas para α na equação (4.7.7).

Os vetores de direção são atualizados, usando os resíduos

$$\mathbf{p}^{(i)} = \mathbf{r}^{(i)} + \beta_i\ \mathbf{p}^{(i-1)},$$

onde a escolha de

$\beta_i = \dfrac{(\mathbf{r}^{(i)})^{\mathrm{T}}\ \mathbf{r}^{(i)}}{(\mathbf{r}^{(i-1)})^{\mathrm{T}}\ \mathbf{r}^{(i-1)}}$ assegura que $\mathbf{p}^{(i)}$ e $A\,\mathbf{p}^{(i-1)}$ - ou equivalentemente, $\mathbf{r}^{(i)}$ e $\mathbf{r}^{(i-1)}$ - sejam ortogonais. Esta formulação é atribuída a Fletcher e Reeves [Wolfe 78].

Assim, os resíduos $\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \ldots$ e os vetores de direção $\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \mathbf{p}^{(2)}, \ldots$ gerados, satisfazem as relações

$$(\mathbf{r}^{(i)})^{\mathrm{T}}\ \mathbf{r}^{(j)} = 0 \qquad \text{para } i \neq j, \tag{4.7.8}$$

$$(\mathbf{p}^{(i)})^{\mathrm{T}}\ A\,\mathbf{p}^{(j)} = 0 \qquad \text{para } i \neq j, \tag{4.7.9}$$

$$(\mathbf{r}^{(i)})^{\mathrm{T}}\ A\,\mathbf{p}^{(j)} = 0 \qquad \text{para } i \neq j \text{ e } i \neq j+1. \tag{4.7.10}$$

Os vetores resíduos $\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \ldots$ são mutuamente ortogonais e os vetores de direção $\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \mathbf{p}^{(2)}, \ldots$ são mutuamente A-conjugados. Da relação (4.7.8), resulta que $\mathbf{r}^{(s)} = 0$ para algum $s \leq n$. Dessa forma, o método dos Gradientes Conjugados, converge na ausência de erros de arredondamento, em não mais que n iterações [Young 81].

Algoritmo 4.7.3 [Kincaid 90] - Este algoritmo resolve o sistema linear $A\mathbf{x} = \mathbf{b}$ usando o método dos Gradientes Conjugados, onde A é uma matriz simétrica positiva definida. *maxit* é o número máximo de iterações permitido; $\mathbf{p}^{(i)}$ são os vetores de direção; e α_i são os multiplicadores usados na atualização da solução aproximada $\mathbf{x}^{(i)}$ e do resíduo $\mathbf{r}^{(i)}$.

1. Compute $\mathbf{r}^0 = \mathbf{b} - A\,\mathbf{x}^0$ para alguma aproximação $\mathbf{x}^0$.
2. **for** $i = 1$ **until** *maxit* **do**
3. **if** $i = 1$ **then**
4. $\mathbf{p}^1 = \mathbf{r}^0$
5. **else**
6. $\mathbf{p}^{(i)} = \mathbf{r}^{(i)} + \dfrac{(\mathbf{r}^{(i)})^{\mathrm{T}}\ \mathbf{r}^{(i)}}{(\mathbf{r}^{(i-1)})^{\mathrm{T}}\ \mathbf{r}^{(i-1)}}\ \mathbf{p}^{(i-1)}$
7. **endif**
8. $\mathbf{q}^{(i)} = A\,\mathbf{p}^{(i)}$
9. $\alpha_i = \dfrac{(\mathbf{r}^{(i-1)})^{\mathrm{T}}\ \mathbf{r}^{(i-1)}}{(\mathbf{p}^{(i)})^{\mathrm{T}}\ \mathbf{q}^{(i)}}$
10. $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha_i\ \mathbf{p}^{(i)}$
11. $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha_i\ \mathbf{q}^{(i)}$
12. **if** ($\left\| \mathbf{r}^{(i)} \right\|$ < Tolerância) **go to** 14 /* Verifica a convergência; continua se necessário */
13. **endfor**
14. **print** Solução = $\mathbf{x}^i$
15. **print** Resíduo = $\mathbf{r}^{(i)}$

4.8 - Análise de erros na Solução de Sistemas Lineares

Embora, teoricamente os métodos de eliminação conduzam à solução exata, na prática, por causa dos erros de arredondamento, a solução computada $\mathbf{x}_c$ pode ser uma boa solução, ou não. Só não haveria essa introdução de erros, se trabalhássemos com precisão infinita. Essa situação é agravada quando se aplica os métodos na resolução de problemas do mundo real, onde a construção do sistema (matriz A e vetor **b**) estão afetados de erros. Esses erros são chamados erros inerentes ao problema que, muitas vezes, não são conhecidos. Nesta seção, serão apresentados critérios para verificar se uma solução computada de um sistema linear é aceitável ou não, segundo [Hattori 96], [Dorn 72], [Albrecht 73] e [Steinberg 74].

Passemos a definir alguns tipos de erro. Seja **x** o valor exato, também chamado grandeza verdadeira, e $\mathbf{x}_c$ um valor aproximado de **x**. Então:

Definição 4.8.1 - O **erro absoluto** (E) é a diferença entre o valor exato (**x**) e o valor aproximado ($\mathbf{x}_c$)

$$E = \mathbf{x} - \mathbf{x}_c \,.\, \square$$

Na maioria dos casos, estamos interessados no valor absoluto de E, $|E|$. Dizemos assim que $\mathbf{x} = \mathbf{x}_c \pm |E|$, em que $|E|$ representa uma incerteza no valor de $\mathbf{x}_c$.

Definição 4.8.2 - O **erro relativo** é a razão entre o erro absoluto e o valor exato

$$e = \frac{|E|}{|\mathbf{x}|} = \frac{|\mathbf{x} - \mathbf{x}_c|}{|\mathbf{x}|} \,.\, \square$$

Definição 4.8.3 - O **erro porcentual** (e_p) é obtido multiplicando-se o erro relativo por 100. □

Por exemplo, para $\mathbf{x} = 0{,}9995$ e $\mathbf{x}_c = 0{,}9994$ tem-se

$$E = 0{,}0001, \quad e = 0{,}0001 \quad \text{e} \quad e_p = 0{,}01\%$$

Assim, suponha que $\mathbf{x}' \in \Re^n$ é uma aproximação para $\mathbf{x} \in \Re^n$. Para uma dada norma de vetor, $\| \bullet \|$, dizemos que

$$\varepsilon_{abs} = \|\mathbf{x}' - \mathbf{x}\|$$

é o erro absoluto em $\mathbf{x}$ enquanto que se $\mathbf{x} \neq 0$ então

$$\varepsilon_{rel} = \frac{\|\mathbf{x}' - \mathbf{x}\|}{\|\mathbf{x}\|}$$

descreve o erro relativo em $\mathbf{x}$.

O valor absoluto fornece uma maneira conveniente de medir o "tamanho" de números reais, ou até mesmo números complexos. Para muitas situações, entretanto, é suficiente medir o tamanho de um vetor de dimensão n, através de uma norma.

Definição 4.8.4 - A norma vetorial, denotada por $f = \| \bullet \|$, é uma função f: $\Re^n \rightarrow \Re$ que satisfaz as seguintes propriedades:

1. $\| \mathbf{x} \| \geq 0$; $\| \mathbf{x} \| = 0$ implica $\mathbf{x} = 0$ (vetor nulo),
2. $\|k\,\mathbf{x}\| = |k| \, \| \mathbf{x} \|$, $k \in \Re$ e $\mathbf{x} \in \Re^n$,
3. $\| \mathbf{x} + \mathbf{y} \| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$, $\mathbf{x}, \mathbf{y} \in \Re^n$.

Índices nas barras duplas são usados para distinguir entre os vários tipos de norma. Uma classe útil de normas de vetor é a norma-p, definida por:

$$\|\mathbf{x}\|_p = \left(|x_1|^p + \cdots + |x_n|^p\right)^{\frac{1}{p}}, \qquad p \geq 1.$$

Destas, as normas 1, 2 e ∞ são as mais importantes:

$\|\mathbf{x}\|_1 = |x_1| + \cdots + |x_n|$, também chamada de norma uniforme.

$\|\mathbf{x}\|_2 = \left(|x_1|^2 + \cdots + |x_n|^2\right)^{\frac{1}{2}} = \left(\mathbf{x}^T\mathbf{x}\right)^{\frac{1}{2}}$, chamada também de norma euclidiana que é equivalente ao comprimento ou distância da geometria analítica.

$\|\mathbf{x}\|_\infty = \max_i \{|x_i|\}$ norma ∞ também chamada de norma de Chebyschev.

Um vetor unitário com respeito a uma dada norma $\| \bullet \|$ é um vetor $\mathbf{x}$ que satisfaz $\|\mathbf{x}\| = 1$.

A análise de algoritmos para matrizes freqüentemente requer o uso de normas de matriz. Por exemplo, a qualidade da solução de um sistema linear pode ser pobre se a matriz dos coeficientes é "quase singular". Para quantificar a noção de quase singularidade necessitamos de uma medida de distância no espaço vetorial das matrizes. Normas de matriz fornecem essa medida.

Como $\Re^{m \times n}$ é isomórfico a $\Re^{mn}$, a definição de uma norma de matriz deverá ser equivalente a definição de uma norma de vetor [Golub 87].

Definição 4.8.5 - A norma de matriz, denotada por $f = \| \bullet \|$, é uma função f: $\Re^{m\times n} \to \Re$ que satisfaz as seguintes propriedades:

1. $\| A \| \geq 0$; $\quad A \in \Re^{m\times n}$, $(\| A \| = 0 \text{ se } A = 0)$,
2. $\| kA \| = |k| \, \| A \|$, $\quad k \in \Re, A \in \Re^{m\times n}$
3. $\| A + B \| \leq \|A\| + \|B\|$, $\quad A, B \in \Re^{m\times n}$

Como em normas de vetor, usamos a notação de barras duplas com subscritos para designar norma de matriz, isto é, $\|A\| = f(A)$.

As normas de matriz mais comumente utilizadas são aquelas induzidas pelas normas 1, 2 e ∞ de vetor:

$$\|A\|_1 = \max_{x \neq 0} \frac{\|A\mathbf{x}\|_1}{\|\mathbf{x}\|_1} = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{ij}|,$$

$$\|A\|_2 = \max_{x \neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = (\text{maior auto-valor de } A^T A)^{1/2},$$

$$\|A\|_\infty = \max_{x \neq 0} \frac{\|A\mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty} = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|.$$

Observe que as normas de matriz foram definidas em função das normas de vetor (lembre-se que $A\mathbf{x}$ é um vetor).

Ao aplicarmos um método obtemos a solução computada $\mathbf{x}_c$, que poder ser uma boa solução, ou não. Apresentaremos um meio para verificar se uma solução computada de um sistema linear é aceitável ou não.

Discutiremos o difícil problema de determinar o erro de uma solução aproximada (sem o conhecimento da solução). Apresentaremos e usaremos normas como uma maneira de medir o "tamanho" de vetores e matrizes.

Se $\mathbf{x}_c$ é uma solução computada para o sistema linear $A\mathbf{x} = \mathbf{b}$, então seu erro absoluto é a diferença

$$\mathbf{e} = \mathbf{x} - \mathbf{x}_c.$$

Este erro é, naturalmente, não conhecido por nós. Mas podemos sempre computar o erro residual

$$\mathbf{r} = A\mathbf{x} - A\mathbf{x}_c$$

uma vez que $A\mathbf{x}$ é justamente $\mathbf{b}$. O resíduo, então, mede quão bem $\mathbf{x}_c$ satisfaz o sistema linear $A\mathbf{x} = \mathbf{b}$. Se $\mathbf{r}$ é o vetor nulo, então $\mathbf{x}_c$ é a (exata) solução, isto é, $\mathbf{e}$ é, então, zero. Espera-se que cada entrada do vetor $\mathbf{r}$ seja pequena, se $\mathbf{x}_c$ é uma boa aproximação para a solução $\mathbf{x}$.

O tamanho do vetor residual, $\mathbf{r} = \mathbf{b} - A\mathbf{x}_c$, de uma solução computada $\mathbf{x}_c$ não é sempre um indicador seguro do tamanho do erro, $\mathbf{e} = \mathbf{x} - \mathbf{x}_c$, nesta solução computada. Se ou não um pequeno resíduo implica em um "pequeno" erro, depende do tamanho da matriz dos coeficientes e de sua inversa. Para essa discussão, necessitamos de um meio de medir o "tamanho" de vetores de ordem n e matrizes $n \times n$.

Um outro critério é recomputar a solução, introduzindo uma perturbação "pequena" no problema, alterando ligeiramente um ou mais dados (computando a solução em precisão dupla, caso seja possível, é uma prática comum) e verificar se vai haver grandes alterações na solução.

Vamos agora tratar de obter uma estimativa da relação entre as perturbações relativas nos dados ($\mathbf{b}$ e A) e na solução. Comecemos perturbando $\mathbf{b}$ e calculando a perturbação em $\mathbf{x}$. Seja

$$A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$$

o sistema perturbado. Temos

$$A\mathbf{x} + A\delta\mathbf{x} = \mathbf{b} + \delta\mathbf{b},$$

$$A\delta\mathbf{x} = \delta\mathbf{b},$$

e, em alguma norma,

$$\|\delta\mathbf{x}\| \leq \|A^{-1}\| \|\delta\mathbf{b}\|. \tag{4.8.1}$$

Perturbando, agora, A

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b},$$

$$A\mathbf{x} + A\,\delta\mathbf{x} + \delta A\mathbf{x} + \delta A\delta\mathbf{x} = \mathbf{b},$$

$$A\,\delta\mathbf{x} + \delta A(\mathbf{x} + \delta\mathbf{x}) = 0,$$

de onde se retira

$$\delta\mathbf{x} = -A^{-1}\delta A(\mathbf{x} + \delta\mathbf{x}),$$

e, em alguma norma,

$$\|\delta\mathbf{x}\| \leq \|A^{-1}\| \|\delta A\| \|\mathbf{x} + \delta\mathbf{x}\|,$$

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x} + \delta\mathbf{x}\|} \leq \|A^{-1}\| \|A\| \frac{\|\delta A\|}{\|A\|}, \tag{4.8.2}$$

que expressa a perturbação relativa na solução $\mathbf{x}$, devida à perturbação relativa em A.

O número

$$\kappa = \left\| A^{-1} \right\| \left\| A \right\| \tag{4.8.3}$$

é chamado número de condição da matriz A, em relação a alguma norma dada.

Da desigualdade

$$\|b\| = \|A\ \mathbf{x}\| \leq \|A\| \|\mathbf{x}\|$$

tira-se

$$\frac{1}{\|\mathbf{x}\|} \leq \frac{\|A\|}{\|\mathbf{b}\|}.$$

Multiplicando ambos os membros de (4.8.2) por $\frac{1}{\|\mathbf{x}\|}$ e utilizando a última desigualdade, obtemos

$$\frac{\|\delta\ \mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A^{-1}\| \|A\| \frac{\|\delta\ \mathbf{b}\|}{\|\mathbf{b}\|}, \tag{4.8.4}$$

que dá uma estimativa da perturbação relativa em **x**, devida à perturbação relativa em **b**.

A interpretação que damos a (4.8.2) e (4.8.3) é a seguinte: se $\kappa(A)$ for grande, uma pequena perturbação relativa em A e **b** produzirá grandes perturbações relativas na solução **x**.

Observemos que $\kappa(A) \geq 1$, o que se obtém tomando a norma de $I = A\,A^{-1}$

$$1 = \|I\| = \left\| A\,A^{-1} \right\| \leq \|A\| \left\| A^{-1} \right\|. \square$$

Vemos que a estimativa do número de condição é crucial para obter uma estimativa da qualidade da solução computada. É interessante averiguar as respostas a duas questões:

(1) Podemos tratar previamente a matriz A de modo que menos erros de arredondamento seja gerado durante a eliminação ?

(2) Tendo uma solução computada, podemos melhorar a sua precisão de alguma forma ?

As seções 4.81 e 4.8.2 tentarão responder a essas questões [Hattori 96].

4.8.1 - Escalamento

Da análise de erros no produto interno, e considerando que a maioria das operações nos algoritmos de triangularização podem ser agrupadas na forma de produtos internos, fica evidente que é conveniente termos todos os elementos da matriz na mesma ordem de grandeza, isto é, termos a matriz bem *equilibrada* . Evitamos, assim, ter multiplicadores muito pequenos e soma de termos de ordem de grandeza muito diferentes. Se tal não ocorre para a matriz de um dado sistema, $A\mathbf{x} = \mathbf{b}$, podemos encontrar **x**, através da solução de um outro sistema melhor equilibrado.

Se $E = diag(e_1, e_2, \ldots, e_n)$ e $D = diag(d_1, d_2, \ldots, d_n)$, onde $e_i, d_i \neq 0$, o sistema $EAD\mathbf{y} = E\mathbf{b}$ é obtido multiplicando-se a i-ésima equação do sistema por e_i, e efetuando-se a substituição de variáveis $\mathbf{x} = D\mathbf{y}$, o que equivale a multiplicar a j-ésima coluna de A por d_j. Uma transformação $\bar{A} = EAD$, A $m \times n$, E e D diagonais com $e_i, d_i \neq 0$, é um escalamento da matriz A.

Seja B a base usada na mantissa e defina as matrizes diagonais D_1 e D_2 por

$$D_1 = \text{diag}[B^{r_1}, \ldots, B^{r_n}] \text{ e } \quad D_2 = \text{diag}[B^{c_1}, \ldots, B^{c_n}].$$

A solução do sistema $A\mathbf{x} = \mathbf{b}$ pode ser obtida, resolvendo o sistema escalado

$(D_1^{-1} A D_2)\,\mathbf{w} = D_1^{-1}\,\mathbf{b}$:

$P(D_1^{-1} A D_2) = LU$ (fatoração),

$L\,\mathbf{v} = P(D_1^{-1}\,\mathbf{b})$ (substituição progressiva),

$U\mathbf{w} = \mathbf{v}$ (substituição regressiva),

$\mathbf{x} = D_2$ (compensa escalamento).

Os escalamentos de A, $\mathbf{b}$ e $\mathbf{w}$ requerem $O(n^2)$ operações de multiplicação. Observe que a multiplicação $D_1^{-1}A$ divide as linhas de A por $B^{r_1}, \ldots, B^{r_n}$ sem introduzir erros de arredondamento, escalando as equações, enquanto que D_2 escala as incógnitas.

Assim, escolhendo D_1 e D_2 de modo que $\kappa_\infty(D_1^{-1} A D_2)$ seja o menor possível, devemos obter $\mathbf{x}_c$ com maior precisão.

Um desses métodos é o escalamento de linhas. Neste método $D_2 = \text{I}$ e D_1 é escolhido de modo que cada linha de $D_1^{-1}A$ tenha a mesma norma ∞. O escalamento de linha minimiza a possibilidade de adição de um número muito pequeno a um número muito grande durante a eliminação.

A escolha dos elementos de D_1 e D_2, como potências da base usada na mantissa, procura evitar introdução de erros no escalamento.

A idéia do escalamento de linhas pode ser incorporado no algoritmo de Eliminação de Gauss, conforme mostra o algoritmo abaixo.

Algoritmo 4.8.1 Fatoração LU com escalamento de linhas e *pivoteamento* parcial.

1. **for** $i = 1$ **until** n **do** /* Calcula o fator de escalamento de cada linha */
2. $S_i := \max_j |a_{ij}|$
3. **endfor**
4. **for** $k = 1$ **until** $n - 1$ **do**
   /* encontra o maior elemento escalado na coluna k */
5. Encontre r tal que $\frac{|a_{rk}|}{S_r} = \max_i \frac{|a_{ik}|}{S_i}$, com $i = k, k+1, \ldots, n$;
6. **if** $a_{rk} = 0$ **then** a matriz A é singular; **stop**
7. Permute as linha r e k
8. **for** $i = k + 1$ **until** n **do**
9. $a_{ik} := \frac{a_{ik}}{a_{kk}}$
10. **for** $j = k + 1$ **until** n **do**
11. $a_{ij} := a_{ij} - a_{ik} a_{kj}$
12. **endfor**
13. **endfor**
14. **endfor** □

Escalamento visando equilibrar a matriz do problema, é uma etapa importante na solução de sistemas lineares de grande porte. As operações de escalamento não afetam os elementos nulos de uma matriz, e portanto não alteram sua estrutura e esparsidade.

4.8.2 - Refinamento Iterativo

Suponha que $A\mathbf{x} = \mathbf{b}$ tenha sido resolvido pela fatoração $PA = LU$ e que desejamos melhorar a precisão da solução computada $\mathbf{x}_c$. Calculemos

$$\begin{aligned} \mathbf{r} &= \mathbf{b} - A\,\mathbf{x}_c\,, \\ L\mathbf{y} &= P\mathbf{r}\,, \\ U\mathbf{z} &= \mathbf{y}\,, \\ \mathbf{v} &= \mathbf{x}_c + \mathbf{z}. \end{aligned} \qquad (4.8.5)$$

Se a aritmética usada fosse exata, $\mathbf{v}$ deveria satisfazer exatamente o sistema

$$A\mathbf{v} = A(\mathbf{x}_c + \mathbf{z}) = \mathbf{b} - \mathbf{r} + A\mathbf{z} = \mathbf{b}$$

e podemos encarar $\mathbf{z}$ como sendo a correção necessária em $\mathbf{x}_c$, para atingir a solução correta. Na realidade, $\mathbf{z}$ também não é a correção procurada, porque pode estar sujeito a erro e, pior ainda, se for computada ingenuamente, $\mathbf{v}$ não será mais preciso que $\mathbf{x}_c$. Para que $\mathbf{z}$ seja realmente uma correção é preciso computar o resíduo em precisão dupla daquela utilizada no cálculo de $PA = LU$, $\mathbf{x}$, $\mathbf{y}$ e $\mathbf{z}$. Como a correção será também aproximada, podemos repetir os cálculos indicados

em (4.8.5) usando, agora, **v** como nova aproximação da solução O processo pode ser resumido no algoritmo 4.8.2.

Algoritmo 4.8.2 - Refinamento iterativo

1. Efetuar a fatoração $PA = LU$; $\mathbf{x} = 0$ (t dígitos)
2. **while** (não converge) **do**
3. $\mathbf{r} := \mathbf{b} - A\mathbf{x}$; (2t dígitos)
4. Resolva $L\mathbf{y} = P\mathbf{r}$; (t dígitos)
5. Resolva $U\mathbf{z} = \mathbf{y}$; (t dígitos)
6. $\mathbf{x} := \mathbf{x} + \mathbf{z}$ (t dígitos)
7. **endwhile**; □

Este método é conhecido como **refinamento iterativo**. Cabem aqui algumas observações. Primeira, o calculo do resíduo **r** deve ser feito utilizando a matriz original A e não a recuperada pelo produto $P^{T}LU$. Segunda, em computadores que só trabalham com precisão simples e dupla, sua capacidade deverá ser estendida por software, se quisermos utilizar o refinamento iterativo quando os dados já estiverem em precisão dupla (muitos processadores já possuem hardware para precisão quádrupla). Terceira, a necessidade de reter uma cópia da matriz A é um outro incômodo, se os espaço de memória for crítico.

No algoritmo 4.8.2, omitimos o critério de convergência. Dois critérios de convergência são usados comumente:

$$\text{a)}\ \frac{\|\mathbf{z}\|}{\|\mathbf{x}\|} \leq B^{-t} \qquad \text{b)}\ \|\mathbf{x}\| + \lambda\|\mathbf{z}\| = \|\mathbf{x}\|.$$

em que λ é um número pequeno (5 a 10) e a norma recomendada é 1 ou ∞, por serem mais econômicas de computar. O critério de parada (a), depende dos parâmetros ambientais do processador de ponto flutuante enquanto que o (b) é independente desses parâmetros; o segundo critério se presta para uma implementação portátil enquanto que o primeiro, para uma implementação transportável.

4.9 - Resumo

Tratamos da solução de um sistema linear $A\mathbf{x} = \mathbf{b}$ com uma matriz A, $n \times n$, não singular. O problema pode ser desdobrado na solução de dois sistemas:

$$L\mathbf{y} = \mathbf{b}$$

$$U\mathbf{x} = \mathbf{y}$$

onde L e U são obtidas pela Fatoração da matriz A na forma LU. Para manter a estabilidade dos algoritmos da Fatoração LU, é necessário usar uma técnica chamada *pivoteamento* parcial. Finalmente, foi mostrado como implementar a fatoração LU com *pivoteamento* parcial, fazendo escrituração eficiente das permutações de linhas.

Vimos também que a solução $\mathbf{x}'$ de um sistema linear $A\mathbf{x} = \mathbf{b}$ pode ser obtida utilizando-se um método iterativo, que consiste em calcular uma seqüência $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(k)}, \ldots$ de aproximações de $\mathbf{x}'$, sendo dada uma aproximação inicial $\mathbf{x}^{(0)}$.

Vimos, também, dois métodos para melhorar uma solução computada. Um deles dá um tratamento preliminar à matriz A para reduzir o κ (escalamento). O outro melhora a solução computada, via Fatoração LU, por meio do refinamento iterativo.

Em geral, o custo de obtenção da resolução de sistemas de equações lineares aumenta com a ordem n da matriz associada ao sistema. Se n é pequeno, mesmo um algoritmo pouco eficiente terá um baixo custo computacional para executá-lo. Em muitos casos, quando n cresce surge uma situação onde o tempo de execução do algoritmo não é aceitável. O próximo capítulo descreve duas abordagens para o processamento paralelo de sistemas lineares.

Capítulo 5

Resultados e Discussões

5.1 - Introdução

Neste capítulo serão apresentadas as descrições das matrizes dos coeficientes dos sistemas lineares usados nos testes, um plano de testes, os resultados dos testes realizados e uma avaliação comparativa entre o processamento serial e o paralelo dos métodos utilizados. Nas análises, tomaremos como base a ordem da matriz, o tempo de execução e a qualidade da solução encontrada.

5.2 - Descrição das matrizes

A matriz *A* usada nos testes com a Eliminação de Gauss e Fatoração LU é do tipo geral. Os elementos desta matriz foram gerados através de um gerador de números aleatórios descrito por Schrage [Schrage 79] chamado SNADUN(). Os elementos do vetor dos termos independentes **b** são tais que todos os elementos do vetor solução **x** sejam iguais a 1. Os testes foram realizados com sistemas lineares de ordem 100, 200, 500, 600, 700, 900, 1.000, 1.500 e 2.000.

A matriz *A* usada nos testes com o método iterativo de Gauss-Jacobi é do tipo estritamente diagonal dominante. Os testes foram realizados com sistemas lineares de ordem 500, 1.000, 1.300, 1.500, 2.000, 2.500, 3.000, 4.000 e 5.000. Os elementos do vetor dos termos independentes **b** são tais que todos os elementos do vetor solução **x** sejam iguais a 1.

A matriz *A* usada nos testes com o método iterativo dos Gradientes Conjugados é do tipo simétrica positiva definida estritamente diagonal dominante. Os testes foram realizados com sistemas lineares de ordem 500, 1.000, 2.000, 3.000, 4.000 e 5.000. Os elementos do vetor dos termos independentes **b** são tais que todos os elementos do vetor solução **x** sejam iguais a 1.

5.2.1 - Matriz geral

Para gerar os elementos da matriz geral,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

que foi usada nos testes com a Eliminação de Gauss e a Fatoração LU, usamos o algoritmo 5.1.

Algoritmo 5.1 Este algoritmo gera a matriz geral de números reais em precisão dupla usada nos testes com a Eliminação de Gauss e a Fatoração LU. n é o número de equações e incógnitas; a_{ij} para $i, j = 1, 2, 3, \ldots, n$ são os elementos da matriz A; *semente* é um número real fornecido pelo usuário com $0 < semente \leq 2^{S-1} - 1$, onde S é o número de bits utilizado na representação de inteiros; *aleat* é o número real aleatório retornado pela rotina.

```
1. for j = 1 until n do
2.     for i = 1 until n do
3.         call SNADUN( semente , aleat )
4.         if ( MOD( semente , 2) = 0 ) then
5.             sinal : = +1
6.         else
7.             sinal : = -1
8.         endif
9.         a_ij = aleat × sinal
10.    endfor
11.endfor
```

5.2.2 - Matriz estritamente diagonal dominante

A matriz de números reais em precisão dupla usada nos testes com o método iterativo de Gauss-Jacobi deve ser estritamente diagonal dominante, ou seja,

$$|a_{ii}| > \sum_{j=1, j\neq i} |a_{ij}|, \qquad \text{para } i = 1, 2, 3, \ldots, n.$$

Os elementos da matriz estritamente diagonal dominante são gerados pelo algoritmo 5.2.

Algoritmo 5.2 Este algoritmo gera a matriz estritamente diagonal dominante de números reais, em precisão dupla, usada nos testes com o método iterativo de Gauss-Jacobi. n é o número de equações e incógnitas; a_{ij} para i , $j = 1, 2, 3, \ldots, n$, são os elementos da matriz A; *semente* é um número real fornecido pelo usuário com $0 < semente \leq 2^{S-1} - 1$, onde S é o número de bits utilizado na representação de inteiros; *aleat* é um número real, em precisão dupla, aleatório retornado pela rotina.

1. **for** $j = 1$ **until** n **do**
2. **for** $i = 1$ **until** n **do**
3. call SNADUN(*semente* , *aleat*)
4. **if** (MOD(*semente* , 2) = 0) **then**
5. sinal : = +1
6. **else**
7. sinal : = −1
8. **endif**
9. a_{ij} = *aleat* × sinal
10. **endfor**
11. **endfor**
12. **for** $i = 1$ **until** n **do**
13. sum : = 0
14. **for** $j = 1$ **until** n **do**
15. **if** (j ≠ i) **then**
16. sum : = sum + $\left| a_{ij} \right|$
17. **endif**
18. a_{ij} = sum + sum
19. **endfor**
20. **endfor**

Em ambos os casos anteriores, os elementos do vetor dos termos independentes **b** são gerados, de modo que a solução é $\mathbf{x} = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^T$. O algoritmo é:

Algoritmo 5.3 Este algoritmo gera a matriz simétrica positiva definida (SPD), estritamente diagonal dominante, de números reais em precisão dupla, usada nos testes com o método iterativo dos Gradientes Conjugados. n é o número de equações e incógnitas; a_{ij}, para $i, j = 1, 2, 3, \ldots, n$, são os elementos da matriz A.

1. **for** $i = 1$ **until** n **do**
2. soma = 0.0D0
3. **for** $j = 1$ **until** n **do**
4. **if** ($j < i$) **then do**
5. $a_{ij} = a_{ji}$
6. **else**
7. a_{ij} = Número_em_precisão_dupla($i + j$) / 1000.0D0
8. **endif**
9. soma = soma + valor_absoluto(a_{ij})
10. **endfor**
11. a_{ij} = soma + soma
12. **endfor**

Algoritmo 5.4 Este algoritmo gera os elementos do vetor dos termos independentes **b**, de modo que a solução do sistema seja $\mathbf{x} = [\,1 \quad 1 \quad 1 \quad \ldots. \; 1\,]^{T}$. a_{ij}, para i, $j = 1, 2, 3, \ldots, n$, são os elementos da matriz A; $b_1, b_2, \cdots, b_n$ são os elementos do vetor **b**.

1. **for** $i = 1$ **until** n **do**
2. soma = 0.0D0
3. **for** $j = 1$ **until** n **do**
4. soma = soma + a_{ij}
5. **endfor**
6. b_i = **soma**
7. **endfor**

5.3 - Observações sobre implementação

Na implementação das rotinas decidiu-se:

1. não usar o processamento de matrizes por colunas;
2. não adotar método algum para economizar espaço de memória para armazenar a matriz dos coeficientes; essas matrizes são inteiramente armazenadas;
3. para o cálculo do resíduo da solução de um sistema linear, foi escolhida a norma ∞ ou de Chebyshev; e
4. A tolerância ε usada nas rotinas serial e paralela para os métodos iterativos, é 1×10^{-12} .

5.3.1 - Processamento de matrizes por linhas

Na implementação dos algoritmos, foi utilizada a linguagem de programação FORTRAN. Nas operações entre matrizes e vetores, usamos o processamento por linhas.

5.3.2 - Cálculo dos resíduos

Para medir a qualidade dos resultados obtidos nos testes, usamos a norma infinita

$$\|a\|_{\infty} = \max_{1 \le i \le n} |a_i| .$$

O algoritmo que faz o cálculo dos resíduos é:

Algoritmo 5.5 - Este algoritmo faz o cálculo dos resíduos do sistema $A\mathbf{x} = \mathbf{b}$. n é o número de equações e incógnitas; aa_{ij} , para i , j = 1, 2, 3, ... , n, são os elementos da matriz AA, que contém os elementos da matriz original A; $bb_1, bb_2, \cdots , bb_n$ são elementos do vetor **bb**, que contém os elementos originais de **b**; $x_1, x_2, \ldots , x_n$ são os elementos do vetor solução **x**.

```
1. for  i = 1 until  n  do
2.      v_i = 0.0D0
3. endfor
4. for  j = 1 until  n  do
5.      for  i = 1 until  n  do
6.           v_i = v_i + aa_ij × x_j
7.      endfor
8. endfor
9. for  i = 1 until  n  do
10.     resid_i = abs ( bb_i  -  v_i )
11.endfor
12. índice = ISAMAX( n , resid , 1);
13. print "Resíduo=", resid_indice
```

5.3.3 - Critério de interrupções das iterações

Para interromper as iterações nos testes com o método iterativo de Gauss-Jacobi, foi usado o seguinte critério:

- Teste da variação absoluta: $\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| < \varepsilon_1$. Se $\varepsilon_1 = 10^{-p}$ e $\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| < \varepsilon_1$, então todos os componentes de $\mathbf{x}^{(k+1)}$ e $\mathbf{x}^{(k)}$ têm os primeiros p dígitos coincidentes.

5.3.4 - Análise do desempenho

O tempo de execução da Tarefa-Mestra foi computado de duas formas. Na primeira, usamos o comando do Sistema Operacional AIX™, timex, cuja sintaxe é:

timex < programa_executável>.

Na Segunda, o tempo de execução foi calculado através do uso da função do FORTRAN, chamada DATE_AND_TIME( ). Usamos essa função no começo e no final do código da Tarefa-Mestra e depois calculamos a diferença dos tempos encontrados. O tempo computado é dado em s (segundos).

5.4 - Plano de testes

Os métodos que foram paralelizados são:

- Eliminação de Gauss
- Fatoração LU
- Método iterativo de Gauss-Jacobi
- Método iterativo dos Gradientes Conjugados

Na implementação das rotinas para o processamento paralelo dos métodos citados acima foram usadas duas abordagens:

- Primeira abordagem
- Segunda abordagem

5.4.1 - Primeira abordagem

A idéia inicial, foi distribuir processamento entre as Tarefas-Escravas, sem minimizar a passagem de parâmetros entre a Tarefa-Mestra e as Tarefas-Escravas. O processamento realizado pelos algoritmos paralelos para a Eliminação de Gauss, Fatoração LU e método iterativo de Gauss-Jacobi ocorre de acordo com os seguintes algoritmos:

Algoritmo 5.6 - Algoritmo da Tarefa-Mestra, usando a primeira abordagem para a Eliminação de Gauss e Fatoração LU.

1. **while** ($i \leq$ número_linhas − 1) **do**
2. Encontra a linha pivô p do estágio i.
3. Distribui a linha pivô p para as Tarefas-Escravas.
4. Distribui os blocos da matriz A para as Tarefas-Escravas.
5. Recebe de cada Tarefa escrava os blocos da matriz A, que foram atualizados pelo processo de eliminação.
6. **endwhile**

Algoritmo 5.7 - Algoritmo da Tarefa-Escrava, usando a primeira abordagem para a Eliminação de Gauss e Fatoração LU.

1. Recebe a linha pivô p.
2. Recebe um bloco b da matriz A.
3. Executa a Eliminação de Gauss no bloco b usando a linha pivô p.
4. Envia o bloco b, que foi atualizado pela Eliminação de Gauss para a Tarefa-Mestra.

A Figura 5.1 ilustra o funcionamento dos algoritmos 5.6 e 5.7. Ela mostra como se dá a comunicação entre as tarefas, para os métodos diretos, utilizando a abordagem inicial de comunicação. As setas indicam passagem de mensagens entre as tarefas. Veja que em cada passo da Eliminação, a Tarefa-Mestra envia a cada Tarefa-Escrava a linha pivô e um dos blocos da matriz A. Em seguida, a Tarefa-Mestra espera para receber de volta esses blocos, atualizados pelas Tarefas-Escravas. Observe que o tráfego entre as Tarefas é bastante acentuado. Essa comunicação ocorre até o último passo da Eliminação.

O trabalho de cada Tarefa-Escrava consiste em receber a linha pivô e um dos blocos da matriz A, e aplicar o método de Eliminação sobre esses dados. Em seguida, a Tarefa-Escrava envia o bloco atualizado para a Tarefa-Mestra.

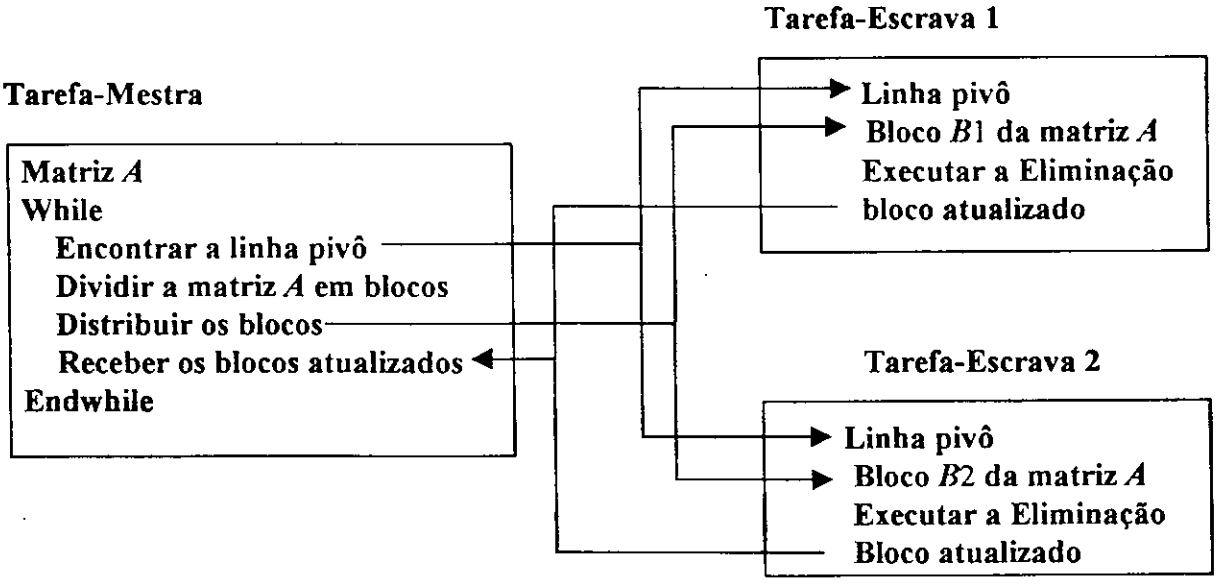

Figura 5.1 – Primeira abordagem de comunicação entre as tarefas – Métodos diretos

Algoritmo 5.8 - Algoritmo da Tarefa-Mestra usando a primeira abordagem para o método iterativo de Gauss-Jacobi.

1. divide a matriz A em k blocos, cada bloco tendo m linhas.
2. divide o vetor independente b em k vetores partição, cada vetor tendo m elementos.
3. **while** (iteração ≤ MAXITE) **do**
4. Distribui o vetor aproximação $\mathbf{x}^0$ para as k Tarefas-Escravas.
5. Distribui os k blocos da matriz A para as k Tarefas-Escravas.
6. distribui as k partições do vetor **b** para as k Tarefas-Escravas.
7. **while** (i ≤ k) **do**
8. Recebe de cada Tarefa-Escrava uma partição do novo vetor aproximação $\mathbf{x}^0$.
9. **endwhile**
10. **endwhile**

Algoritmo 5.9 - Algoritmo da Tarefa-Escrava, usando a primeira abordagem para o método iterativo de Gauss-Jacobi.

1. Recebe o vetor aproximação $\mathbf{x}^0$.
2. Recebe um dos k blocos da matriz A.
3. Recebe uma das k partições do vetor **b**.
4. Calcula uma das k partições do novo vetor iteração.
5. Envia a partição do novo vetor aproximação para a Tarefa-Mestra.

A Figura 5.2 ilustra o funcionamento dos algoritmos 5.8 e 5.9. Ela mostra como ocorre o tráfego de dados entre as tarefas, para o método de Gauss-Jacobi, utilizando a abordagem inicial de comunicação. Observe que a cada iteração, a Tarefa-Mestra envia para cada Tarefa-Escrava o vetor aproximação $\mathbf{x}^\circ$, um dos blocos da matriz A e uma das partições do vetor dos termos independentes **b**. Feito isto, a Tarefa-Mestra espera para receber, das Tarefas-Escravas, partições de vetor que vão compor o novo vetor aproximação $\mathbf{x}^\circ$. Este comunicação se repete até que um critério de parada seja satisfeito.

O trabalho de cada Tarefa-Escrava é receber o vetor aproximação $\mathbf{x}^\circ$, um bloco de dados da matriz A e uma das partições do vetor **b**. Em seguida, ela aplica o método de Gauss-Jacobi sobre esses dados, e o resultado dessa operação, que é uma partição de vetor, é enviado para a Tarefa-Mestra.

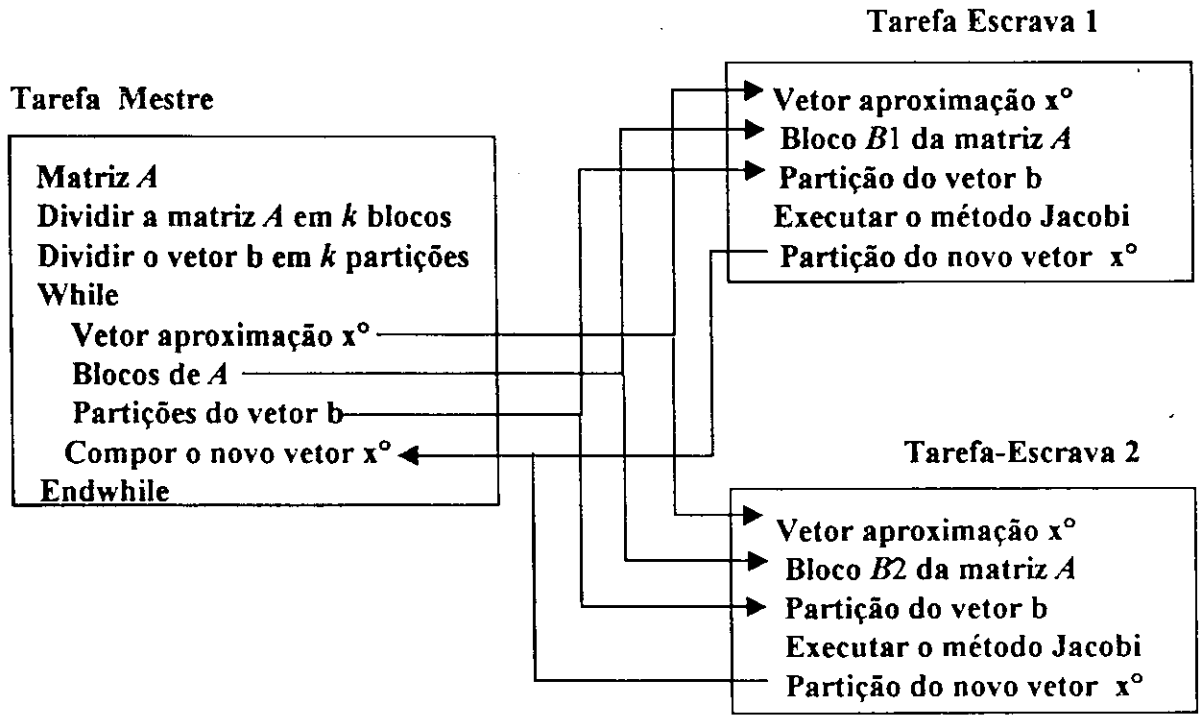

Figura 5.2 – Primeira abordagem de comunicação entre as tarefas – Método de Gauss-Jacobi

5.4.2 - Segunda abordagem

A idéia foi distribuir o processamento entre as Tarefas-Escravas, com o menor grau de comunicação entre as Tarefas-Escravas e a Tarefa-Mestra, ou seja, a passagem de parâmetros entre a Tarefa-Mestra e as Tarefas-Escravas foi minimizada. Os algoritmos paralelos que usam essa nova abordagem de comunicação entre as Tarefas, são como segue:

Algoritmo 5.10 - Algoritmo da Tarefa-Mestra utilizando a segunda abordagem para a **Eliminação de Gauss e Fatoração LU.**

1. divide a matriz *A* em *k* blocos, cada bloco tendo *m* linhas
2. divide o vetor independente **b** em *k* partições, cada partição contendo *m* elementos
3. determina a primeira linha pivô, utilizando o pivoteamento parcial
4. criação e distribuição das *k* Tarefas-Escravas
5. **for** *i* = 1 **until** *k* **do**
6. envia a linha pivô para a Tarefa-Escrava *i*
7. envia o bloco *i* para a Tarefa-Escrava *i*
8. **endfor**
9. true = 1
10. **while**(true = 1) **do**
11. **for** *i* = 1 **until** *k* **do**
12. recebe da Tarefa-Escrava *i* informações sobre a linha do bloco *i* que é candidata a ser a nova linha pivô do próximo passo da eliminação
13. **endfor**
14. escolhe a nova linha pivô entre as candidatas
15. **for** *i* =1 **until** *k* **do**
16. envia para cada Tarefa-Escrava uma mensagem contendo a informação de qual tarefa escrava contém a nova linha pivô
17. **endfor**
18. recebe de uma das Tarefas-Escravas a nova linha pivô
19. recebe a linha pivô que foi usada na eliminação e a armazena na matriz *A*
20. **if** (for o último passo da eliminação) **then do**
21. encerra o processo de eliminação
22. mata as Tarefas-Escravas; true = 0
23. **endif**
24. **for** *i* = 1 **until** *k* **do**
25. envia a nova linha pivô para cada Tarefa-Escrava
26. **endfor**
27. **endwhile**
28. substituição regressiva;
29. **print** 'solução:', **x**

Algoritmo 5.11 - Algoritmo da Tarefa-Escrava utilizando a segunda abordagem para a **Eliminação de Gauss e Fatoração LU.**

```
1. recebe a linha pivô que será usada na primeira eliminação
2. recebe um bloco da matriz A que sofrerá a eliminação
3. true = 1
4. while ( true = 1) do
5.     aplica o processo de eliminação
6.     pesquisa no bloco pela linha que é candidata a ser a linha pivô para a próxima
       eliminação
7.     envia para a Tarefa-Mestra informações sobre a linha candidata a linha pivô
8.     recebe da Tarefa-Mestra o resultado da pesquisa
9.     if ( a linha pivô estiver nesta tarefa ) then do
10.        envia para a Tarefa-Mestra a linha pivô
11.        envia a linha que foi usada como linha pivô para ser armazenada em A
12.    endif
13.    if ( ainda tiver trabalho) do
14.       recebe a nova linha pivô que será usada na próxima eliminação
15.    else
16.       true = 0
17.    endif
18.endwhile
```

A Figura 5.3 ilustra o funcionamento dos algoritmos 5.10 e 5.11. Nela é mostrado como se dá a comunicação entre as tarefas, para os métodos diretos, utilizando a segunda abordagem comunicação. As setas indicam passagem de mensagens entre as tarefas.

A Tarefa-Mestre distribui a linha pivô e os blocos da matriz A entre as Tarefas-Escravas. Cada Tarefa-Escrava, recebe a linha pivô e um dos blocos da matriz A e entra em um *loop*. Neste *loop* ela aplica o método de Eliminação sobre os seus dados, e a linha que foi usada como linha pivô é enviada de volta para a Tarefa-Mestra, a fim de atualiza a matriz A. Em seguida, cada Tarefa-Escrava pesquisa no seu bloco de dados atualizados, pela candidata a linha pivô. Essas candidatas são enviadas à Tarefa-Mestra, que elege a nova linha pivô do próximo passo de Eliminação. Finalmente, a Tarefa-Mestra envia a nova linha pivô para cada uma das Tarefas-Escravas, que por sua vez, executam o próximo passo da Eliminação, sobre seus dados. Ao final do processo de Eliminação, a matriz A estará pronta para sofrer o processo de substituição progressiva e/ou regressiva.

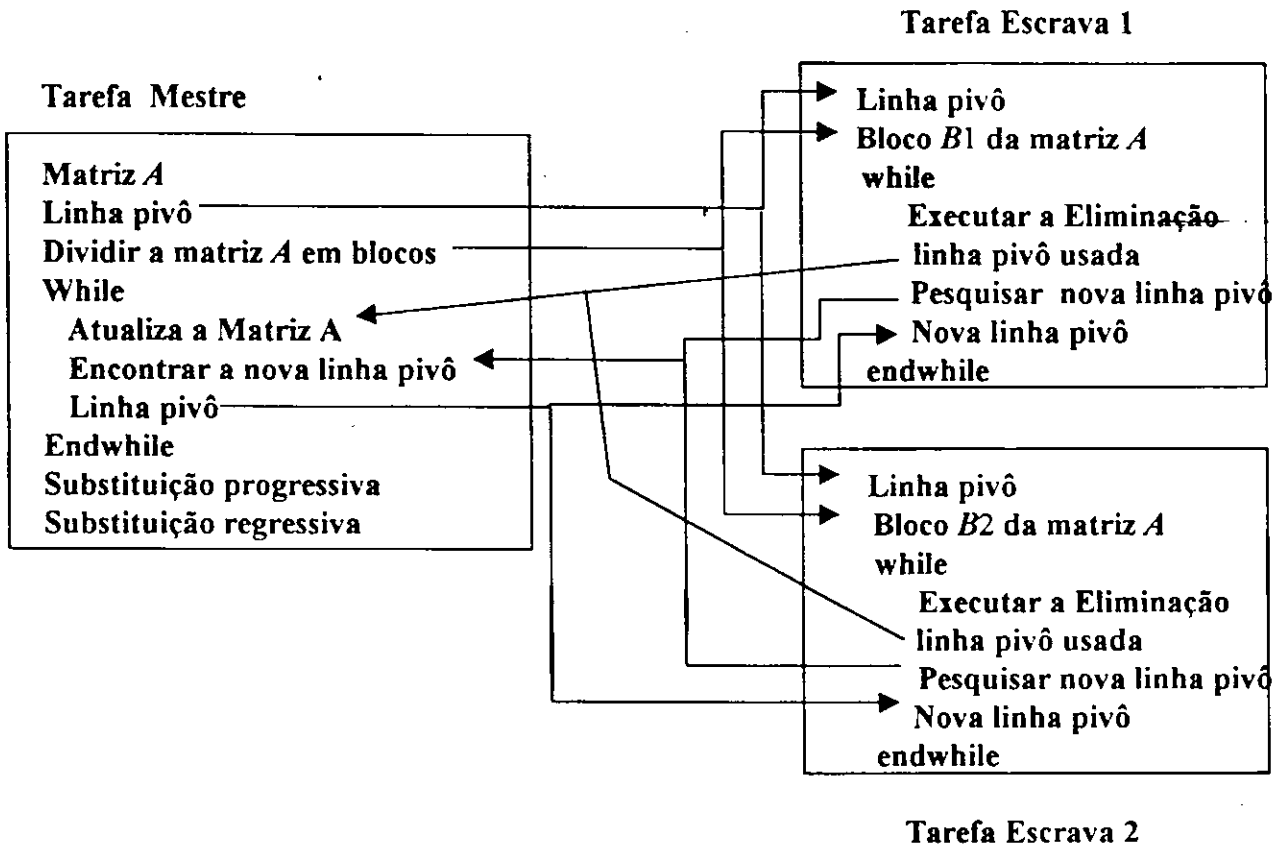

Figura 5.3 – Segunda abordagem de comunicação entre as tarefas – Métodos diretos

Algoritmo 5.12 - Algoritmo da Tarefa-Mestra utilizando a segunda abordagem para o método iterativo de **Gauss-Jacobi**.

```
1.  /* inicialização do vetor aproximação inicial */
2.  tolerância = 1.0D-12
3.  for i = 1 until  n do
4.     x( i ) = 0.0D0
5.  endfor
6.  for i = 1 until n do
7.        xold( i ) = x( i )
8.  endfor
9.  a matriz A é dividida em k blocos, cada bloco tendo m linhas
10. o vetor independente b é dividido em k partições, cada partição contendo m elementos
11. distribuição dos k blocos da matriz A
12. for i = 1 until k do
13.       envia o bloco i da matriz A para a Tarefa-Escrava i
14.       envia a partição i do vetor b para a tarefa i
15.       envia o vetor aproximação xold para a tarefa i
16. endfor
17. ite = 0
18. while ( ite ≤ maxit) do
19.       ite = ite + 1
20.       for i = 1 until k do
21.             recebe de cada Tarefa-Escrava uma partição do novo vetor aproximação x
22.       endfor
23.       /* essas partições com m elementos vão formar o novo vetor aproximação x */
          /* que deverá ter m × k elementos.                                        */
          /* faz o teste de parada                                                  */
24.       if ( teste_de_parada < tolerância ) then do
25.             exit
26.       else
27.             continue
28.       endif
29.       for i = 1 until n do
30.           xold( i ) = x ( i )
31.       endfor
32.       for i = 1 until k do
33.           envia para cada Tarefa-Escrava o novo vetor aproximação xold
34.       endfor
35. endwhile
36. for i = 1 until k do
37.       mata a Tarefa-Escrava i
38. endfor
39. print 'solução:', x
```

Algoritmo 5.13 - Algoritmo da Tarefa-Escrava utilizando a segunda abordagem para o método iterativo de **Gauss-Jacobi**.

1. recebe um dos bloco da matriz A, contendo m linhas
2. recebe uma partição do vetor **b** contendo m elementos
3. receber o vetor aproximação **xold**
4. true = 1
5. **while** (true = 1) **do**
6. /* cálculo das m componentes do próximo vetor aproximação */
7. aplica o método de Gauss-Jacobi, no bloco recebido, usando **xold**
8. envia a partição do novo vetor aproximação para a Tarefa-Mestra
9. recebe da Tarefa-Mestra o novo vetor aproximação **xold**
10. **endwhile**

A Figura 5.4 ilustra o funcionamento dos algoritmos 5.12 e 5.13. Ela mostra como ocorre a passagem de mensagens entre as tarefas, para o método de Gauss-Jacobi, utilizando a segunda abordagem de comunicação.

A Tarefa-Mestra, inicialmente, distribui para cada Tarefa-Escrava um dos blocos da matriz A, uma das partições do vetor **b** e o vetor aproximação. Em seguida, ela entra em um *loop* e espera para receber partições de vetor, vindas das Tarefas-Escravas. Essas partições vão compor o novo vetor aproximação, que é enviado de volta para cada uma das Tarefas-Escravas. A Tarefa-Mestra permanece neste *loop* até que um critério de parada seja satisfeito.

O trabalho da Tarefa-Escrava, é no inicio, receber um dos blocos da matriz A, uma das partições do vetor **b** e o vetor aproximação. Depois ela entra em *loop*, executando o método de Gauss-Jacobi. O resultado dessa operação é uma partição de vetor, que irá compor o novo vetor aproximação. Essa partição é então levada à Tarefa-Mestra, que se encarregará de receber as outras partições, vindas das outras Tarefas-Escravas, e finalmente irá compor o novo vetor aproximação. Esse novo vetor aproximação é mandado de volta para as Tarefas-Escravas, para que elas executem novamente o método de Gauss-Jacobi.

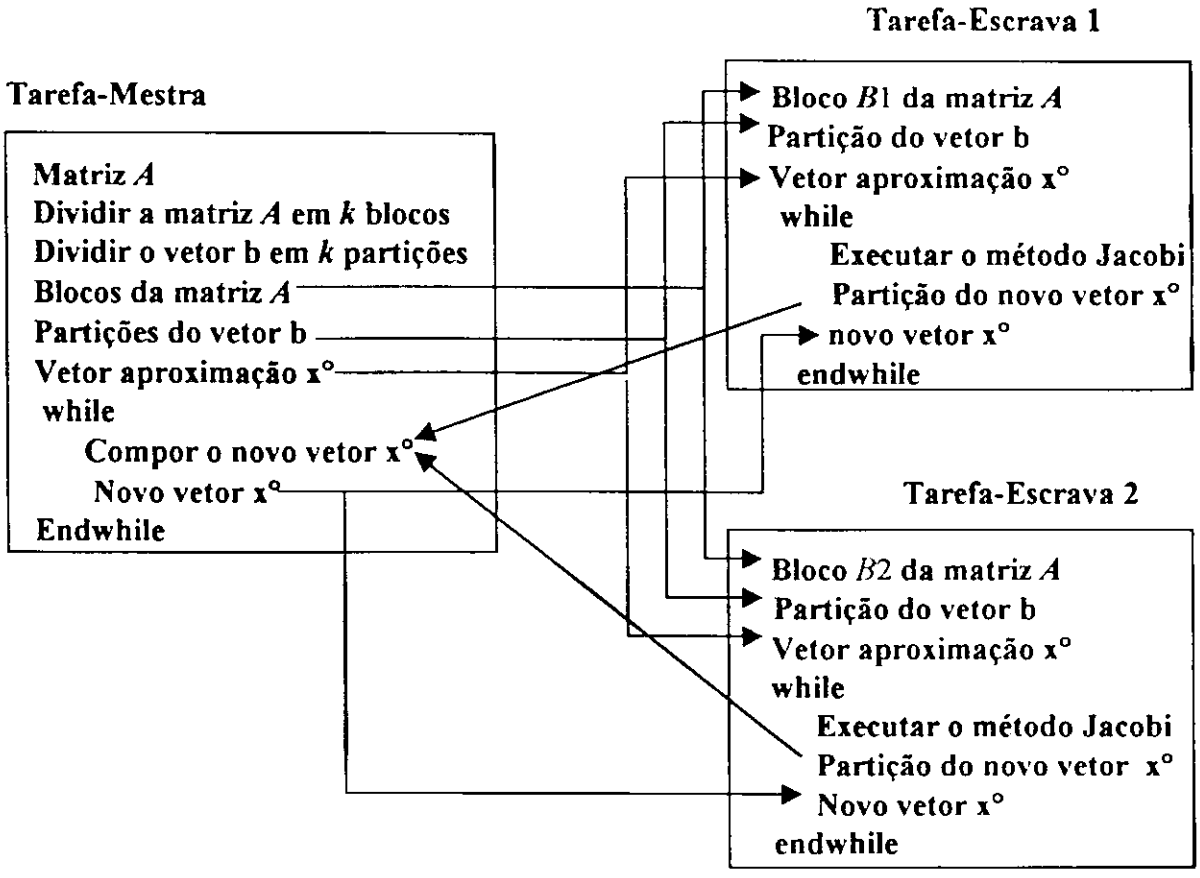

Figura 5.4 – Segunda abordagem de comunicação entre as tarefas para o método de Gauss-Jacobi

Algoritmo 5.14 - Algoritmo da Tarefa-Mestra, utilizando a segunda abordagem para o método iterativo dos **Gradientes Conjugados**. n é o número de equações e incógnitas do sistema linear. A matriz A deve ser simétrica positiva definida.

```
1.  tolerância = 1.0D-12
2.  maxit = 50
3.  for i = 1 until n do    /* vetor aproximação inicial */
4.     x( i ) = 0.0D0
5.  endfor
6.  r = b - Ax          /* computa o resíduo r = b - Ax */
7.  for i = 1 until n do
8.        p( i ) = r( i )
9.  endfor
10. c = produto_interno( r , r )
11. a matriz A é dividida em k blocos, cada bloco possuindo m linhas
12. distribuição dos k blocos da matriz A para as Tarefas-Escravas
13. for i = 1 until k do
14.        envia o bloco i para a Tarefa-Escrava i
15.        envia o vetor p para a Tarefa-Escrava i
16. endfor
17. for t = 1 until maxit do
18.        if ( sqrt(produto_interno( p , p )) < tolerância ) go to 44
19.        for i = 1 until k do
20.              recebe de cada Tarefa-Escrava uma partição do vetor q
21.        endfor
22.        α = c / produto_interno( p , q )
23.        for i = 1 until n do
24.              x( i ) = x( i ) + α × p( i )
25.        endfor
26.        for i = 1 until n do
27.             r( i ) = r( i ) - α × q( i )
28.        endfor
29.        d = produto_interno( r , r )
30.        for i = 1 until n do
31.                 p( i ) = r( i ) + (d / c ) × p( i )
32.        endfor
33.        c = d
34.        for i = 1 until k do
35.              envia para cada Tarefa-Escrava o novo vetor p
36.        endfor
37. endfor
38. continue
39. for i = 1 until k do
40.        mata a Tarefa-Escrava i
41. endfor
42. print 'A solução é:', x
43. print 'O resíduo é:', r
```

Algoritmo 5.15 - Algoritmo da Tarefa-Escrava utilizando a segunda abordagem para o método iterativo dos **Gradientes Conjugados**. n é o número de incógnitas de cada equação do sistema linear.

```
1. recebe um dos blocos da matriz A contendo m linhas
2. recebe o vetor p com n componentes
3.    /*   cálculo das m componentes do próximo vetor q  */
4. true = 1
5. while( true = 1) do
6.       for i = 1 until  m  do
7.              soma = 0.0D0
8.              for j = 1 until n do
9.                     soma = soma + (  a( i , j ) × p( j )  )
10.            endfor
11.            wq( i ) = soma
12.      endfor
13.      Envia o vetor wq com m componentes para a Tarefa-Mestra, que irá compor o vetor q
14.      recebe da Tarefa-Escrava o novo vetor p
15.endwhile
```

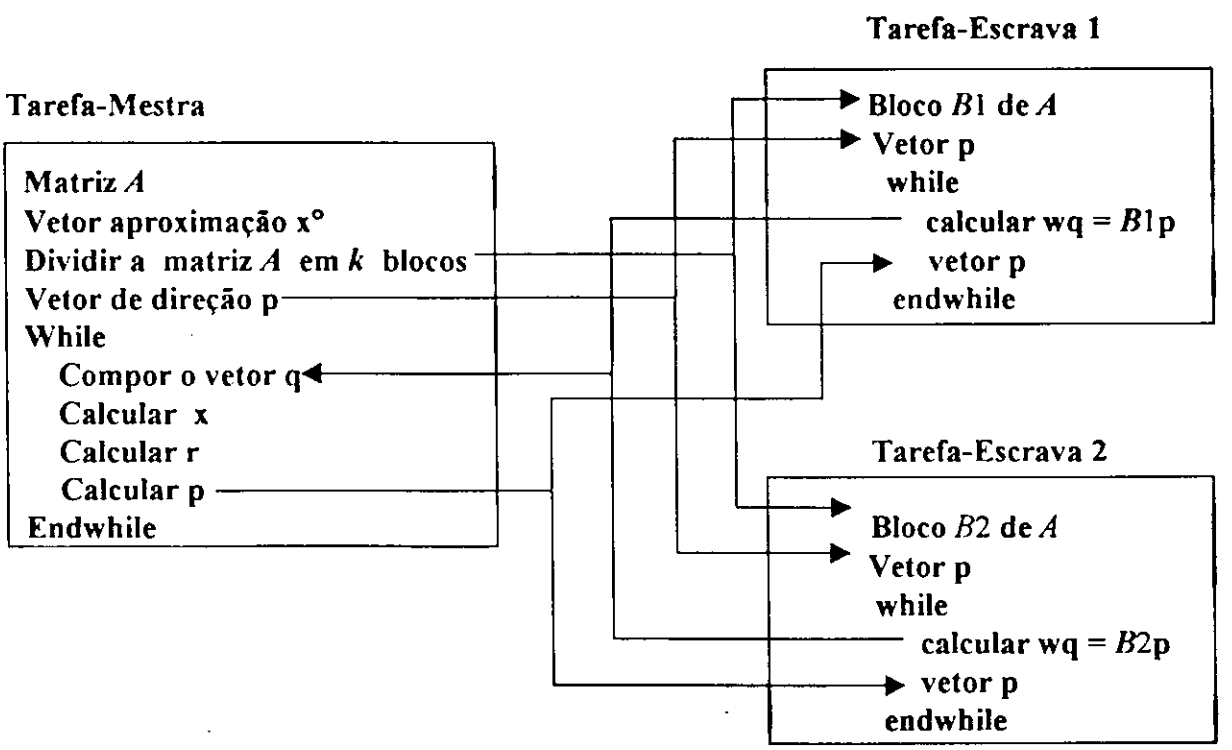

Figura 5.5 – Segunda abordagem de comunicação entre as tarefas para o método dos Gradientes Conjugados.

A Figura 5.5 ilustra o funcionamento dos algoritmos 5.14 e 5.15. Ela ilustra como ocorre a passagem de mensagens entre as tarefas, para o método dos Gradientes Conjugados, utilizando a segunda abordagem de comunicação.

Inicialmente, a Tarefa-Mestra, distribui para cada Tarefa-Escrava um dos blocos da matriz A e o vetor de direção **p**. Em seguida, ela entra em um *loop* e espera para receber partições de vetor, vindas das Tarefas-Escravas. Essas partições vão compor o vetor **q**, usado para calcular a próxima aproximação **x** e o resíduo correspondente. Depois, a Tarefa-Mestra faz um novo cálculo do vetor de direção **p** e este é enviado de volta para cada uma das Tarefas-Escravas. A Tarefa-Mestra permanece neste *loop* até que um critério de parada seja satisfeito.

O trabalho da Tarefa-Escrava, é no inicio, receber um dos blocos da matriz A e o vetor de direção **p**. Depois ela entra em *loop*, executando a multiplicação do bloco pelo vetor **p**. O resultado dessa operação é uma partição de vetor, que irá compor o novo vetor **q**. Essa partição é então levada à Tarefa-Mestra, que se encarregará de receber as outras partições, vindas das outras Tarefas-Escravas, e finalmente compor o novo vetor **q**.

5.5 - Resultados dos testes

Nesta seção, serão apresentados os resultado de testes de execução das rotinas serial e paralela, que implementam a Eliminação de Gauss, a Fatoração LU, o método iterativo de Gauss-Jacobi e o método iterativo dos Gradientes Conjugados.

5.5.1 - Resultados dos testes usando a primeira abordagem

As Tabelas 5.1, 5.2 e 5.3 mostram um resumo dos testes realizados utilizando a primeira abordagem. Nestas Tabelas, os melhores tempos produzidos pelo algoritmo paralelo de cada método (Eliminação de Gauss, Fatoração LU e o método iterativo de Gauss-Jacobi) são comparados com o tempo obtido pelo algoritmo serial para sistemas lineares de ordem 100, 200, 500 e 1.000 respectivamente. Não foram realizados testes com o algoritmo paralelo para o método iterativo dos Gradientes Conjugados, usando a abordagem inicial de comunicação entre as Tarefas, pois, os resultados dos testes com os algoritmos paralelos dos outros métodos, não foram satisfatórios.

Matriz	Melhor tempo computado pelo algoritmo paralelo, usando a primeira abordagem : T_{A_1}	Tempo computado pelo algoritmo serial : T_s	$\frac{T_{A_1}}{T_s}$
100×100	21,33 s	0,17 s	125,47
200×200	70,11 s	1,50 s	46,74
500×500	668,84 s	35,76 s	18,70
1.000×1.000	5.698,45 s	729,11 s	7,81
2.000×2.000	*	12.043,50 s	*

Tabela 5.1 - Comparação dos tempos paralelos obtidos usando a primeira abordagem e o tempo serial para a Eliminação de Gauss

Matriz	Melhor tempo computado pelo algoritmo paralelo, usando a primeira abordagem : T_{A_1}	Tempo computado pelo algoritmo serial : T_S	$\frac{T_{A_1}}{T_s}$
100×100	21,27 s	0,18 s	118,16
200×200	53,02 s	1,43 s	37,07
500×500	485,89 s	35,00 s	13,88
1.000×1.000	4.806,12 s	704,27 s	6,82
2.000×2.000	*	11.616,87 s	*

Tabela 5.2 - Comparação dos tempos paralelos obtidos usando a primeira abordagem e o tempo serial para a Fatoração LU

Matriz	Melhor tempo computado pelo algoritmo paralelo, usando a primeira abordagem : T_{A_1}	Tempo computado Pelo algoritmo serial : T_S	$\frac{T_{A_1}}{T_s}$
500×500	30,58 s	3,69 s	8,28
1.000×1.000	129,95 s	18,16 s	7,15
1.300×1.300	202,72 s	33,36 s	6,07
1.500×1.500	227,35 s	45,76 s	4,96
2.000×2.000	350,51 s	153,16 s	2,28
2.500×2.500	592,78 s	239,05 s	2,47
3.000×3.000	468,63 s	301,52 s	1,55

Tabela 5.3 - Comparação dos tempos paralelos obtidos usando a primeira abordagem e o tempo serial para o método iterativo de Gauss-Jacobi

As Tabelas 5.1, 5.2 e 5.3 mostram que a primeira abordagem, usada para a paralelização da Eliminação de Gauss, Fatoração LU e método iterativo de Gauss-Jacobi, não foi satisfatória. Para um sistema de ordem 2.000, os algoritmos paralelos dos métodos diretos não forneceram resposta, em virtude da passagem de parâmetros ser muito intensa. A razão entre o melhor tempo computado pelo algoritmo paralelo, (T_{A_1}). e o tempo computado pelo algoritmo serial para sistemas de ordem 100, 200, 500 e 1.000, mostra que o tempo paralelo é muito maior que o tempo serial (T_S).

Os resultados completos dos testes utilizando a primeira abordagem, são mostrados nas Tabelas 5.4 a 5.20. Inicialmente, foi investigado o desempenho dos algoritmos paralelos, em função do número de máquinas, do número de Tarefas-Escravas e do tamanho da matriz utilizada no processamento. Nas Tabelas 5.4 a 5.8 são mostrados os resultados produzidos pelos testes com a Eliminação de Gauss. As Tabelas 5.9 a 5.13 e 5.14 a 5.20 mostram os resultados produzidos pelos testes com a Fatoração LU e método iterativo de Gauss-Jacobi, respectivamente. Como referência, as Tabelas mostram o tempo do processamento serial para cada caso. Pode-se observar que os tempos de processamento paralelo, são sempre maiores que

os do processamento serial, independente da configuração (nº de máquinas × nº de Tarefas-Escravas × tamanho da matriz).

Os resultados obtidos, portanto, demonstraram a inviabilidade dos algoritmos usando a primeira abordagem, devido à forma como é feita a comunicação entre a Tarefa-Mestra e as Tarefas-Escravas.

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	32,62 s	24,34 s	21,33 s
6 Tarefas-Escravas	26,85 s	28,50 s	25,10 s
7 Tarefas-Escravas	52,87 s	37,88 s	30,14 s
8 Tarefas-Escravas	31,89 s	31,99 s	34,38 s
9 Tarefas-Escravas	31,71 s	35,38 s	31,69 s
10 Tarefas-Escravas	50,03 s	45,52 s	42,82 s
20 Tarefas-Escravas	90,34 s	76,78 s	64,42 s
Resíduo	0,11060516249550289 6E-04		
	Processamento Serial		
Tempo	0,17 s		
Resíduo	0,746069872548105195E-13		

Tabela 5.4 - Eliminação de Gauss - Matriz 100×100

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	103,72 s	105,14 s	92,21 s
6 Tarefas-Escravas	133,25 s	116,60 s	104,77 s
7 Tarefas-Escravas	84,34 s	82,85 s	70,11 s
8 Tarefas-Escravas	88,94 s	89,29 s	92,55 s
9 Tarefas-Escravas	92,93 s	88,80 s	89,27 s
10 Tarefas-Escravas	135,54 s	144,93 s	134,95 s
20 Tarefas-Escravas	221,28 s	191,35 s	181,19 s
Resíduo	0,7156496589733052 36E-04		
	Processamento Serial		
Tempo	1,5 s		
Resíduo	0,2700062395888380 71E-12		

Tabela 5.5 - Eliminação de Gauss - Matriz 200×200

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	1.078,59 s	1.055,01 s	948,46 s
6 Tarefas-Escravas	1.248,34 s	1.063,84 s	877,56 s
7 Tarefas-Escravas	1.454,08 s	1.157,70 s	802,73 s
8 Tarefas-Escravas	668,84 s	768,55 s	774,14 s
9 Tarefas-Escravas	1.061,41 s	1.080,33 s	1.096,83 s
10 Tarefas-Escravas	1.427,03 s	1.154,59 s	1.132,98 s
20 Tarefas-Escravas	1.519,66 s	1.564,82 s	1.317,34 s
Resíduo	0,3504198539623359 92E-03		
	Processamento Serial		
Tempo	35,76 s		
Resíduo	0,1605826582817826 42E-11		

Tabela 5.6 - Eliminação de Gauss - Matriz 500×500

	Processamento Paralelo
	7 máquinas
5 Tarefas-Escravas	8.735,38 s
6 Tarefas-Escravas	7.553,37 s
7 Tarefas-Escravas	5.698,45 s
8 Tarefas-Escravas	6.568,69 s
9 Tarefas-Escravas	6.507,02 s
10 Tarefas-Escravas	5.926,71 s
20 Tarefas-Escravas	6.785,22 s
Resíduo	0,106228066211422334E-02
	Processamento Serial
Tempo	729,11 s
Resíduo	0,765165708571657888E-11

Tabela 5.7 - Eliminação de Gauss - Matriz 1.000×1.000

	Processamento Paralelo
	7 máquinas
5 Tarefas-Escravas	*
7 Tarefas-Escravas	*
9 Tarefas-Escravas	*
20 Tarefas-Escravas	*
Resíduo	*
	Processamento Serial
Tempo	12.043,50 s
Resíduo	0,524380538990953937E-11

(*) - A rede ficou congestionada pelo tráfego intenso

Tabela 5.8 - Eliminação de Gauss - Matriz 2.000×2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	26,98 s	23,51 s	21,31 s
6 Tarefas-Escravas	46,26 s	46,10 s	21,27 s
7 Tarefas-Escravas	49,13 s	33,54 s	24,00 s
8 Tarefas-Escravas	31,91 s	30,33 s	30,98 s
9 Tarefas-Escravas	30,72 s	32,92 s	31,03 s
10 Tarefas-Escravas	45,50 s	41,87 s	38,39 s
20 Tarefas-Escravas	88,99 s	80,13 s	67,42 s
Resíduo	0,184830463389323851E-04		
	Processamento Serial		
Tempo	0,18 s		
Resíduo	0,488498130835068878E-13		

Tabela 5.9 - Fatoração LU - Matriz 100×100

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	85,48 s	73,17 s	68,21 s
6 Tarefas-Escravas	75,99 s	63,58 s	60,45 s
7 Tarefas-Escravas	84,02 s	73,96 s	53,02 s
8 Tarefas-Escravas	85,24 s	87,74 s	86,32 s
9 Tarefas-Escravas	85,35 s	90,28 s	92,77 s
10 Tarefas-Escravas	155,81 s	134,04 s	143,47 s
20 Tarefas-Escravas	255,52 s	239,76 s	223,92 s
Resíduo	0,44693726025002433 7E-04		
	Processamento Serial		
Tempo	1,43 s		
Resíduo	0,136335387423969223E-12		

Tabela 5.10.- Fatoração LU - Matriz 200×200

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	604,16 s	598,14 s	590,84 s
6 Tarefas-Escravas	677,02 s	492,99 s	485,89 s
7 Tarefas-Escravas	785,87 s	494,06 s	487,14 s
8 Tarefas-Escravas	617,76 s	564,25 s	540,20 s
9 Tarefas-Escravas	612,88 s	591,76 s	540,34 s
10 Tarefas-Escravas	723,19 s	538,21 s	466,67 s
20 Tarefas-Escravas	774,52 s	757,44 s	720,64 s
Resíduo	0,279305974634880272E-03		
	Processamento Serial		
Tempo	35 s		
Resíduo	0,390798504668055102E-12		

Tabela 5.11 - Fatoração LU - Matriz 500×500

	Processamento Paralelo
	7 máquinas
5 Tarefas-Escravas	7.783,32 s
6 Tarefas-Escravas	6.046,83 s
7 Tarefas-Escravas	4.806,12 s
8 Tarefas-Escravas	5.807,40 s
9 Tarefas-Escravas	6.065,65 s
10 Tarefas-Escravas	6.133,19 s
20 Tarefas-Escravas	6.497,63 s
Resíduo	0,104953488562475883E-02
	Processamento Serial
Tempo	704,27 s
Resíduo	0,130029320644098334E-11

Tabela 5.12 - Fatoração LU - Matriz 1.000×1.000

	Processamento Paralelo
	7 máquinas
6 Tarefas-Escravas	*
7 Tarefas-Escravas	*
8 Tarefas-Escravas	*
9 Tarefas-Escravas	*
10 Tarefas-Escravas	*
Resíduo	*
	Processamento Serial
Tempo	11.616,87 s
Resíduo	0,30269120543380267 9E-11

(*) - A rede ficou congestionada pelo tráfego intenso

Tabela 5.13 - Fatoração LU - Matriz 2.000x2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	45,55 s	43,35 s	30,58 s
7 Tarefas-Escravas	61,92 s	63,47 s	33,44 s
9 Tarefas-Escravas	73,12 s	66,81 s	53,28 s
10 Tarefas-Escravas	51,56 s	51,11 s	50,40 s
14 Tarefas-Escravas	73,87 s	60,39 s	40,11 s
21 Tarefas-Escravas	76,62 s	58,02 s	44,64 s
28 Tarefas-Escravas	75,43 s	67,85 s	50,96 s
Resíduo	0,00000000000000000E+00		
	Processamento Serial		
Tempo	3,69 s		
Resíduo	0,00000000000000000E+00		

Tabela 5.14 - Método Iterativo de Gauss-Jacobi - Matriz 500x500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	162,36 s	136,63 s	145,43 s
7 Tarefas-Escravas	227,50 s	225,80 s	129,95 s
9 Tarefas-Escravas	179,94 s	181,38 s	176,52 s
10 Tarefas-Escravas	159,05 s	157,34 s	159,86 s
14 Tarefas-Escravas	176,78 s	178,92 s	120,63 s
21 Tarefas-Escravas	214,51 s	164,32 s	131,19 s
28 Tarefas-Escravas	204,15 s	170,59 s	136,48 s
Resíduo	0,00000000000000000E+00		
	Processamento Serial		
Tempo	18,16 s		
Resíduo	0,00000000000000000E+00		

Tabela 5.15 - Método Iterativo de Gauss-Jacobi - Matriz 1.000x1.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	298,85 s	293,26 s	255,37 s
7 Tarefas-Escravas	389,95 s	240,49 s	249,06 s
9 Tarefas-Escravas	320,15 s	202,72 s	206,22 s
10 Tarefas-Escravas	286,62 s	293,78 s	291,65 s
14 Tarefas-Escravas	305,26 s	310,93 s	226,10 s
21 Tarefas-Escravas	285,65 s	226,46 s	229,34 s
28 Tarefas-Escravas	273,92 s	280,91 s	237,54 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	33, 36 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.16 - Método Iterativo de Gauss-Jacobi - Matriz 1.300×1.300

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	407,50 s	390,96 s	354,05 s
7 Tarefas-Escravas	506,79 s	499,71 s	297,79 s
9 Tarefas-Escravas	402,17 s	388,26 s	387,40 s
10 Tarefas-Escravas	352,94 s	328,76 s	319,18 s
14 Tarefas-Escravas	335,78 s	338,62 s	227,35 s
21 Tarefas-Escravas	376,91 s	314,92 s	239,44 s
28 Tarefas-Escravas	358,36 s	355,99 s	338,71 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	45,76 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.17 - Método Iterativo de Gauss-Jacobi - Matriz 1.500×1.500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	672,23 s	616,88 s	736,17 s
7 Tarefas-Escravas	516,07 s	472,70 s	448,24 s
9 Tarefas-Escravas	785,01 s	553,88 s	553,08 s
10 Tarefas-Escravas	550,62 s	527,24 s	350,51 s
14 Tarefas-Escravas	642,39 s	423,24 s	431,51 s
21 Tarefas-Escravas	621,83 s	437,61 s	450,95 s
28 Tarefas-Escravas	539,78 s	460,18 s	386,73 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	153,16 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.18 - Método Iterativo de Gauss-Jacobi - Matriz 2.000×2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	948,30 s	923,92 s	771,54 s
7 Tarefas-Escravas	1.112,89 s	1.103,27 s	710,69 s
9 Tarefas-Escravas	864,74 s	873,85 s	879,65 s
10 Tarefas-Escravas	810,99 s	786,37 s	791,59 s
14 Tarefas-Escravas	797,73 s	631,81 s	602,66 s
21 Tarefas-Escravas	789,93 s	603,15 s	621,17 s
28 Tarefas-Escravas	877,94 s	725,52 s	592,78 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	239,05 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.19 - Método Iterativo de Gauss-Jacobi - Matriz 2500×2500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	1.294,69 s	1.365,09 s	1.207,13 s
7 Tarefas-Escravas	873,97 s	795,74 s	669,33 s
9 Tarefas-Escravas	897,50 s	587,28 s	629,69 s
10 Tarefas-Escravas	724,26 s	630,13 s	606,05 s
14 Tarefas-Escravas	717,34 s	604,91 s	498,97 s
21 Tarefas-Escravas	588,28 s	469,63 s	474,99 s
28 Tarefas-Escravas	612,20 s	524,40 s	478,48 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	301,52 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.20 - Método Iterativo de Gauss-Jacobi - Matriz 3000×3000

5.5.2 - Resultados dos testes usando a segunda abordagem

Para obter um ganho significativo no desempenho dos algoritmos paralelos, para os métodos diretos e iterativos, decidimos minimizar a passagem de parâmetros entre a Tarefa-Mestra e as Tarefas-Escravas.

Na nova abordagem, os tempos computados pelos novos algoritmos paralelos caíram drasticamente. O resumo dos resultados obtidos é mostrado nas Tabelas 5.21 a 5.27.

Matriz	Melhor tempo computado Pelo algoritmo paralelo, usando a primeira abordagem : T_{A_1}	Melhor tempo computado pelo algoritmo paralelo, usando a segunda abordagem : T_{A_2}
100×100	21,33 s	3,68 s
200×200	70,11 s	7,92 s
500×500	668,84 s	49,84 s
1.000×1.000	5.698,45 s	119,49 s

Tabela 5.21 - Melhores tempos de cada abordagem para a Eliminação de Gauss

Na Tabela 5.21, vemos o quanto os tempos obtidos pelo algoritmo paralelo, utilizando a segunda abordagem, são melhores que os tempos obtidos pelo algoritmo paralelo, utilizando a primeira abordagem, para a Eliminação de Gauss.

Matriz	Melhor Tempo computado pelo algoritmo paralelo, usando a segunda abordagem: T_{A_2}	Nº de Tarefas Escravas	Nº de máquinas	Tempo computado pelo algoritmo serial T_s
100×100	3,68 s	5	7	0,17 s
200×200	7,92 s	5	6	1,50 s
500×500	49,84 s	7	5	35,76 s
600×600	62,95 s	10	7	91,22 s
700×700	43,39 s	5	7	174,34 s
900×900	96,57 s	8	6	455,85 s
1.000×1.000	119,49 s	6	6	729,11 s
1.500×1.500	324,15 s = 5,4 min	9	7	6.700,04 s = 1,8 h
2.000×2.000	1.058,80 s = 17,6 min	14	7	12.043,50 s = 3,3 h

Tabela 5.22 - Comparação entre os tempos paralelos obtidos pela segunda abordagem e os tempos seriais da Eliminação de Gauss.

A Tabela 5.22 mostra que, para sistemas lineares de ordem superior a 700, os tempos computados pelo algoritmo paralelo, utilizando a segunda abordagem, são melhores que os tempos do algoritmo serial para certas configurações (nº de Tarefas-Escravas × nº de máquinas). Para um sistema de ordem 2.000, o algoritmo serial leva em média 3,3 horas para resolvê-lo, enquanto que o algoritmo paralelo o resolve em 17,6 minutos.

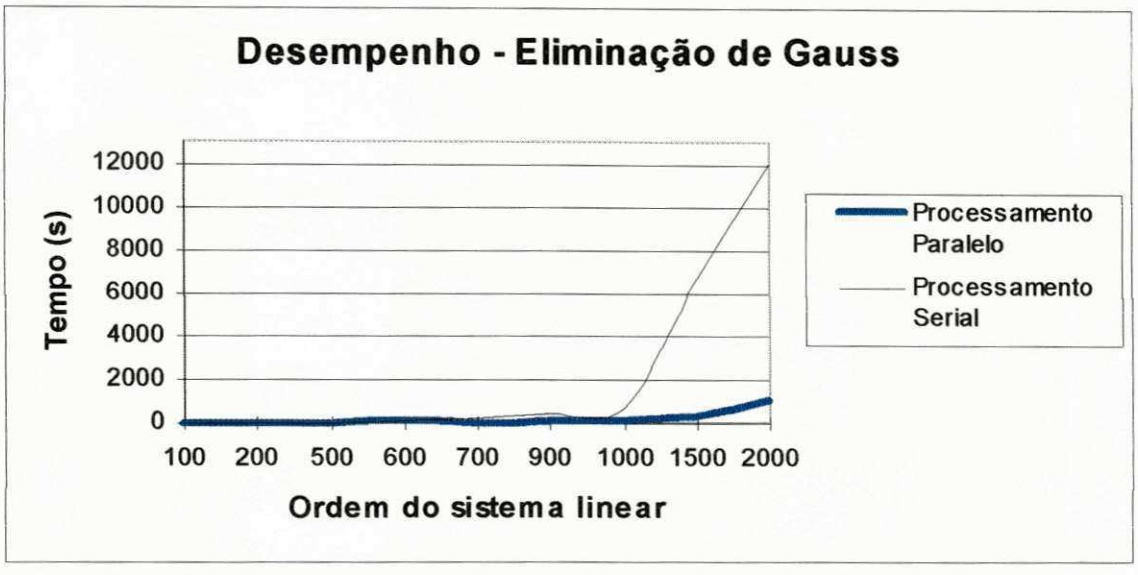

Figura 5.6 – Desempenho do algoritmo paralelo para sistemas de ordem até 2.000 – Eliminação de Gausss

Os resultados da Tabela 5.22 estão ilustrados no gráfico da Figura 5.6, onde fica claro a superioridade dos algoritmos paralelos para sistemas de ordem superior a 1.000.

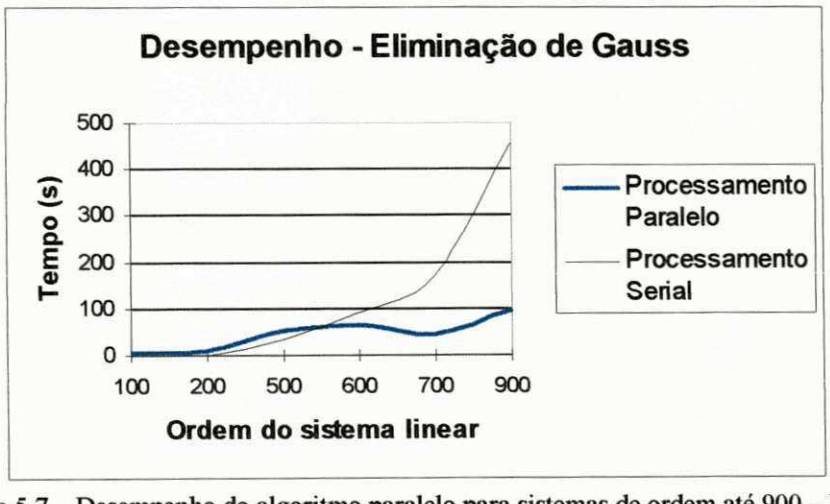

Figura 5.7 – Desempenho do algoritmo paralelo para sistemas de ordem até 900 – Eliminação de Gauss

A Figura 5.7 mostra mais detalhes dos resultados obtidos nos testes com sistemas lineares de ordem até 900. Veja que a partir de sistemas com ordem superior a 600, os tempos do algoritmo paralelo já são bem melhores que o tempo computado pelo algoritmo serial

Matriz	Melhor tempo computado Pelo algoritmo paralelo, Usando a primeira abordagem : T_{A_1}	Melhor tempo computado pelo algoritmo paralelo, usando a segunda abordagem : T_{A_2}
100×100	21,27 s	3,38 s
200×200	53,02 s	6,89 s
500×500	485,89 s	27,00 s
1.000×1.000	4.806,12 s	138,60 s

Tabela 5.23 - Melhores tempos de cada abordagem para a Fatoração LU

Na Tabela 5.23 vemos que os tempos obtidos pelo algoritmo paralelo, utilizando a segunda abordagem, são bastante melhores que os tempos obtidos pelo algoritmo paralelo, usando a primeira abordagem, para a Fatoração LU.

Matriz	Melhor Tempo computado pelo algoritmo paralelo, usando a segunda abordagem: T_{A_2}	Nº de Tarefas Escravas	Nº de máquinas	Tempo computado pelo algoritmo serial T_s
100×100	3,38 s	5	7	0,18 s
200×200	6,89 s	5	7	1,43 s
500×500	27,00 s	5	7	35,00 s
600×600	68,81 s	7	7	95,76 s
700×700	60,83 s	5	7	177,70 s
900×900	103,24 s	6	6	462,40 s
1.000×1.000	138,60 s	6	6	704,27 s
1.500×1.500	348,06 s = 5,8 min	10	6	7.748,90 s = 2,1 h
2.000×2.000	1.146,17 s = 19,1 min	14	6	11.616,87 s = 3,22 h

Tabela 5.24 - Comparação entre os tempos paralelos obtidos usando a segunda abordagem e os tempos seriais da Fatoração LU

A Tabela 5.24 mostra que para sistemas lineares de ordem superior a 500, os tempos computados pelo algoritmo paralelo, utilizando a segunda abordagem são melhores que os tempos do algoritmo serial para certas configurações (nº de Tarefas-Escravas × nº de máquinas). Para um sistema de ordem 2.000, o algoritmo serial leva, em média, 3,22 horas para resolvê-lo, enquanto que o algoritmo paralelo o resolve em 19,1 minutos.

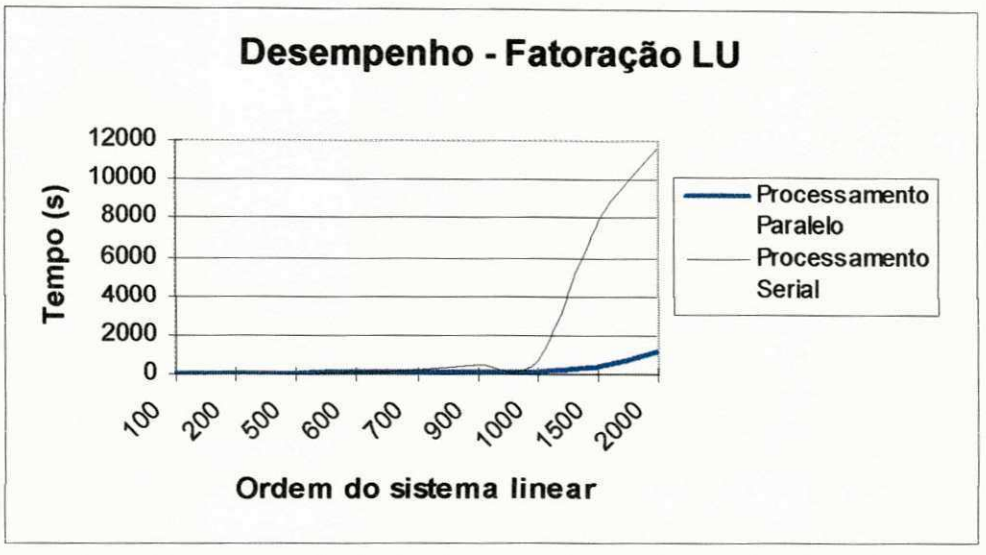

Figura 5.8 – Desempenho do algoritmo paralelo para sistemas de ordem até 2.000 – Fatoração LU

Os resultados da Tabela 5.24 estão ilustrados no gráfico da Figura 5.8, onde fica claro a superioridade dos algoritmos paralelos para sistemas de ordem superior a 1.000.

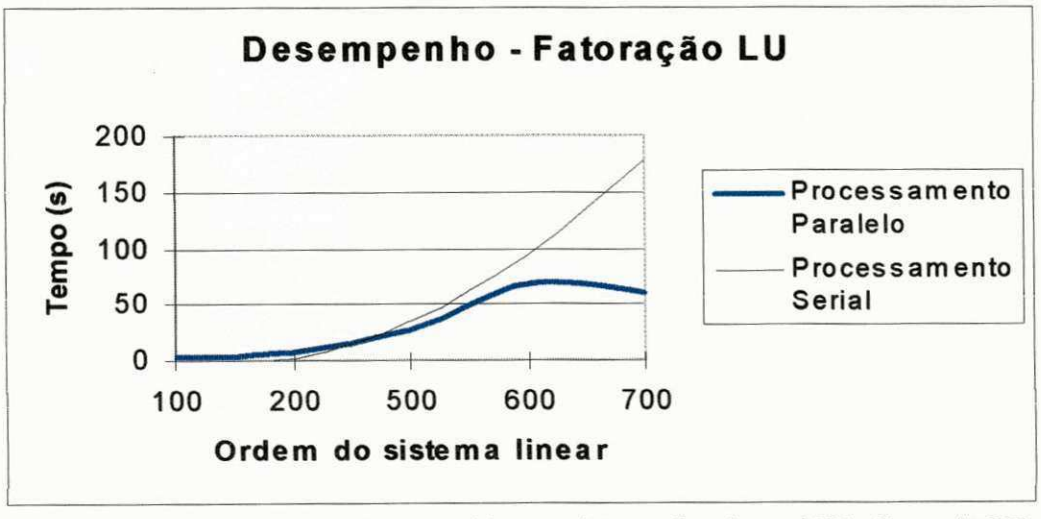

Figura 5.9 – Desempenho do algoritmo paralelo para sistemas de ordem até 700 - Fatoração LU

A Figura 5.9 mostra mais detalhes dos resultados obtidos nos testes com sistemas lineares de ordem até 700. A partir de sistemas com ordem superior a 500, os tempos do algoritmo paralelo já são bem melhores que o tempo computado pelo algoritmo serial

Matriz	Melhor tempo computado pelo algoritmo paralelo, usando a primeira abordagem : T_{A_1}	Melhor tempo computado pelo algoritmo paralelo, usando a segunda abordagem : T_{A_2}
500×500	30,58 s	3,39 s
1.000×1.000	129,95 s	18,23 s
1.300×1.300	202,72 s	37,04 s
1.500×1.500	227,35 s	52,42 s
2.000×2.000	350,51 s	141,99 s
2.500×2.500	592,78 s	202,98 s
3.000×3.000	474,99 s	186,44 s

Tabela 5.25 - Melhores tempos de cada abordagem para o método de Gauss-Jacobi

Mais uma vez a segunda abordagem se mostrou eficiente, quando aplicada ao método iterativo de Gauss-Jacobi, como é mostrado na Tabela 5.25. Nela, vemos que os tempos obtidos pelo algoritmo paralelo, usando a segunda abordagem, são bastantes melhores que os tempos obtidos pelo algoritmo paralelo, usando a primeira abordagem.

Matriz	Melhor Tempo computado pelo algoritmo paralelo usando a segunda abordagem: T_{A_2}	Nº de Tarefas Escravas	Nº de máquinas	Tempo computado pelo algoritmo serial T_s
500×500	3,39 s	6	5	36,90 s
1.000×1.000	18,23 s	6	7	18,16 s
1.300×1.300	37,04 s	21	7	33,36 s
1.500×1.500	52,42 s	35	7	45,76 s
2.000×2.000	141,99 s	42	6	153,16 s
2.500×2.500	202,98 s	42	6	239,05 s
3.000×3.000	186,44 s	35	6	301,52 s
4.000×4.000	831,70 s	10	7	3.593,43 s
5.000×5.000	1.772,82 s	9	7	6.216,99 s

Tabela 5.26 - Comparação entre os tempos paralelos obtidos usando a segunda abordagem e os tempos seriais do método iterativo de Gauss-Jacobi.

A Tabela 5.26 mostra que, para sistemas lineares de ordem superior a 2.000, os tempos computados pelo algoritmo paralelo, usando a segunda abordagem, são melhores que os tempos do algoritmo serial, para algumas configurações (n° de Tarefas-Escravas × n° de máquinas). Um sistema linear de ordem 5.000 leva, em média, 1,7 horas para ser resolvido pelo algoritmo serial (Gauss-Jacobi), enquanto que o algoritmo paralelo leva, em média, 29 minutos usando no processamento 9 Tarefas-Escravas e 7 máquinas.

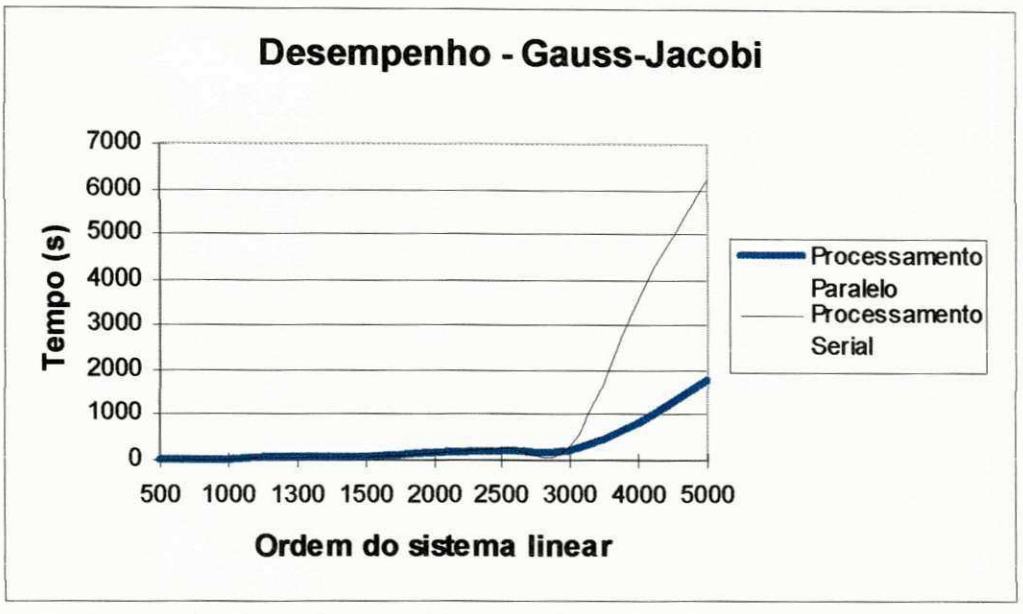

Figura 5.10 – Desempenho do algoritmo paralelo para sistemas de ordem até 5.000 – Gauss-Jacobi

A Figura 5.10 mostra claramente que o algoritmo paralelo para o método de Gauss-Jacobi é bastante eficiente na resolução de sistemas lineares de ordem superior a 3000.

Matriz	Melhor Tempo Computado pelo algoritmo paralelo usando a segunda abordagem: T_{A_2}	Nº de Tarefas Escravas	Nº de máquinas	Tempo computado pelo algoritmo serial T_S
500×500	3,45 s	5	6	2,87 s
1.000×1.000	40,00 s	14	7	40,17 s
2.000×2.000	92,35 s	42	7	201,29 s
3.000×3.000	122,63 s	35	7	475,75 s
4.000×4.000	617,34 s	21	7	2.671,68 s
5.000×5.000	1.222,25 s	35	7	6.882,99 s

Tabela 5.27 - Comparação entre os tempos paralelos obtidos usando a segunda abordagem e os tempos seriais do método iterativo dos Gradientes Conjugados.

A Tabela 5.27 mostra que, para sistemas lineares de ordem superior a 2.000, os tempos computados pelo algoritmo paralelo, usando a segunda abordagem, são melhores que os tempos do algoritmo serial, para o método iterativo dos Gradientes Conjugados.

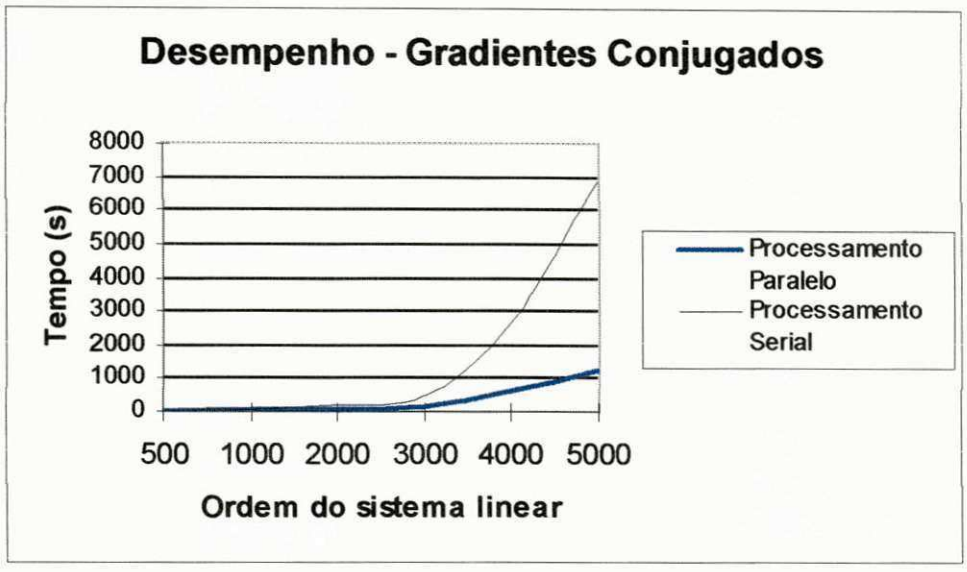

Figura 5.11 – Desempenho do algoritmo paralelo para sistemas de ordem até 5.000 - método dos Gradientes Conjugados.

Na Figura 5.11 vemos que o algoritmo paralelo para o método dos Gradientes Conjugados é bastante eficiente na resolução de sistemas lineares de ordem superior a 1.500.

Os resultados completos dos testes, usando a segunda abordagem, são mostrados nas Tabelas 5.28 a 5.60. Investigamos o desempenho dos algoritmos paralelos, em função do número de máquinas, do número de Tarefas-Escravas e do tamanho da matriz utilizada no processamento. Nas Tabelas 5.28 a 5.36, são mostrados os resultados produzidos pelos testes com a Eliminação de Gauss. As Tabelas 5.37 a 5.45 e 5.46 a 5.54, mostram os resultados produzidos pelos testes com a Fatoração LU e o método iterativo de Gauss-Jacobi, respectivamente. As Tabelas 5.55 a 5.60, mostram os resultados produzidos pelos testes com o método dos Gradientes Conjugados. Como referência, as Tabelas mostram o tempo do processamento serial, para cada caso.

Para os testes realizados com a **Eliminação de Gauss**, pode-se observar que os tempos obtidos pelo algoritmo paralelo, para sistemas de ordem 600, são menores que os tempos obtidos pelo algoritmo serial para algumas configurações (nº de máquinas × nº de Tarefas-Escravas). O mesmo se verifica para sistemas lineares de ordem 700 e 900. Para sistemas de ordem a partir de 1.000, os tempos obtidos pelo algoritmo paralelo são menores que os tempos obtidos pelo algoritmo serial, para todas as configurações (nº de máquinas × nº de Tarefas-Escravas) testadas, como é mostrado nas Tabelas 5.28 a 5.36.

Considerando os testes realizados com a **Fatoração LU**, pode-se observar que os tempos obtidos pelo algoritmo paralelo, para sistemas de ordem 500, são menores que os tempos obtidos

pelo algoritmo serial, para algumas configurações (nº de máquinas × nº de Tarefas-Escravas). O mesmo se verifica para sistemas de ordem 600, 700 e 900. Para sistemas de ordem a partir de 1.000, os tempos obtidos pelo algoritmo paralelo são menores que os tempos obtidos pelo algoritmo serial, para todas as configurações (nº de máquinas × nº de Tarefas-Escravas) testadas, como é mostrado nas Tabelas 5.37 a 5.45.

Já com relação aos testes realizados com o **método iterativo de Gauss-Jacobi**, pode-se observar que os tempos obtidos pelo algoritmo paralelo, para sistemas de ordem 2.000, são menores que os tempos obtidos pelo algoritmo serial, para algumas configurações (nº de máquinas × nº de Tarefas-Escravas). O mesmo se verifica para sistemas de ordem 2.500 e 3.000. Já para sistemas de ordem superior a 3.000, o ganho no desempenho do algoritmo paralelo é bastante razoável, como é mostrado nas Tabelas 5.46 a 5.54.

Considerando os testes realizados com o **método iterativo dos Gradientes Conjugados**, pode-se observar que os tempos obtidos pelo algoritmo paralelo, para sistemas de ordem 2.000, são menores que os tempos obtidos pelo algoritmo serial para algumas configurações (nº de máquinas × nº de Tarefas-Escravas). O mesmo se verifica para sistemas lineares de ordem 3.000. Para sistemas de ordem a partir de 4.000, os tempos obtidos pelo algoritmo paralelo são bastante menores que os tempos obtidos pelo algoritmo serial, para todas as configurações (nº de máquinas × nº de Tarefas-Escravas) testadas, como é mostrado nas Tabelas 5.55 a 5.60.

Os resultados obtidos , portanto, demonstraram a viabilidade dos algoritmos paralelos, usando a segunda abordagem.

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	5,30 s	5,06 s	3,68 s
6 Tarefas-Escravas	4,37 s	4,42 s	4,13 s
7 Tarefas-Escravas	4,72 s	4,71 s	6,57 s
8 Tarefas-Escravas	6,53 s	7,20 s	6,41 s
9 Tarefas-Escravas	6,63 s	7,23 s	6,75 s
10 Tarefas-Escravas	7,09 s	9,57 s	7,04 s
14 Tarefas-Escravas	8,78 s	8,84 s	7,82 s
21 Tarefas-Escravas	14,37 s	17,67 s	12,40 s
28 Tarefas-Escravas	18,12 s	15,82 s	15,44 s
Resíduo	0,11060516249550289 6E-04		
	Processamento Serial		
Tempo	0,17 s		
Resíduo	0,746069872548105195E-13		

Tabela 5.28 - Eliminação de Gauss - Matriz 100×100

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	13,62 s	7,92 s	11,07 s
6 Tarefas-Escravas	14,14 s	13,78 s	14,04 s
7 Tarefas-Escravas	23,54 s	15,03 s	14,60 s
8 Tarefas-Escravas	19,16 s	17,33 s	17,55 s
9 Tarefas-Escravas	21,59 s	22,62 s	20,44 s
10 Tarefas-Escravas	28,59 s	22,50 s	22,64 s
14 Tarefas-Escravas	30,83 s	32,37 s	30,83 s
21 Tarefas-Escravas	49,31 s	46,02 s	50,03 s
28 Tarefas-Escravas	66,45 s	64,01 s	61,53 s
Resíduo	0,7156496589733052 36E-04		
	Processamento Serial		
Tempo	1,5 s		
Resíduo	0,27000623958883807 1E-12		

Tabela 5.29 - Eliminação de Gauss - Matriz 200×200

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	93,47 s	96,25 s	64,11 s
6 Tarefas-Escravas	85,86 s	86,71 s	79,56 s
7 Tarefas-Escravas	49,84 s	72,62 s	76,63 s
8 Tarefas-Escravas	75,20 s	73,59 s	74,39 s
9 Tarefas-Escravas	95,08 s	99,06 s	95,39 s
10 Tarefas-Escravas	95,15 s	94,12 s	94,45 s
14 Tarefas-Escravas	110,72 s	110,75 s	98,41 s
21 Tarefas-Escravas	155,05 s	151,35 s	147,74 s
28 Tarefas-Escravas	197,82 s	194,90 s	199,11 s
Resíduo	0,3504198541719460 99E-03		
	Processamento Serial		
Tempo	35,76 s		
Resíduo	0,1605826582817826 42E-11		

Tabela 5.30 - Eliminação de Gauss - Matriz 500×500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	81,43 s	75,94 s	82,94 s
6 Tarefas-Escravas	81,11 s	73,98 s	74,61 s
7 Tarefas-Escravas	75,69 s	76,77 s	75,58 s
8 Tarefas-Escravas	78,67 s	66,54 s	64,74 s
9 Tarefas-Escravas	77,40 s	75,88 s	65,93 s
10 Tarefas-Escravas	72,45 s	71,56 s	62,95 s
14 Tarefas-Escravas	89,83 s	79,10 s	72,95 s
21 Tarefas-Escravas	119,18 s	106,37 s	99,60 s
28 Tarefas-Escravas	171,16 s	146,74 s	129,22 s
Resíduo	0,5655891396401102 61E-03		
	Processamento Serial		
Tempo	91,22 s		
Resíduo	0,7460698725481051 95E-12		

Tabela 5.31 - Eliminação de Gauss - Matriz 600×600

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	234,28 s	229,78 s	43,39 s
6 Tarefas-Escravas	271,14 s	251,06 s	54,37 s
7 Tarefas-Escravas	189,44 s	173,77 s	58,80 s
8 Tarefas-Escravas	160,51 s	151,09 s	59,94 s
9 Tarefas-Escravas	195,90 s	192,97 s	64,62 s
10 Tarefas-Escravas	190,12 s	189,22 s	73,14 s
14 Tarefas-Escravas	188,04 s	175,74 s	92,61 s
21 Tarefas-Escravas	230,80 s	202,23 s	124,14 s
28 Tarefas-Escravas	345,35 s	292,34 s	163,24 s
Resíduo	0,5451467635619877 68E–03		
	Processamento Serial		
Tempo	174,34 s		
Resíduo	0,8704148513061227 28E–12		

Tabela 5.32 - Eliminação de Gauss - Matriz 700×700

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	115,48 s	105,73 s	108,15 s
6 Tarefas-Escravas	112,19 s	139,42 s	103,11 s
7 Tarefas-Escravas	103,66 s	99,92 s	103,77 s
8 Tarefas-Escravas	114,87 s	96,57 s	100,51 s
9 Tarefas-Escravas	148,57 s	146,18 s	133,06 s
10 Tarefas-Escravas	151,45 s	120,34 s	108,57 s
14 Tarefas-Escravas	239,67 s	241,58 s	156,55 s
21 Tarefas-Escravas	328,32 s	306,74 s	196,47 s
28 Tarefas-Escravas	474,32 s	419,88 s	242,57 s
Resíduo	0,8855120746287248 32E–03		
	Processamento Serial		
Tempo	455,85 s		
Resíduo	0,1541877736599417 40E–11		

Tabela 5.33 - Eliminação de Gauss - Matriz 900×900

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	139,81 s	139,22 s	130,33 s
6 Tarefas-Escravas	124,71 s	119,49 s	122,53 s
7 Tarefas-Escravas	138,19 s	133,83 s	123,75 s
8 Tarefas-Escravas	139,29 s	132,92 s	127,95 s
9 Tarefas-Escravas	138,04 s	184,81 s	131,28 s
10 Tarefas-Escravas	188,70 s	183,47 s	181,79 s
14 Tarefas-Escravas	188,69 s	186,17 s	176,04 s
21 Tarefas-Escravas	263,21 s	287,34 s	232,02 s
28 Tarefas-Escravas	305,46 s	296,95 s	271,10 s
Resíduo	0,1062280661341341 58E–02		
	Processamento Serial		
Tempo	729,11 s		
Resíduo	0,7651657085716578 88E–11		

Tabela 5.34 - Eliminação de Gauss - Matriz 1.000×1.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	514,81 s	526,51 s	542,65 s
6 Tarefas-Escravas	682,34 s	433,50 s	451,83 s
7 Tarefas-Escravas	561,10 s	386,44 s	391,19 s
8 Tarefas-Escravas	499,57 s	338,74 s	351,73 s
9 Tarefas-Escravas	446,53 s	374,99 s	324,15 s
10 Tarefas-Escravas	405,77 s	432,36 s	449,94 s
14 Tarefas-Escravas	430,44 s	383,80 s	321,68 s
21 Tarefas-Escravas	558,11 s	482,31 s	383,22 s
28 Tarefas-Escravas	620,55 s	595,04 s	470,27 s
Resíduo	0,271727348012618108E-02		
	Processamento Serial		
Tempo	6.700,04 s		
Resíduo	0,36735059438797l796E-11		

Tabela 5.35 - Eliminação de Gauss - Matriz 1.500×1.500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	2.552,64 s	2.930,58 s	2.705,49 s
7 Tarefas-Escravas	2.636,19 s	1.838,26 s	1.595,85 s
9 Tarefas-Escravas	2.566,85 s	2.475,16 s	2.563,16 s
11 Tarefas-Escravas	1.842,48 s	2.203,00 s	2.765,39 s
14 Tarefas-Escravas	1.379,65 s	1.182,86 s	1.058,80 s
21 Tarefas-Escravas	1.249,46 s	1.217,93 s	1.069,98 s
28 Tarefas-Escravas	1.382,67 s	1.440,06 s	1.255,78 s
35 Tarefas-Escravas	1.621,92 s	1.861,50 s	1.916,73 s
42 Tarefas-Escravas	1.912,45 s	2.225,22 s	2.232,10 s
Resíduo	0,411590085764856894E-02		
	Processamento Serial		
Tempo	12.043,50 s		
Resíduo	0,5243805389909539 37E-11		

Tabela 5.36 - Eliminação de Gauss - Matriz 2.000×2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	6,45 s	4,35 s	3,38 s
6 Tarefas-Escravas	4,79 s	4,63 s	5,24 s
7 Tarefas-Escravas	5,59 s	4,88 s	5,50 s
8 Tarefas-Escravas	6,53 s	5,75 s	7,48 s
9 Tarefas-Escravas	6,86 s	6,36 s	6,10 s
10 Tarefas-Escravas	8,57 s	7,85 s	7,04 s
14 Tarefas-Escravas	10,12 s	10,06 s	12,27 s
21 Tarefas-Escravas	16,17 s	15,36 s	15,41 s
28 Tarefas-Escravas	18,71 s	19,68 s	17,60 s
Resíduo	0,110605162495502896E-04		
	Processamento Serial		
Tempo	0,18 s		
Resíduo	0,4884981308350688 78E-13		

Tabela 5.37 - Fatoração LU - Matriz 100×100

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	7,20 s	8,48 s	6,89 s
6 Tarefas-Escravas	8,33 s	7,99 s	8,31 s
7 Tarefas-Escravas	9,70 s	9,22 s	9,81 s
8 Tarefas-Escravas	11,38 s	10,79 s	10,48 s
9 Tarefas-Escravas	11,68 s	11,52 s	11,29 s
10 Tarefas-Escravas	13,18 s	13,01 s	12,83 s
14 Tarefas-Escravas	18,93 s	17,76 s	17,66 s
21 Tarefas-Escravas	26,46 s	25,96 s	27,02 s
28 Tarefas-Escravas	34,89 s	34,86 s	34,78 s
Resíduo	0,715649659075445754E-04		
	Processamento Serial		
Tempo	1,43 s		
Resíduo	0,136335387423969223E-12		

Tabela 5.38 - Fatoração LU - Matriz 200×200

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	30,47 s	29,93 s	27,00 s
6 Tarefas-Escravas	34,01 s	29,23 s	27,65 s
7 Tarefas-Escravas	35,06 s	35,13 s	32,18 s
8 Tarefas-Escravas	42,37 s	38,37 s	37,28 s
9 Tarefas-Escravas	43,55 s	40,51 s	40,73 s
10 Tarefas-Escravas	46,11 s	45,33 s	44,50 s
14 Tarefas-Escravas	62,70 s	59,92 s	57,23 s
21 Tarefas-Escravas	85,47 s	87,13 s	85,43 s
28 Tarefas-Escravas	109,44 s	110,99 s	114,47 s
Resíduo	0,350419853955230565E-03		
	Processamento Serial		
Tempo	35 s		
Resíduo	0,390798504668055102E-12		

Tabela 5.39 - Fatoração LU - Matriz 500×500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	82,02 s	78,32 s	75,60 s
6 Tarefas-Escravas	75,87 s	74,38 s	70,07 s
7 Tarefas-Escravas	79,32 s	69,36 s	68,81 s
8 Tarefas-Escravas	77,60 s	78,53 s	70,44 s
9 Tarefas-Escravas	78,22 s	81,04 s	76,40 s
10 Tarefas-Escravas	81,16 s	74,21 s	80,79 s
14 Tarefas-Escravas	111,33 s	108,70 s	90,79 s
21 Tarefas-Escravas	153,09 s	147,33 s	128,68 s
28 Tarefas-Escravas	215,74 s	199,96 s	177,38 s
Resíduo	0,5655891396667556l4E-03		
	Processamento Serial		
Tempo	95,76 s		
Resíduo	0,596855898038484156E-12		

Tabela 5.40 - Fatoração LU - Matriz 600×600

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	84,21 s	68,63 s	60,83 s
6 Tarefas-Escravas	64,66 s	63,55 s	71,00 s
7 Tarefas-Escravas	75,55 s	70,33 s	75,06 s
8 Tarefas-Escravas	93,52 s	79,77 s	77,58 s
9 Tarefas-Escravas	81,71 s	81,09 s	80,12 s
10 Tarefas-Escravas	89,44 s	83,41 s	85,35 s
14 Tarefas-Escravas	216,87 s	204,66 s	114,89 s
21 Tarefas-Escravas	283,35 s	241,26 s	161,16 s
28 Tarefas-Escravas	208,86 s	210,58 s	206,01 s
Resíduo	0,54514676351580249OE-03		
	Processamento Serial		
Tempo	177,7 s		
Resíduo	0,74606987254810519 5E-12		

Tabela 5.41 - Fatoração LU - Matriz 700×700

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	109,88 s	111,49 s	162,28 s
6 Tarefas-Escravas	157,08 s	103,24 s	104,80 s
7 Tarefas-Escravas	147,76 s	149,84 s	106,15 s
8 Tarefas-Escravas	202,93 s	170,76 s	138,79 s
9 Tarefas-Escravas	214,86 s	197,79 s	166,54 s
10 Tarefas-Escravas	147,78 s	141,21 s	162,32 s
14 Tarefas-Escravas	173,27 s	172,75 s	163,79 s
21 Tarefas-Escravas	247,80 s	223,91 s	226,61 s
28 Tarefas-Escravas	294,24 s	289,66 s	296,47 s
Resíduo	0,885512074644267955E-03		
	Processamento Serial		
Tempo	462,40 s		
Resíduo	0,142108547152020637E-11		

Tabela 5.42 - Fatoração LU - Matriz 900×900

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	154,05 s	147,14 s	164,34 s
6 Tarefas-Escravas	221,40 s	138,60 s	141,32 s
7 Tarefas-Escravas	148,20 s	151,69 s	145,22 s
8 Tarefas-Escravas	238,26 s	173,46 s	171,27 s
9 Tarefas-Escravas	227,10 s	169,34 s	173,03 s
10 Tarefas-Escravas	203,35 s	162,49 s	162,85 s
14 Tarefas-Escravas	224,62 s	221,87 s	198,65 s
21 Tarefas-Escravas	278,32 s	269,85 s	263,00 s
28 Tarefas-Escravas	388,48 s	413,70 s	373,80 s
Resíduo	0,106228066142638466E-02		
	Processamento Serial		
Tempo	704,27 s		
Resíduo	0,130029320644098334E-11		

Tabela 5.43 - Fatoração LU - Matriz 1.000×1.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	563,22 s	559,58 s	595,05 s
6 Tarefas-Escravas	575,65 s	489,90 s	510,88 s
7 Tarefas-Escravas	487,67 s	441,90 s	451,75 s
8 Tarefas-Escravas	570,33 s	399,13 s	421,71 s
9 Tarefas-Escravas	511,31 s	559,84 s	398,17 s
10 Tarefas-Escravas	478,47 s	421,58 s	348,06 s
14 Tarefas-Escravas	521,38 s	474,45 s	472,45 s
21 Tarefas-Escravas	603,48 s	685,15 s	644,01 s
28 Tarefas-Escravas	777,88 s	670,97 s	756,75 s
Resíduo	0,2717273480882909l0E-02		
	Processamento Serial		
Tempo	7.748,90 s		
Resíduo	0,187583282240666449E-11		

Tabela 5.44 - Fatoração LU - Matriz 1.500×1.500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	1.171,20 s	1.723,56 s	4.170,06 s
7 Tarefas-Escravas	1.543,02 s	1.398,25 s	3.375,26 s
9 Tarefas-Escravas	2.187,12 s	2.193,66 s	2.648,08 s
11 Tarefas-Escravas	1.358,49 s	2.308,07 s	2.238,36 s
14 Tarefas-Escravas	1.225,93 s	1.146,17 s	1.280,50 s
21 Tarefas-Escravas	1.473,34 s	1.358,18 s	1.430,42 s
28 Tarefas-Escravas	1.433,97 s	1.441,66 s	1.707,46 s
35 Tarefas-Escravas	2.445,46 s	2.310,63 s	2.412,82 s
42 Tarefas-Escravas	2.862,18 s	2.663,15 s	2.849,24 s
Resíduo	0,411590085733593014E-03		
	Processamento Serial		
Tempo	11.616,87 s		
Resíduo	0,3026912054338026 79E-11		

Tabela 5.45 - Fatoração LU - Matriz 2.000×2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	3,68 s	3,43 s	4,18 s
6 Tarefas-Escravas	3,39 s	3,50 s	4,61 s
7 Tarefas-Escravas	4,10 s	4,68 s	4,92 s
8 Tarefas-Escravas	4,33 s	4,04 s	4,45 s
9 Tarefas-Escravas	4,31 s	4,18 s	4,17 s
10 Tarefas-Escravas	5,66 s	5,83 s	5,17 s
14 Tarefas-Escravas	5,02 s	4,77 s	6,15 s
21 Tarefas-Escravas	6,27 s	6,02 s	5,94 s
28 Tarefas-Escravas	9,06 s	6,61 s	6,41 s
Resíduo	0,0000000000000000000E+00		
	Processamento Serial		
Tempo	3,69 s		
Resíduo	0,0000000000000000000E+00		

Tabela 5.46 - Método Iterativo de Gauss-Jacobi - Matriz 500×500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	27,32 s	22,50 s	20,23 s
6 Tarefas-Escravas	20,09 s	19,30 s	18,23 s
7 Tarefas-Escravas	19,52 s	20,41 s	18,73 s
8 Tarefas-Escravas	20,22 s	18,87 s	18,95 s
9 Tarefas-Escravas	19,85 s	18,85 s	18,77 s
10 Tarefas-Escravas	24,82 s	23,16 s	24,61 s
14 Tarefas-Escravas	24,82 s	23,58 s	21,62 s
21 Tarefas-Escravas	22,58 s	20,19 s	18,98 s
28 Tarefas-Escravas	31,54 s	21,47 s	22,90 s
Resíduo	0,00000000000000000E+00		
	Processamento Serial		
Tempo	18,16 s		
Resíduo	0,00000000000000000E+00		

Tabela 5.47 - Método Iterativo de Gauss-Jacobi - Matriz 1.000×1.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	78,93 s	80,24 s	70,45 s
6 Tarefas-Escravas	69,84 s	68,83 s	66,42 s
7 Tarefas-Escravas	61,60 s	59,25 s	56,89 s
8 Tarefas-Escravas	61,36 s	59,80 s	56,89 s
9 Tarefas-Escravas	60,18 s	57,04 s	55,54 s
10 Tarefas-Escravas	52,64 s	49,14 s	52,50 s
14 Tarefas-Escravas	52,68 s	46,45 s	41,33 s
21 Tarefas-Escravas	44,58 s	45,41 s	37,04 s
28 Tarefas-Escravas	48,77 s	43,38 s	37,62 s
Resíduo	0,00000000000000000E+00		
	Processamento Serial		
Tempo	33,36 s		
Resíduo	0,00000000000000000E+00		

Tabela 5.48 - Método Iterativo de Gauss-Jacobi - Matriz 1.300×1.300

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	111,72 s	117,08 s	105,32 s
6 Tarefas-Escravas	92,96 s	89,19 s	91,01 s
7 Tarefas-Escravas	92,01 s	78,34 s	83,06 s
8 Tarefas-Escravas	88,51 s	75,66 s	74,28 s
9 Tarefas-Escravas	76,30 s	76,38 s	66,94 s
10 Tarefas-Escravas	72,77 s	73,52 s	70,58 s
14 Tarefas-Escravas	69,61 s	62,21 s	59,18 s
21 Tarefas-Escravas	56,36 s	55,11 s	53,51 s
28 Tarefas-Escravas	60,29 s	54,68 s	53,73 s
35 Tarefas-Escravas	53,73 s	53,31 s	52,42 s
42 Tarefas-Escravas	57,15 s	55,01 s	56,69 s
Resíduo	0,00000000000000000E+00		
	Processamento Serial		
Tempo	45,76 s		
Resíduo	0,00000000000000000E+00		

Tabela 5.49 - Método Iterativo de Gauss-Jacobi - Matriz 1.500×1.500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	227,15 s	142,10 s	146,46 s
6 Tarefas-Escravas	207,29 s	211,17 s	214,04 s
7 Tarefas-Escravas	216,66 s	184,05 s	188,98 s
8 Tarefas-Escravas	176,65 s	203,45 s	177,28 s
9 Tarefas-Escravas	181,66 s	167,21 s	173,11 s
10 Tarefas-Escravas	177,51 s	181,12 s	196,15 s
14 Tarefas-Escravas	161,41 s	172,30 s	161,68 s
21 Tarefas-Escravas	149,95 s	145,78 s	144,31 s
28 Tarefas-Escravas	153,05 s	152,85 s	142,37 s
35 Tarefas-Escravas	148,92 s	144,23 s	142,47 s
42 Tarefas-Escravas	156,47 s	148,31 s	141,99 s
47 Tarefas-Escravas	161,94 s	149,09 s	145,75 s
Resíduo	0,00000000000000000000E-00		
	Processamento Serial		
Tempo	153,16 s		
Resíduo	0,00000000000000000000E-00		

Tabela 5.50 - Método Iterativo de Gauss-Jacobi - Matriz 2.000×2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	392,47 s	344,75 s	336,76 s
6 Tarefas-Escravas	330,83 s	276,67 s	275,10 s
7 Tarefas-Escravas	284,34 s	291,26 s	263,37 s
8 Tarefas-Escravas	258,67 s	256,91 s	262,60 s
9 Tarefas-Escravas	246,76 s	250,83 s	253,56 s
10 Tarefas-Escravas	240,48 s	239,06 s	236,19 s
14 Tarefas-Escravas	227,93 s	228,66 s	220,25 s
21 Tarefas-Escravas	221,19 s	213,49 s	211,24 s
28 Tarefas-Escravas	215,64 s	207,48 s	203,96 s
35 Tarefas-Escravas	218,72 s	209,63 s	206,63 s
42 Tarefas-Escravas	221,07 s	211,93 s	202,98 s
Resíduo	0,00000000000000000000E-00		
	Processamento Serial		
Tempo	239,05 s		
Resíduo	0,00000000000000000000E-00		

Tabela 5.51 - Método Iterativo de Gauss-Jacobi - Matriz 2.500×2.500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	599,24 s	515,33 s	1002,49 s
6 Tarefas-Escravas	691,97 s	625,21 s	360,52 s
7 Tarefas-Escravas	501,60 s	581,53 s	336,21 s
8 Tarefas-Escravas	398,18 s	404,76 s	398,38 s
9 Tarefas-Escravas	337,75 s	326,46 s	353,11 s
10 Tarefas-Escravas	349,97 s	285,86 s	287,48 s
14 Tarefas-Escravas	273,86 s	284,42 s	233,11 s
21 Tarefas-Escravas	261,27 s	206,91 s	216,23 s
28 Tarefas-Escravas	269,84 s	190,76 s	184,21 s
35 Tarefas-Escravas	238,67 s	186,44 s	195,22 s
42 Tarefas-Escravas	300,42 s	210,93 s	195,06 s
Resíduo	0,00000000000000000000E-00		
	Processamento Serial		
Tempo	301,52 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.52 - Método Iterativo de Gauss-Jacobi - Matriz 3.000×3.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	1.991,82 s	2.611,49 s	1.350,54 s
7 Tarefas-Escravas	2.079,30 s	1.954,08 s	1.359,85 s
9 Tarefas-Escravas	2.460,56 s	1.795,53 s	1.239,71 s
10 Tarefas-Escravas	1.416,52 s	1.526,05 s	831,70 s
14 Tarefas-Escravas	1.311,74 s	1.403,11 s	1.591,15 s
21 Tarefas-Escravas	1.300,38 s	1.577,99 s	1.612,27 s
28 Tarefas-Escravas	1.221,76 s	1.096,02 s	1.433,54 s
35 Tarefas-Escravas	1.553,82 s	1.496,39 s	1.384,62 s
42 Tarefas-Escravas	1.235,49 s	1.487,54 s	1.707,09 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	3.593,43 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.53 - Método Iterativo de Gauss-Jacobi - Matriz 4.000×4.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	2.734,01 s	3.240,01 s	3.701,85 s
7 Tarefas-Escravas	3.023,86 s	3.488,72 s	2.540,74 s
9 Tarefas-Escravas	2.572,48 s	2.984,57 s	1.772,82 s
10 Tarefas-Escravas	2.387,16 s	2.723,56 s	3.141,90 s
14 Tarefas-Escravas	2.318,39 s	2.748,95 s	2.481,23 s
21 Tarefas-Escravas	2.262,49 s	2.495,38 s	2.324,87 s
28 Tarefas-Escravas	2.999,50 s	2.096,80 s	2.367,09 s
35 Tarefas-Escravas	2.979,36 s	2.076,33 s	2.348,05 s
42 Tarefas-Escravas	2.430,99 s	2.293,94 s	2.321,99 s
Resíduo	0,00000000000000000000E+00		
	Processamento Serial		
Tempo	6.216,99 s		
Resíduo	0,00000000000000000000E+00		

Tabela 5.54 - Método Iterativo de Gauss-Jacobi - Matriz 5.000×5.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	3,81 s	3,45 s	4,22 s
7 Tarefas-Escravas	5,51 s	3,91 s	3,78 s
9 Tarefas-Escravas	5,13 s	5,03 s	3,74 s
10 Tarefas-Escravas	5,71 s	3,67 s	4,55 s
14 Tarefas-Escravas	5,88 s	4,99 s	4,20 s
21 Tarefas-Escravas	6,25 s	5,76 s	4,83 s
28 Tarefas-Escravas	7,27 s	6,17 s	5,68 s
35 Tarefas-Escravas	7,77 s	8,89 s	6,20 s
42 Tarefas-Escravas	9,36 s	7,45 s	6,96 s
Resíduo	0,46042054475763035 5E-13		
	Processamento Serial		
Tempo	2,87 s		
Resíduo	0,46042054475763035 5E-13		

Tabela 5.55 - Método dos Gradientes Conjugados - Matriz 500×500

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	78,4 s	75,5 s	77,1 s
7 Tarefas-Escravas	58,7 s	62,1 s	55,0 s
9 Tarefas-Escravas	49,0 s	48,1 s	46,4 s
10 Tarefas-Escravas	44,3 s	44,2 s	43,6 s
14 Tarefas-Escravas	58,9 s	42,9 s	40,0 s
21 Tarefas-Escravas	42,6 s	40,6 s	43,6 s
28 Tarefas-Escravas	48,4 s	46,0 s	43,5 s
35 Tarefas-Escravas	55,1 s	51,2 s	48,3 s
42 Tarefas-Escravas	59,7 s	56,3 s	54,8 s
Resíduo	0,81025163181671917 7E-13		
	Processamento Serial		
Tempo	40,17 s		
Resíduo	0,81025163181671917 7E-13		

Tabela 5.56 - Método dos Gradientes Conjugados - Matriz 1.000×1.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	220,69 s	195,78 s	201,18 s
7 Tarefas-Escravas	149,39 s	151,68 s	155,27 s
9 Tarefas-Escravas	132,64 s	127,65 s	111,57 s
10 Tarefas-Escravas	127,91 s	158,15 s	154,17 s
14 Tarefas-Escravas	149,27 s	174,52 s	165,60 s
21 Tarefas-Escravas	132,96 s	117,53 s	121,19 s
28 Tarefas-Escravas	115,94 s	116,45 s	108,05 s
35 Tarefas-Escravas	124,66 s	118,06 s	108,05 s
42 Tarefas-Escravas	129,81 s	96,05 s	92,35 s
Resíduo	0,428691799418971195E-13		
	Processamento Serial		
Tempo	201,29 s		
Resíduo	0,428691799418971195E-13		

Tabela 5.57 - Método dos Gradientes Conjugados - Matriz 2.000×2.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	519,98 s	587,18 s	485,27 s
7 Tarefas-Escravas	341,27 s	364,30 s	357,82 s
9 Tarefas-Escravas	287,64 s	266,21 s	274,82 s
10 Tarefas-Escravas	251,41 s	236,11 s	229,35 s
14 Tarefas-Escravas	206,15 s	183,97 s	181,39 s
21 Tarefas-Escravas	190,22 s	141,86 s	149,10 s
28 Tarefas-Escravas	176,87 s	143,79 s	139,35 s
35 Tarefas-Escravas	176,83 s	139,38 s	122,63 s
42 Tarefas-Escravas	171,83 s	147,42 s	131,85 s
Resíduo	0,104420576998773973E-12		
	Processamento Serial		
Tempo	475,75 s		
Resíduo	0,104420576998773973E-12		

Tabela 5.58 - Método dos Gradientes Conjugados - Matriz 3.000×3.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	1.315,62 s	1.308,00 s	908,14 s
7 Tarefas-Escravas	1.216,48 s	1.208,09 s	1.009,72 s
9 Tarefas-Escravas	884,57 s	943,36 s	928,57 s
10 Tarefas-Escravas	822,89 s	918,41 s	807,06 s
14 Tarefas-Escravas	721,10 s	769,93 s	666,84 s
21 Tarefas-Escravas	807,46 s	724,44 s	617,34 s
28 Tarefas-Escravas	689,66 s	727,00 s	618,43 s
35 Tarefas-Escravas	635,42 s	774,85 s	637,39 s
42 Tarefas-Escravas	662,22 s	707,47 s	632,15 s
Resíduo	0,538805926524021561E-13		
	Processamento Serial		
Tempo	2.671,68 s		
Resíduo	0,538805926524021561E-13		

Tabela 5.59 - Método dos Gradientes Conjugados - Matriz 4.000×4.000

	Processamento Paralelo		
	5 máquinas	6 máquinas	7 máquinas
5 Tarefas-Escravas	1.600,63 s	2.373,89 s	2.508,27 s
7 Tarefas-Escravas	2.779,25 s	1.975,60 s	1.995,72 s
9 Tarefas-Escravas	1.328,48 s	1.581,95 s	1.650,96 s
10 Tarefas-Escravas	1.387,84 s	1.789,01 s	1.639,93 s
14 Tarefas-Escravas	1.294,72 s	1.601,53 s	1.392,56 s
21 Tarefas-Escravas	1.318,35 s	1.279,22 s	1.290,51 s
28 Tarefas-Escravas	1.227,51 s	1.357,78 s	1.249,61 s
35 Tarefas-Escravas	1.250,09 s	1.324,54 s	1.222,25 s
42 Tarefas-Escravas	1.279,64 s	1.346,54 s	1.234,48 s
Resíduo	0.858167651402136165E-13		
	Processamento Serial		
Tempo	6.882,99 s		
Resíduo	0.858167651402136165E-13		

Tabela 5.60 - Método dos Gradientes Conjugados - Matriz 5.000×5.000

Para fazer uma comparação entre os dois métodos iterativos estudados realizamos testes com um sistema linear, cuja matriz dos coeficientes é SPD (Simétrica Positiva Definida) e estritamente diagonal dominante. Os resultados são mostrados nas Tabelas 5.61 e 5.62:

Nº de Equações / Método	500	1.000	2.000	3.000	4.000	5.000
Gauss-Jacobi	42	42	42	42	42	42
Gradientes Conjugados	32	33	33	33	34	34

Tabela 5.61 - Número de equações × Número de iterações - Matriz SPD estritamente diagonal dominante

	Resíduo - Processamento paralelo	
Ordem do Sistema	Gauss-Jacobi	Gradientes Conjugados
500	0,23464963305741548 5E–08	0,460420544757630355E–13
1.000	0,97934389486908 9127E–08	0,810251631816719177E–13
2.000	0,40090526454150 6767E–08	0,428691799418971195E–13
3.000	0,90803951025009 1553E–08	0,104420576998773973E–12
4.000	0,16210833564400 6729E–07	0,538805926524021561E–13
5.000	0,25363988243043 4227E–07	0,858167651402136165E–13

Tabela 5.62 - Número de equações x Resíduo - Matriz SPD estritamente diagonal dominante para os métodos iterativos

A Tabela 5.61 mostra a relação entre o número de equações e o número de iterações, para os métodos iterativos testados. São apresentados os números de iterações, para sistemas lineares de ordem 500, 1.000, 2.000, 3.000, 4.000 e 5.000. O valor de tolerância usado foi de 10^{-12}. O número de iterações realizadas para resolver um dado sistema linear com a matriz dos coeficientes SPD, estritamente diagonal dominante é menor no método dos Gradientes Conjugados, se comparado ao método iterativo de Gauss-Jacobi.

Já a Tabela 5.62 mostra a relação entre o número de equações e o resíduo, para os métodos iterativos testados. São apresentados os resíduos, para sistemas lineares de ordem 500, 1.000, 2.000, 3.000, 4.000 e 5.000. O valor de tolerância usado foi de 10^{-12}. A precisão obtida nos testes manteve a mesma grandeza.

	Resíduo - Processamento paralelo	
Ordem da Matriz	Eliminação de Gauss	Fatoração LU
100	0,11060516249550 2896E–04	0,11060516249550 2896E–04
200	0,71564965897330 5236E–04	0,71564965907544 5754E–04
500	0,35041985417194 6099E–03	0,35041985395523 0565E–03
600	0,56558913964011 0261E–03	0,59685589803848 4156E–03
700	0,54514676356198 7768E–03	0,54514676351580 2490E–03
900	0,88551207462872 4832E–03	0,14210854715202 0637E–03
1.000	0,10622806613413 4158E–02	0,13002932064409 8334E–02
1.500	0,27172734801261 8108E–02	0,27172734808829 0910E–02
2.000	0,41159008576485 6894E–02	0,41159008573359 3014E–02

Tabela 5.63 - Resíduos obtidos nos testes com os métodos diretos

A precisão obtida nos testes com os algoritmos paralelos para os métodos diretos, foi decrescendo, à medida que a ordem do sistema linear cresceu. Como mostrado na Tabela 5.63.

Considerando o método de **Eliminação de Gauss**, os testes mostraram que as configurações (n° de Tarefas-Escravas × n° de máquinas) que produziram o melhor desempenho do algoritmo paralelo, para os valores de n testados, são mostradas na Tabela 5.64. Esta Tabela mostra que, na maioria dos casos, o melhor tempo é obtido com 7 ou 6 máquinas e que para sistemas lineares de ordem n = 600, 700, 900, 1.000, 1.500, 2.000, o número ótimo de Tarefas-Escravas deve ser 10, 5, 8, 6, 9 e 14, respectivamente.

Ordem do sistema	n° de Tarefas-Escravas	n° de máquinas
600	10	7
700	5	7
900	8	6
1.000	6	6
1.500	9	7
2.000	14	7

Tabela 5.64 - Configuração dos melhores resultados obtidos para a Eliminação de Gauss.

Para a **Fatoração LU**, as configurações (n° de Tarefas-Escravas × n° de máquinas) que produziram o melhor desempenho do algoritmo paralelo, para os valores de n testados, são mostradas na Tabela 5.65. O número de máquinas deve ser 6 ou 7. O número ótimo de Tarefas-Escravas para sistemas lineares de ordem n = 500, 600, 700, 900, 1.000, 1.500, 2.000 deve ser 5, 7, 5, 6, 6, 10 e 14, respectivamente.

Ordem do Sistema	n° de Tarefas-Escravas	n° de máquinas
500	5	7
600	7	7
700	5	7
900	6	6
1.000	6	6
1.500	10	7
2.000	14	6

Tabela 5.65 - Configuração dos melhores resultados para a Fatoração LU.

Para o **método iterativo de Gauss-Jacobi**, as configurações (n° de Tarefas-Escravas × n° de máquinas) que produziram o melhor desempenho do algoritmo paralelo, são mostradas na Tabela 5.66. O número de máquinas deve ser 6 ou 7, dependendo do valor de n. O número ótimo de Tarefas-Escravas para sistemas lineares de ordem n = 2.000, 2.500, 3.000, 4.000, 5.000 deve ser de 42, 42, 35, 10 e 9, respectivamente.

Ordem do Sistema	n° de Tarefas-Escravas	n° de máquinas
2.000	42	6
2.500	42	6
3.000	35	6
4.000	10	7
5.000	9	7

Tabela 5.66 - Configuração dos melhores resultados para o método iterativo de Gauss-Jacobi.

Considerando o **método iterativo dos gradientes conjugados**, as configurações (nº de Tarefas-Escravas × nº de máquinas) que produziram o melhor desempenho do algoritmo paralelo, para os valores de n testados, são mostradas na Tabela 5.67. O número de máquinas deve ser 7. O número ótimo de Tarefas-Escravas para sistemas lineares de ordem n = 2.000, 3.000, 4.000, 5.000, deve ser de 14, 42, 35, 21 e 35, respectivamente.

Ordem do Sistema	nº de Tarefas-Escravas	Nº de máquinas
1.000	14	7
2.000	42	7
3.000	35	7
4.000	21	7
5.000	35	7

Tabela 5.67 - Configuração dos melhores resultados para o método iterativo dos gradientes conjugados.

5.6 - Resumo

Este capitulo descreveu as matrizes usadas nos testes com os métodos para resolução de sistemas lineares; o algoritmo que gerou essas matrizes; os algoritmos paralelos da primeira e segunda abordagem; os resultado dos testes dos algoritmos da primeira e segunda abordagem.

Foi feita também uma comparação entre os dois métodos iterativos. A precisão dos métodos iterativos foi comparada com a precisão dos métodos diretos.

Capítulo 6

Conclusões

6.1 - Introdução

O presente trabalho teve por objetivo investigar a viabilidade da utilização, em máquinas paralelas, de métodos clássicos de solução de sistemas lineares de grande porte. Para tanto, foi realizada uma análise comparativa entre dois tipos de métodos: métodos diretos, representados pela Eliminação de Gauss e Fatoração LU, e métodos iterativos, representados pelos métodos de Gauss-Jacobi e o dos Gradientes Conjugados.

Os algoritmos paralelos foram implementados nas estações IBM PowerPC do Laboratório de Computação (LabCom) da Universidade Federal da Paraíba - UFPB, Campus II, em Campina Grande e processados em uma rede de computadores instalada no CENAPAD-NE (Centro Nacional de Processamento de Alto Desempenho no Nordeste), em Fortaleza. A rede dispõe dos seguintes equipamentos, baseados no sistema IBM™ RS/6000 Scalable POWERparallel System (SP) da plataforma IBM RISC System/6000:

- Um (1) IBM RISC6000, modelo 590 Scalable POWERparallel System (SP2), com 4 processadores RISC super-escalares de arquitetura POWER2, interconectados por um SWITCH de alta velocidade, 256 MB de memória RAM por nó, 2 GB de capacidade de armazenamento em disco rígido (SCSI-2), 256 bit no barramento de memória e sistema operacional AIX v.3.2.5. A taxa de transferência de dados, ponto-a-ponto, é de 320 Mb/s e a Latência de hardware é de 500 ns. Cada nó tem desempenho máximo de 266 MFLOPS, fazendo com que o SP2 atinja um desempenho total de 1 GFLOPS.
- Um (1) IBM RISC6000, modelo 7011/250, processador RISC POWER PC 601-66 MHz ligado ao SP2, operando como estação de controle, sistema operacional AIX v.3.2.5, 64 MB de memória RAM, 2 GB de capacidade de armazenamento em disco rígido (SCSI-2) e 64 bit no barramento de memória.
- Dois (2) IBM RISC6000, modelo 7013/590 processador RISC POWER2 66 MHz operando como servidores de arquivos, sistema operacional AIX v.3.2.5, 128 MB de

memória RAM, 20 GB de capacidade de armazenamento em disco rígido (SCSI-2) e 256 bit no barramento de memória. Eles estão interligados ao SP2, por uma rede FDDI.

A rede utiliza o software para computação paralela chamado PVM, ou Máquina Virtual Paralela (*Parallel Virtual Machine*). Uma máquina virtual paralela é um conjunto de computadores conectados por uma rede de comunicação, que trabalham cooperativamente para resolver um "grande" problema computacional. O modelo de programação usado pelo PVM é o Mestre-Escravo, onde um programa de controle (mestre) é responsável pela criação de Tarefas-Escravas e pela coleta dos resultados das mesmas. As Tarefas-Escravas realizam os cálculos propriamente ditos.

Para a transferência e o processamento dos algoritmos, foram usados dois serviços da INTENET: o FTP e o TELNET. O FTP é um protocolo usado para a transferência de arquivos entre computadores. O TELNET é um serviço que permite a um usuário entrar em uma outra máquina ligada à Internet, transformando a máquina local em um terminal da máquina remota. Vários testes com os programas paralelos foram processados a partir dos terminais do Laboratório de Automação e Processamento de Sinais (LAPS) do Departamento de Engenharia Elétrica da UFPB.

Este trabalho foi pioneiro, no sentido de mostrar a viabilidade de um processamento de alto desempenho à distância. Essa utilização foi feita dentro da filosofia para a qual o CENAPAD-NE foi criado, isto é, de ser um centro regional que pudesse ser usado por usuários de outras localidades, o que se verificou no presente caso.

6.2 - Avaliação comparativa dos métodos

Os testes com os métodos diretos foram realizados, usando sistemas lineares para os quais os elementos da matriz dos coeficientes foram gerados aleatoriamente. Já os testes com o método iterativo de Gauss-Jacobi foram realizados, inicialmente, usando sistemas lineares cuja matriz dos coeficientes era estritamente diagonal dominante e posteriormente, visando uma comparação dos resultados com aqueles obtidos com o método dos Gradientes Conjugados, foram realizados testes com método de Gauss-Jacobi utilizando sistemas lineares com matriz simétrica positiva definida estritamente diagonal dominante.

6.2.1 – Minimização de passagem de parâmetros

Os primeiros algoritmos paralelos que foram implementados não visavam minimizar a passagem de parâmetros entre a Tarefa-Mestra e as Tarefas-Escravas. Foi constatado que os tempos obtidos com esses algoritmos foram extremamente altos, em comparação com os tempos

obtidos com o algoritmo serial. A primeira tentativa que se fez para resolver esse problema foi trabalhar com a transposta da matriz original, reescrevendo o código dos algoritmos para que fizessem as operações nas colunas da matriz transposta, visando reduzir o tempo gasto pelos programas, em FORTRAN, no acesso à memória da máquina. Entretanto, essa tentativa não resultou numa melhoria significativa nos tempos obtidos pelos algoritmos paralelos.

Em seguida, na busca da melhoria dos tempos, foi usada uma nova abordagem na comunicação entre as Tarefas-Escravas e a Tarefa- Mestra. Nessa abordagem, a carga de trabalho é distribuída para as Tarefas-Escravas e a partir daí, a passagem de parâmetros entre as Tarefas Escravas e a Tarefa Mestre é minimizada. Como esperado, verificou-se que o bom desempenho dos programas paralelos depende da maneira como é feita a comunicação entre a Tarefa-Mestra e as Tarefas-Escravas (Paradigma de Programação Mestre-Escravo). A passagem de parâmetros entre as Tarefas deve ser a menor possível, a fim de reduzir o *overhead* devido à comunicação.

No caso da Eliminação de Gauss e Fatoração LU, após a distribuição dos blocos da matriz A para as Tarefas-Escravas, os dados que trafegam pela rede correspondem apenas à nova linha pivô que será usada no próximo passo da eliminação e a um vetor que vai atualizando a matriz A (que está armazenada na Tarefa-Mestra). Ao final do processamento paralelo, a matriz A estará pronta para ser submetida aos processos de substituição (progressiva e/ou regressiva).

No caso do métodos iterativos estudados, após a distribuição dos blocos da matriz A para as Tarefas-Escravas, o tráfego pela rede é composto, unicamente, por um vetor de n números reais.

Desse modo, constatamos que o maior gargalo da execução de programas paralelos é a comunicação entre as tarefas cooperantes. Portanto, esta comunicação deve ser minimizada para evitar uma queda de despenho na execução dos algoritmos paralelos.

6.2.2 – Síntese dos resultados

Na resolução de sistemas lineares não singulares $A\mathbf{x} = \mathbf{b}$, usando os algoritmos paralelos foi verificado que para sistemas lineares de ordem n:

- Para n = 100, 200, 500, para os métodos diretos; n = 500, 1.000, 1.300, 1.500, para o método de Gauss-Jacobi e n = 500, 1.000, para o método dos Gradientes Conjugados, não houve ganho no desempenho, em comparação com os tempos obtidos pelo algoritmo serial.
- Para n = 600, 700, 900 para os métodos diretos; n = 2.000, 2.500, 3.000, para o método de Gauss-Jacobi e n = 2.000, 3.000, para o método dos Gradientes

Conjugados, houve um ganho razoável no desempenho, somente para algumas configurações (nº de Tarefas-Escravas × nº de máquinas).

- Para n = 1.000, 1.500, 2.000, para os métodos diretos; n = 4.000, 5.000, para o método de Gauss-Jacobi e n = 4.000, 5.000, para o método dos Gradientes Conjugados, houve ganho de desempenho bastante significativo.

Foi verificado que a precisão obtida nos testes com os algoritmos paralelos, para os métodos diretos, decresce à medida que a ordem do sistema linear aumenta. Já a precisão obtida nos testes com os algoritmos paralelos, para os métodos iterativos, manteve a mesma grandeza, para todos os valores de ordem de sistemas testados.

Os testes mostraram que o melhor desempenho dos algoritmos paralelos, para cada método estudado, é obtido com o número de máquinas m:

- m = 6 ou m = 7, para a Eliminação de Gauss.
- m = 6 ou m = 7, para a Fatoração LU.
- m = 6 ou m = 7, para o método de Gauss-Jacobi.
- m = 7, para o método dos Gradientes Conjugados.

O número de Tarefas-Escravas que produziu o melhor desempenho para os métodos estudados foi:

- Eliminação de Gauss: para n = 600, 700, 900, 1.000, 1.500 e 2.000, o número ótimo de Tarefas-Escravas foi 10, 5, 8, 6, 9 e 14, respectivamente.
- Fatoração LU: para n = 500, 600, 700, 900, 1.000, 1.500 e 2.000, o número ótimo de Tarefas-Escravas foi 5, 7, 5, 6, 6, 10 e 14, respectivamente.
- Método de Gauss-Jacobi: para n = 2.000, 2.500, 3.000, 4.000 e 5.000, o número ótimo de Tarefas-Escravas foi 42, 42, 35, 10 e 9, respectivamente.
- Método dos Gradientes Conjugados: para n = 1.000, 2.000, 3.000, 4.000 e 5.000, o número ótimo de Tarefas-Escravas foi 14, 42, 35, 21 e 35, respectivamente.

Considerando os métodos iterativos, não foi possível fazer testes com os algoritmos paralelos e seriais para sistemas lineares com mais de 5.000 equações usando os recursos disponíveis no CENAPAD-NE, atualmente. Essa impossibilidade se deve à limitação da partição alocada. Já para os métodos diretos, não foi possível fazer testes com os algoritmos paralelos com sistemas lineares com mais de 2.000 equações, devido à propagação de erros de arredondamento no processo de eliminação.

Para uma mesma precisão, verificamos um melhor desempenho do algoritmo dos Gradientes Conjugados, com relação ao de Gauss-Jacobi

Visando minimizar o problema de propagação de erros de arredondamento nos métodos diretos, foram usadas técnicas de Pivoteamento e Escalamento. No caso dos métodos iterativos isso não foi necessário, visto que a convergência, uma vez assegurada, independe da aproximação inicial. Desta forma, para os métodos iterativos, somente os erros de arredondamento cometidos na última iteração afetam a solução.

Podemos afirmar que os programas paralelos, implementados para os métodos diretos, se prestam aos sistemas lineares de ordem $700 \leq n \leq 2.000$, obtendo um bom desempenho usando os recursos atuais do CENAPAD-NE. Os programas paralelos para os métodos iterativos, quando há convergência garantida, são bastante vantajosos na resolução de sistemas lineares de ordem $n \geq 3.000$. O único limitante encontrado para os testes com esse método foi a falta de memória na partição alocada. Todas as operações foram feitas em precisão dupla.

6.3 - Perspectivas e trabalhos futuros

Vários grupos de pesquisa têm desenvolvido sistemas que, como o PVM, dão assistência a programadores no uso de computação distribuída. Entre os mais conhecidos está o MPI (*Message Passing Interface*).

O MPI é um software para computação paralela que usa um conjunto padronizado de rotinas para passagem de mensagens (*Message Passing*). Ele foi cuidadosamente projetado para executar eficientemente, em diferentes tipos de máquinas. A interface definida não é muito diferente dos padrões PVM, NX, Express, P4, etc. e acrescenta extensões que permitem maior flexibilidade. É compatível com sistemas de memória distribuída, memória compartilhada e qualquer combinação destes. O conceito de **comunicadores** no MPI, permite prover um elevado nível de segurança na comunicação, permitindo diferenciar mensagens de bibliotecas de mensagens de usuários.

Uma tendência para o futuro é a criação do PVMPI pelo Laboratório Nacional de Oak Ridge e a Universidade do Tennessee, tentando viabilizar uma mistura entre as melhores características do PVM e do MPI.

As funções do PVMPI seriam:

- Utilizar implementações especificas de hardware, quando disponíveis em máquinas multiprocessadoras;
- Permitir acesso à idéia de uma máquina virtual, com controle de recursos e tolerância à falhas;
- Usar a rede de comunicação PVM transparentemente, para transferir dados entre diferentes implementações de MPI, permitindo a interoperabilidade.

A característica final seria um misto entre as funções do PVM e MPI, mantendo, tanto quanto possível, as identidades desses dois sistemas, para passagem de mensagens.

Como proposta para trabalhos futuros sugerimos:

- Implementação dos algoritmos paralelos para resolução de sistemas lineares no sistema MPI.
- Implementar os algoritmos paralelos para o método iterativo dos Gradientes Bi-Conjugados, em que a matriz dos sistemas lineares é não simétrica e não singular.
- Implementação da solução exata de sistemas de equações lineares, utilizando a aritmética residual.
- Resolução de sistemas com matrizes complexas.

Referências Bibliográficas

[Akl 89] Akl, Selim G., *The design and analysis of parallel algorithms*, Prentice-Hall, Englewood Cliffs, N. J. 1989.

[Albrecht 73] Albrecht, Peter, *Análise Numérica*. Livros Técnicos e Científicos Editar S. A., Rio de Janeiro, 1973.

[Barrett 94] Barrett, R., Berry, M., Chan, Tony F., Demmel, J., Donato, J. M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and Van der Vorst, H., *Templates for the Solution of Linear Systems: Building Blocks for Iteratives Methods*. SIAM, Philadelphia, PA, 1994.

[Demidovich 73] Demidovich, B. P., Maron, I. A, *Computational Mathematics*. Mir Publishers, Moscou, 1973.

[Dongarra 79] Dongarra, J. J., Moler, C. B., Bunch, J. R. and Stewart, G. W., *Linpack Users' Guide*. SIAM, Philadelphia, 1993.

[Dongarra 84] Dongarra, J. J., Gustavson, F. G., and Karp, A., *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*. SIAM Review. Vol. 26. N° 1. January, 1984.

[Dorn 72] Dorn, W. S. and McCracken, Daniel D., *Numerical Methods with Fortran IV Case Studies*, John Wiley & Sons, Inc., New York, 1972

[Feng 72] Feng, Tse-Young, *Some characteristies of assocoative/parallel processing*. In: Sagamore Conputer Conference, 1. Proceeding, p. 5-16, Aug. 23-25, 1972.

[Fernandes 93] Fernandes, Edil S. T., e Amorim, Claudio L. de, *Arquiteturas Paralelas Avançadas*. Livro da VI Escuela Brasileño Argentina de Informatica. 1993.

[Flynn 72] Flymm, M. J., *Some computer organization and their effectiveness*. IEEE Transactions on computers, New York, v. 21, n. 9, p. 948-960, Sept. 1972.

[Forsythe 67] Forsythe, G. E. & Moler, C. B., *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Englewood Cliffs.

[Golub 89] G. H. Golub, G. H. and Van Loan, C. F., *Matrix Computations*. Second edition, The Johns Hopkins University Press, Baltimore, Maryland. 1989.

[Handler 77] Handler, W., *The Impact of classification schemes on computers architecture*. In: International Conference on Parallel Processing; Detroit, Aug, p. 23-26, 1977. Proceedings. New York, IEEE, 1977. p. 7-15.

[Hattori 96] Hattori, M. T., e Queiroz, Bruno C. N., *Métodos e Software Numéricos*. Relatório, Departamento de Sistemas e Computação, Universidade Federal da Paraíba, Campus II, Campina Grande, PB.

[Hestenes 52] Hestenes, M. R., and Stiefel, E. R., *Methods of Conjugate Gradient for Solving Linear Systems*, Nat. Bur. J. Res. 49, pp. 409-436, 1952.

[Heller 73] Heller, D., *A survey of parallel algorithms in numerical linear algebra*. SIAM Rev. 20, pp. 440-777, 1978.

[Hopkins 88] Hopkins T. and Phillips C., *Numerical Methods in Practice using a NAG Library*, Addison-Wesley, Publishing Company, Inc., New York, 1988.

[JICS 95] JICS - Joint Institute for Computational Science, *A Beginner's Guide to PVM - Parallel Virtual Machine*, University of Tennessee. Knoxville, TN 37996-1301, USA.

[Lawson 79] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T., "Basic linear algebra subprograms for FORTRAN usage", ACM Trans. Math. Softw. 5, pp. 308-323.

[Ketter 69] Ketter, Robert L., *Modern Methods of Engineering Computation*, McGraw-Hill Book Company, New York, 1969.

[Kincaid 90] Kincaid, D. R., and Cheney, E. W., *Numerical Analysis: The Mathematics of Scientific Computing*, Brooks/Cole Publ., 1990.

[Kronsjö 79] Kronsjö, Lydia., *Algorithms: Their Complexity and Efficiency*, 2ª edição, John Wiley & Sons, Inc., New York, 1979.

[Moler 78] Moler, C. B., Private Communication, 1978.

[Navaux 90] Philippe O. A. Navaux, *Processadores Pipeline e Processamento Vetorial*. VII Escola de Computação, São Paulo, IME-USP, 1990. 98p.

[Patel 94] Patel, Vidal A., *Numerical Analysis for Werth*, Sauders College, 1994.

[Pequeno 83] Pequeno, Mauro Cavalcante, *Biblioteca Educacional de Otimização Não-Linear sem Restrições*. (Dissertação de Mestrado) Universidade Federal da Paraíba, Campina Grande, Brasil, 1983.

[Ralston 65] Ralston, A., *A First Course in Numerical Analysis*, McGraw-Hill Book Company, New York, 1965.

[Sameh 78] Sameh, A. H., and Kuck, D. J., *On stable parallel linear system solvers*. Journal ACM Vol 25, N° 1, January 1978. pp. 81-91.

[Schrage 79] Schrage, L., *A More Portable Fortran Random Number Generator*, ACM Trans. Math. Software 5(1979), pp. 132-138.

[Steinberg 71] Steinberg, David I., *Computational Matrix Algebra*. MacGraw-Hill, Inc., U.S.A., 1974.

[Stewart 73] Stewart, G. W., *Introduction to Matrix Computation*, Academic Press, New York, 1973.

[Strang 79] Strang G., *Linear Algebra and its Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[Sun 87] Sun Microsystems , Inc., XDR: External Data Representation Standard. RFC 1014, Sun Microsystems, Inc., June 1987.

[Sunderam 94] Sunderam, V., Geist, A., Beguelin, A., Dongarra, J. J., Jiang, W., and Manchek, R., *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press. Cambridge, Massachusetts. 1994.

[Young 71] Young, D. M., *Iterative Solution of Large Systems*, Academic Press, New York and London, 1971.

[Young 81] Young, D. M., and Hageman Louis A., *Applied Iterative Methods*, Academic Press, New York and London, 1981.

[Varga 62] Varga, R. S., *Matrix Iterative Analysis*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1962.

[Wolfe 78] Wolfe, M. A., *Numerical Methods for Unconstrained Optimization*, Van Nostrand Reinhold Limited, London, GB, 1978.

Apêndice A

Detalhes sobre o PVM

A1 - Iniciando o PVM

O console PVM, chamado pvm, é a única Tarefa PVM que permite ao usuário, interativamente, começar, consultar e modificar a máquina virtual. O console pode ser iniciado e encerrado múltiplas vezes em qualquer uma das máquinas na máquina virtual, sem afetar o PVM ou qualquer aplicação que possa estar sendo executada.

Quando iniciado, pvm determina se PVM já está em execução; se não, pvm, automaticamente executa o *pvmd* nesta máquina. Uma vez que o PVM é iniciado, o console imprime o pronto

```
pvm>
```

Antes que examinemos os passos para compilar e carregar programas PVM, você deverá estar certo de que pode disparar o PVM e configurar uma máquina virtual. Em qualquer máquina em que o PVM foi instalado você pode digitar

```
% pvm
```

e você terá de volta o pronto do console PVM significando que o PVM está em execução nesta máquina. Você pode adicionar máquinas na máquina virtual, digitando no pronto,

```
pvm> add nome_da_máquina
```

E você pode remover máquinas (exceto aquela onde você está) da sua máquina virtual, digitando

```
pvm> delete nome_da_máquina
```

Se você receber a mensagem "não posso iniciar o *pvmd*", então verifique a seção dos problemas mais comuns de instalação e tente novamente.

Para ver qual é a configuração atual da máquina virtual, digite

```
pvm> conf
```

Para ver qual é a Tarefa PVM que está sendo executada na máquina virtual, digite

```
pvm> ps -a
```

Em qualquer máquina da máquina virtual, você pode digitar

```
% pvm
```

e você receberá a mensagem "pvm já está em execução" e o pronto do console. Quando você tiver terminado os trabalhos na máquina virtual, deverá digitar

```
pvm> halt
```

Este comando mata qualquer Tarefa PVM, encerra as atividades na máquina virtual e sai do console PVM. Este é o método recomendado para terminar o PVM, pois ele assegura que a máquina virtual realmente encerre suas atividades.

Você deverá praticar iniciar, encerrar e adicionar máquinas ao PVM, até que você esteja confortável com o console PVM.

```
pvm> kill  < id_processo >
```

Este comando pode ser usado para encerrar qualquer processo PVM.

```
pvm> quit
```

Este comando sai do console, deixando os *daemons* e as Tarefas PVM em execução.

Se você não quiser digitar uma certa quantidade de nomes de máquinas toda vez que iniciar o PVM, há uma opção para criar um arquivo com os nomes das máquinas que se chama *hostfile*. Você pode listar os nomes de máquinas no arquivo, um por linha e então digite

```
% pvm hostfile
```

O PVM, então, adicionará as máquinas listadas, simultaneamente, antes do pronto do console PVM aparecer. Várias opções podem ser especificadas a cada máquina no *hostfile*.

A.2 - Executando Programas PVM

Nesta seção, aprenderemos como compilar e executar programas PVM.

Para compilar:

- Edita-se os fontes em um diretório apropriado.
- Na compilação deve-se sempre incluir as bibliotecas PVM necessárias (C, FORTRAN).
- Usar o compilador com sintaxe adequada:
  - C:

cc -o <programa> <programa.c> -I/usr/local/pvm3/include -L/usr/local/pvm3/RS6K/lib -lpvm3

  - FORTRAN:

f77 -o <programa> <programa.f> -I/usr/local/pvm3/include -L/usr/local/pvm3/RS6K/lib -lfpvm3 -lpvm3

Para executar programas PVM:

- Antes de executar o programa propriamente dito, deve-se invocar o PVM
- Pode-se fazer isso basicamente de duas formas:
  - Chamando diretamente o *daemon* (em *background*):

    %pvmd3 < arquivo_de_máquinas > &

- Pelo **console pvm**, que permite interação com o usuário:

  % pvm

- Para executar o programa, deve-se sair do console PVM pela opção **quit.**

A.3 - Escrevendo Aplicações PVM

Durante a execução, antes que qualquer outra rotina PVM seja chamada, uma Tarefa precisa primeiro registrar-se no PVM. Isto atribui um número inteiro à Tarefa. Uma vez que a Tarefa tenha completado seu trabalho no PVM, ela precisa informar ao *daemon* PVM que está deixando a máquina virtual. Isto não mata o processo, que pode continuar a executar Tarefas. Entretanto, a Tarefa não poderá interagir com qualquer outra Tarefa.

As seções seguintes mostram um programa exemplo em C e FORTRAN.

Os exemplos dados aqui, ilustram um código, no modelo Mestre-Escravo, que soma as componentes de um vetor de inteiros. A Tarefa-Mestra distribui cinco (5) Tarefas-Escravas, envia a cada uma delas uma partição do vetor, recebe as somas parciais de cada uma das Tarefas-Escravas e adiciona-as para a soma total. As Tarefas-Escravas recebem um vetor de inteiros, somam todas as componentes do vetor e enviam a soma total de volta à Tarefa-Mestra. A Figura A.3.1 mostra o cálculo da norma 1, para um vetor de tamanho 8, usando duas (2) Tarefas-Escravas

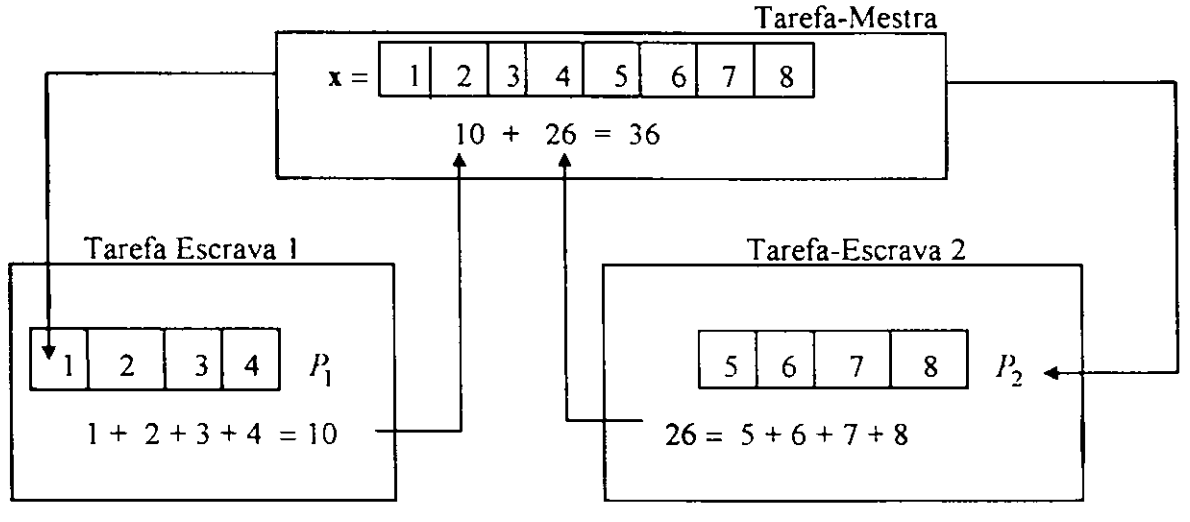

Figura A.3.1 - Cálculo da norma 1

Considerando a Figura A.3.1, a Tarefa-Mestra divide o vetor **x** em duas partições (P_1 e P_2). Cada partição é enviada a cada Tarefa-Escrava. Cada Tarefa-Escrava faz o cálculo da norma 1 da partição recebida e envia o resultado para a Tarefa-Mestra. Finalmente, a Tarefa-Mestra recolhe os resultados parciais das Tarefas-Escravas e obtém a norma 1 do vetor **x**.

A.3.1 - Exemplo em C

```
#include "pvm3.h"
#include <stdio.h>
#define size 1000
#define nprocs 5
main()
{       int mytid, task_ids[ nprocs ],codret, bufid,numt;
        int x[ size ], results[ nprocs ], soma = 0;
        int i, msgmt,msgwk, num_data = size / nprocs
        mytid = pvm_mytid();                                    /*      registro no pvm      */
        for ( i = 0 ; i < size ; i ++ )
                a[ i ] = i % 25;
        /* distribuição das Tarefas-Escravas */
        numt = pvm_spawn( "worker", null, pvmtaskdefault, "", nprocs, task_ids);
        msgmt = 4;
        for ( i = 0 ; i < nprocs ; i ++ )                       /* envia dados para as Tarefas-Escravas */
        {       bufid = pvm_initsend( pvmdatadefault);
                codret = pvm_pkint( &num_data, 1, 1 );
                codret = pvm_pkint( &x[ num_data*i ], num_data, 1 );
                codret = pvm_send( task_ids[ i ], msgmt );
        }
        msgwk = 2;
        soma = 0;
        for ( i = 0 ; i < nprocs ; i ++ )                       /* espera e recebe os resultados */
        {       bufid = pvm_recv( task_ids[ i ], msgwk );
                codret = pvm_upkint( &results[ i ], 1, 1 );
                soma = soma + results[ i ]
        }
        printf("a soma é %d \n", soma);
        codret = pvm_exit();
}
```

Figura A.3.2 - Exemplo de Tarefa-Mestra em C.

A Figura A.3.2 mostra o código em C para a Tarefa-Mestra que está contido no arquivo `master.c`.

```
#include "pvm3.h"
#include <stdio.h>
main()
{       int mytid;
        int  i, soma, *a, msgmt, msgwk, bufid;
        int num_data, master,codret,num_data;
        msgmt = 4;
        msgwk = 2;
        mytid = pvm_mytid();                                    /* Registro no PVM   */
        master = pvm_parent();
        bufid = pvm_recv( master , msgmt );                     /* Recebe uma parte do vetor a ser somado  */
        codret = pvm_upkint( &num_data, 1, 1 );
        a = ( int *)malloc(num_data*sizeof(int));
        codret = pvm_upkint( a, num_data, 1);
        soma = 0;
        for ( i = 0 ; i < num_data ; i ++ )
                soma = soma + a[ i ];
        bufid = pvm_initsend( PvmDataDefault);          /* Envia a soma computada de volta para a Tarefa-Mestra */
        codret = pvm_pkint( &soma, 1, 1 );
        codret = pvm_send( master, msgwk );
        codret = pvm_exit();
}
```

Figura A.3.3 - Exemplo de Tarefa-Escrava em C.

A Figura A.3.3 mostra o código C para a Tarefa-Escrava da aplicação PVM armazenada no arquivo `worker.c`. Vamos examinar cada Tarefa, para ver como as chamadas PVM são usadas.

A primeira linha de ambas as Tarefas inclui o arquivo de cabeçalho PVM. Este arquivo dá as definições para nomes simbólicos PVM e funções.

A primeira chamada PVM na Tarefa-Mestra informa ao *daemon* PVM de sua existência através do registro da Tarefa na máquina virtual. A função `pvm_mytid()` é usada para este propósito e atribui um identificador de Tarefa (TID) para a Tarefa-Mestra.

```
mytid = pvm_mytid()
```

O resultado retornado é o identificador da Tarefa-Mestra. Não há parâmetros para esta função.

Depois que a Tarefa é registrada na máquina virtual, a Tarefa-Mestra inicializa o vetor cujas componentes serão somadas. Depois, as Tarefas-Escravas são distribuídas através da chamada à rotina `pvm_spawn()`.

```
numt=pvm_spawn("worker",null,pvmtaskdefault,"",nprocs,task_ids)
```

O primeiro parâmetro é o nome do arquivo executável, que será usado como Tarefa-Escrava. O executável precisa residir na máquina em que ele será iniciado. A localização padrão é `$HOME/pvm3/bin/$PVM_ARCH/filename`. O terceiro parâmetro é usado para determinar uma máquina especifica, ou o tipo de arquitetura onde a Tarefa distribuída será executada.

O quinto parâmetro, NPROCS, especifica o número de cópias da Tarefa-Escrava a ser distribuída e `task_ids` é um apontador para um vetor de inteiros que retorna os TID de todas as Tarefas-Escravas distribuídas. A função retorna o número de Tarefas-Escravas que foram distribuídas com sucesso. Se algumas Tarefas não foram lançadas, as (`nprocs-numt`) últimas posições de `task_ids` conterão os códigos de erro das Tarefas-Escravas que não foram lançadas.

Para enviar uma mensagem de uma Tarefa para outra, um *buffer* de emissão é criado para manter os dados. A função `pvm_initsend()` cria e limpa um *buffer*, retornando o identificador do *buffer* em `bufid`.

```
bufid = pvm_initsend(PvmDataDefault)
```

Se um único *buffer* é usado, `pvm_initsend()` precisa ser chamada cada vez que uma nova mensagem for enviada. De outra forma, a nova mensagem será acrescentada à mensagem que já está no *buffer*. O parâmetro `PvmDataDefault` especifica como a codificação será feita. Esta opção usará a codificação de mensagens XDR (*eXtended Data Representation standard*), se a máquina virtual for heterogênea, caso contrário nenhuma codificação é feita.

Antes de emitir um comando de emissão, um *buffer* precisa ser empacotado com os dados a serem enviados. As funções que empacotam dados no *buffer* de emissão ativo são `pvm_pkXXXX()` onde `XXXX` indica o tipo de dado que será empacotado. Os tipos de dados suportados pelo PVM são byte (`byte`), complexo (`cplx`), complexo duplo (`dcplx`), duplo (`double`), real (`float`), inteiro (`int`), longo (`long`) e curto (`short`). O nosso exemplo empacota inteiros e usa a função `pvm_pkint()`.

```
codret=pvm_pkint( &num_data, 1, 1 );
codret=pvm_pkint( &x[ num_data*i ], num_data, 1 );
```

Cada uma das funções citadas acima tem 3 parâmetros. O primeiro, é um apontador para o primeiro item a ser empacotado na mensagem, o segundo parâmetro, é o número total de itens a serem empacotados (não é o número de bytes). E o terceiro parâmetro, é o passo a ser usado para ler os itens da memória.

Uma única mensagem pode conter qualquer número de diferentes tipos de dados e não há limite na complexidade da mensagem. Contudo, você deverá assegurar que a mensagem recebida seja desempacotada, da mesma maneira que foi originalmente empacotada. A função para enviar uma mensagem é `pvm_send()`.

```
codret = pvm_send( task_ids[i], msgmt)
```

Esta função liga um rótulo inteiro (`msgmt=4`) à mensagem e imediatamente envia o conteúdo do *buffer* de emissão para a Tarefa com identificador `task_ids[i]`. O parâmetro `msgmt` pode ser usado para distinguir diferentes tipos de mensagens que a Tarefa pode enviar.

No 2° *loop* da Figura A.3.2, a Tarefa-Mestra limpa o *buffer* de emissão para cada nova mensagem e empacota este *buffer* com duas *strings*:

1) o número de elementos do vetor que seguirá na mensagem e

2) a partição do vetor a ser somada.

Uma vez que cada partição do vetor **x** a ser enviada, começa na posição (`num_dataxi`), o passo para a função de empacotamento é 1. O vetor `task_ids`, que foi retornado da chamada à `pvm_spawn()`, é usado para endereçar cada uma das Tarefas-Escravas que receberá a partição do vetor. O valor `msgmt = 4`, arbitrariamente escolhido, é usado para rotular a mensagem.

Depois que as partições do vetor forem distribuídas, a Tarefa-Mestra precisa receber a soma parcial de cada uma das Tarefas-Escravas. Para receber a mensagem, a Tarefa-Mestra chama a função `pvm_recev()`.

```
bufid = pvm_recv(task_ids[i], msgwk ).
```

Ela receberá uma mensagem da Tarefa identificada por `task_ids[i]`, com rótulo `msgwk`, e a colocará no *buffer* de recepção com identificador `bufid`.

Uma vez que a mensagem tenha chegado, os dados precisam ser desempacotados. As funções para desempacotamento são `pvm_upkXXXX()`, onde XXXX corresponde ao tipo de dado que será desempacotado. As mesmas extensões usadas nas funções `pvm_pkXXXX()` são válidas para `pvm_upkXXXX()`. Por exemplo, como a Tarefa-Mestra está recebendo um inteiro de suas Tarefas-Escravas, ela chama a função para desempacotamento de inteiro.

```
codret=pvm_upkint( &results[ i ], 1, 1 )
```

O primeiro parâmetro é um apontador, para onde o primeiro item desempacotado será armazenado. O segundo e o terceiro são o número de itens a serem desempacotados e o passo a ser usado, para desempacotar os itens para a memória. Nosso exemplo desempacota cada resultado parcial recebido para uma posição diferente do vetor `results` e adiciona-o à variável `soma`.

Depois que a soma é computada e exibida, a Tarefa-Mestra informa ao *daemon* PVM local que está deixando a máquina virtual. Isto é feito através da chamada à rotina `pvm_exit()`.

```
codret = pvm_exit()
```

Quanto à Tarefa-Escrava, depois de registrar-se na máquina virtual, ela espera para receber a partição do vetor. Depois de desempacotar o número de itens de dados da mensagem, a Tarefa aloca bastante espaço para manter o resto dos dados contidos na mensagem. As componentes da partição do vetor são somadas e o total é enviado de volta à Tarefa-Mestra. O identificador da Tarefa-Mestra é retornado pela função `pvm_parent()`.

```
master = pvm_parent()
```

Como a Tarefa-Mestra está esperando uma mensagem com rótulo igual a 2, este valor precisa ser usado na chamada `pvm_send()`.

A.3.2 - Exemplo em FORTRAN

```
      PROGRAM MASTER
      INCLUDE 'fpvm3.h'
      INTEGER SIZE, NPROCS
      PARAMETER (SIZE = 1000, NPROCS = 5)
      INTEGER MYTID, TASKID(NPROCS),CODRET, BUFID
      INTEGER X(SIZE), RESULT(NPROCS), SOMA
      INTEGER I, NDATA,MSGMT,MSGWK
C     REGISTRO NO PVM
      CALL PVMFMYTID(MYTID)
      DO 10 I = 1 , SIZE
            X( I ) = MOD( I , 25 )
10    CONTINUE
C     DISTRIBUÍÇÃO DAS TAREFAS-ESCRAVAS
      CALL PVMFSPAWN('worker', PVMDEFAULT , '*' , NPROCS , TASKID , CODRET )
      NDATA = SIZE / NPROCS
C     ENVIA DADOS PARA AS TAREFAS-ESCRAVAS
      MSGMT=4
      DO 20 I = 0 , NPROCS
         CALL PVMFINITSEND( PVMDEFAULT, BUFID)
         CALL PVMFPACK( INTEGER4, NDATA, 1, 1, CODRET )
         CALL PVMFPACK( INTEGER4, X(NDATA*(I-1)+1), NDATA, 1, CODRET )
         CALL PVMFSEND( TASKID( I ), MSGMT , CODRET )
20    CONTINUE
C     ESPERA E RECEBE OS RESULTADOS
      MSGWK = 2
      SOMA = 0
      DO 30 I = 0, NPROCS
            CALL PVMFRECV( TASKID( I ), MSGWK, BUFID )
            CALL PVMFUNPACK( INTEGER4, RESULT( I ), 1, 1,CODRET )
            SOMA = SOMA + RESULT( I )
30    CONTINUE
      PRINT *,' A SOMA É ', SOMA
      CALL PVMFEXIT(CODRET)
      STOP
      END
```

Figura A.3.4 - Exemplo de Tarefa-Mestra em FORTRAN.

A Figura A.3.4 mostra o código em FORTRAN para a Tarefa-Mestra que está contido no arquivo `master.f`.

```
      PROGRAM WORKER
      INCLUDE 'fpvm3.h'
      INTEGER MYTID
      INTEGER I, SOMA, A( 1000), BUFID,CODRET
      INTEGER NDATA, MASTER,MSGMT, MSGWK
C     REGISTRO NO PVM
      CALL PVMFMYTID( MYTID )
      CALL PVMFPARENT(MASTER)
      MSGMT = 4
      MSGWK = 2
C     RECEBE UM PARTE DO VETOR A SER SOMANDO
      CALL PVMFRECV( MASTER, MSGMT , BUFID)
      CALL PVMFUNPACK( INTEGER4, NDATA, 1, 1, CODRET)
      CALL PVMFUNPACK( INTEGER4, A, NDATA, 1,CODRET)
      SOMA = 0
      DO 10 I = 1 , NDATA
            SOMA = SOMA + A( I )
10    CONTINUE
C
C     ENVIA A SOMA COMPUTADA DE VOLTA AO PROGRAMA MESTRE
      CALL PVMFINITSEND( PVMDEFAULT, BUFID)
      CALL PVMFPACK( INTEGER4, SOMA, 1, 1, CODRET )
      CALL PVMFSEND( MASTER, MSGWK , CODRET );
C
      CALL PVMFEXIT(CODRET)
      STOP
      END
```

Figura A.3.5 - Exemplo de Tarefa-Escrava em FORTRAN.

A Figura A.3.5 mostra o código, em FORTRAN, para a Tarefa-Escrava da aplicação PVM, armazenado no arquivo `worker.f`. A seguir, nós examinaremos cada programa para ver como as chamadas PVM são usadas.

As rotinas FORTRAN PVM são rotinas de interface para as correspondentes rotinas em C. O último parâmetro de todas as rotinas em FORTRAN foi projetado para retornar o valor retornado pela função equivalente em C. Para a maioria das rotinas isto não será nada mais que um código de informação para relatar o sucesso da chamada à rotina.

A primeira linha em ambas as Tarefas incluem um arquivo de cabeçalho PVM. Este arquivo dá as definições para nomes simbólicos e rotinas do PVM.

A primeira chamada PVM na Tarefa-Mestra informa ao *daemon* PVM da sua existência, através do registro da Tarefa na máquina virtual. A função `pvmfmytid()` é usada para este propósito e atribui um TID para a Tarefa-Mestra.

```
call pvmfmytid(mytid)
```

O identificador de Tarefa é retornado através do parâmetro inteiro `mytid`.

Depois que a Tarefa é registrada na máquina virtual, a Tarefa-Mestra inicializa o vetor **x**. Depois, as Tarefas-Escravas são distribuídas através da chamada à rotina `pvmfspawn()`.

```
call pvmfspawn('worker',pvmdefault,'*',nprocs,taskid,
codret)
```

O primeiro parâmetro é uma *string* contendo o nome do arquivo executável que será usado como Tarefa-Escrava. O segundo parâmetro é usado para determinar a máquina específica ou o tipo da arquitetura onde a Tarefa distribuída será executada. O parâmetro `nprocs` especifica o número de cópias da Tarefa-Escrava a serem distribuídas e `taskid` um vetor de inteiros que contém os TID de todas as Tarefas-Escravas distribuídas com sucesso. A função retorna o número de Tarefas que foram distribuídas com sucesso, através do parâmetro `codret`. Se algumas Tarefas não puderam ser lançadas, as (`nprocs-codret`) últimas posições de `taskid` conterão os códigos de erro das Tarefas-Escravas que não foram lançadas.

A função `pvmfinitsend()` cria e limpa um *buffer* para emissão e retorna o identificador do *buffer* em `bufid`.

```
call pvmfinitsend(PvmDefault, bufid)
```

Se um único *buffer* é usado, `pvmfinitsend()` precisa ser chamada, toda vez que uma nova mensagem for enviada. De outra forma a nova mensagem será acrescentada à mensagem que já está no *buffer* ativo.

Antes de emitir um comando de emissão, um *buffer* precisa ser empacotado com os dados a serem enviados. A função que empacota dados no *buffer* de emissão ativo é `pvmfpack()`.

```
call pvmfpack(integer4, ndata, 1, 1, codret )
call pvmfpack(integer4,x(ndata*i-1)+1),ndata,1,codret)
```

O primeiro parâmetro inteiro especifica o tipo de dado a ser empacotado. Os valores possíveis são:

```
string bytel integer2 integer4 real4 complex8 real8 complex16
```

O segundo argumento é um apontador para o primeiro item a ser empacotado, o terceiro argumento é o número total de itens a serem empacotados (não o número de bytes) e o quarto é o passo a ser usado para ler os itens da memória. Você deverá assegurar que a mensagem recebida seja desempacotada, da mesma maneira que foi originalmente empacotada.

A função para enviar uma mensagem é `pvmfsend()`.

```
call   pvmfsend(taskid(i), msgmt, codret)
```

Esta função liga um rótulo inteiro (`msgmt=4`) à mensagem e imediatamente envia o conteúdo do *buffer* de emissão para a Tarefa com identificador `taskid[i]`.

No 2° *loop* da Figura A.3.4, a Tarefa-Mestra limpa o *buffer* de emissão, para cada nova mensagem, e empacota este *buffer* com o número de elementos do vetor que seguirá na mensagem e a partição do vetor a ser enviada para a Tarefa-Escrava. Uma vez que cada partição do vetor **x** a ser enviado, começa na posição (`num_datax(i-1)+1`), o passo para a função de empacotamento é 1. O vetor `taskid` que foi retornado da chamada à `pvmfspawn()` é usado para endereçar cada uma das Tarefas-Escravas que receberá a partição do vetor. O valor `msgmt` = 4, arbitrariamente escolhido, é usado para rotular a mensagem.

Depois que as partições do vetor tiverem sido distribuídas, a Tarefa-Mestra precisa receber a soma parcial de cada uma das Tarefas-Escravas. Para receber a mensagem, a Tarefa-Mestra chama a função `pvmfrecev()`.

```
call pvmfrecv(taskid(i), msgwk, bufid).
```

Ela receberá uma mensagem de uma Tarefa com identificador `taskid[i]`, com rótulo `msgwk`, e a colocará no *buffer* de recepção ativo com identificador `bufid`.

Uma vez que a mensagem tenha chegado, os dados precisam ser desempacotados. A função para desempacotamento é `pvmfunpack()`.

```
call pvmfunpack( integer4, result(i), 1, 1, codret )
```

O primeiro parâmetro especifica o tipo de dado a ser desempacotado. As opções usadas para `pvmfpack()` são válidas para esta rotina. O segundo parâmetro indica onde o 1° item desempacotado será armazenado. O terceiro e o quarto argumento dão o número de itens a serem desempacotados e o passo a ser usado quando desempacotando os itens para a memória do

usuário. Nosso exemplo desempacota cada resultado parcial recebido das Tarefas-Escravas e o coloca em cada posição diferente do vetor `results` e, finalmente, adiciona-o à variável soma.

Depois que a soma é computada e exibida, a Tarefa-Mestra informa ao *daemon* PVM local que está deixando a máquina virtual. Isto é feito através da chamada à rotina `pvmfexit()`.

```
call  pvmfexit(codret)
```

Quanto à Tarefa-Escrava, depois de registrar-se na máquina virtual, ela espera para receber a partição do vetor a ser somado. Depois de desempacotar o número de itens de dados da mensagem, a Tarefa aloca bastante espaço para manter o resto dos dados contidos na mensagem. Os elementos da partição do vetor são somados e o total é enviado de volta à Tarefa-Mestra. O identificador de Tarefa-Mestra é retornado pela função `pvm_parent()`.

```
master = pvmfparent()
```

Como a Tarefa-Mestra está esperando uma mensagem com rótulo igual a 2, este valor precisa ser usado na chamada `pvm_send()` [JICS 95].

A.4 - Paralelização do produto interno

A computação de um produto interno de dois vetores pode ser facilmente paralelizada; a Tarefa-Mestra divide o vetor **x** em n partições e envia, para cada uma das n Tarefas-Escravas, uma partição do vetor. Cada Tarefa-Escrava computa o produto interno da partição correspondente do vetor. Os resultados das operações efetuadas nas partições devem ser enviados para a Tarefa-Mestra para serem combinados no produto interno global.

A.5 - Paralelização do produto matriz-vetor

O produto matriz-vetor é facilmente paralelizável em máquinas com memória distribuída, através da divisão da matriz em blocos e o envio de cada bloco juntamente com o vetor, às Tarefas-Escravas. Cada Tarefa-Escrava, então, computa apenas uma partição do vetor solução. Vamos ilustrar, com um exemplo, com uma matriz 4×4 e um vetor de tamanho 4, como mostrado na Figura A.3.6.

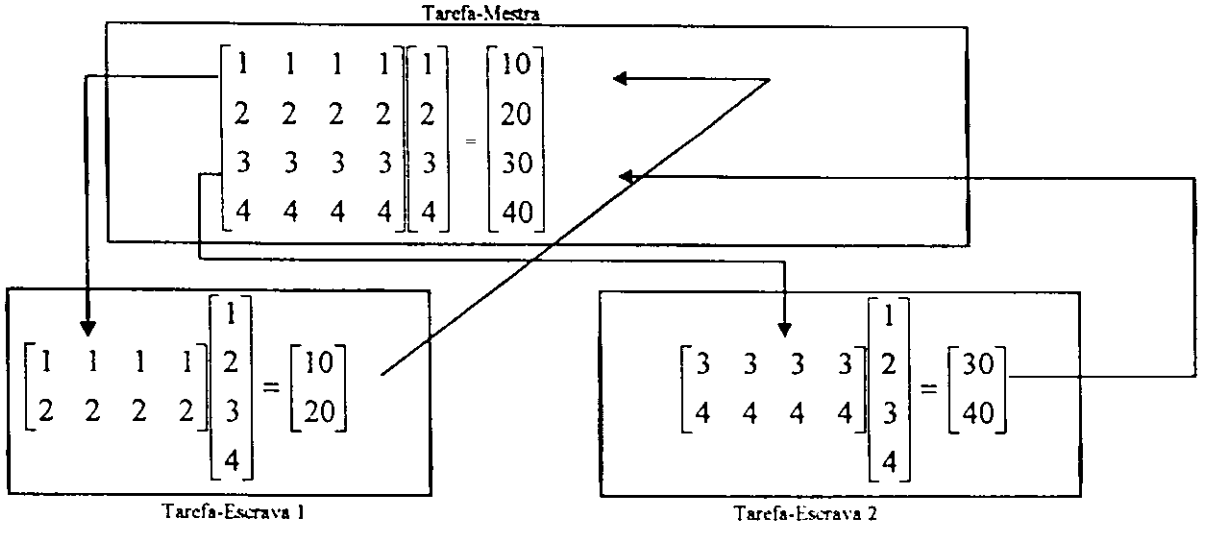

Figura A.3.6 - Multiplicação matriz-vetor

A.6 - Algumas rotinas do PVM

A.6.1 - Controle de Processos

pvmfmytid( ) pvm_mytid( )

```
int tid = pvm_mytid(void)
call pvmfmytid(tid)
```

A rotina `pvm_mytid()` retorna o TID deste processo e pode ser chamada múltiplas vezes. Ela registra este processo no PVM na sua primeira chamada. Qualquer chamada ao sistema PVM (não apenas `pvm_mytid()`) registrará a Tarefa no PVM, se a Tarefa não foi registrada antes desta chamada. É uma prática comum chamar `pvm_mytid()`, primeiro para o registro.

Se o PVM não foi iniciado antes que uma aplicação chame `pvm_mytid()`, o `tid` retornado será < 0.

Parâmetros:

`tid` identificador inteiro do processo PVM. Valores menores que zero indicam erro.

pvmfexit( ) pvm_exit( )

```
int info = pvm_exit(void)
call pvmfexit(info)
```

A rotina `pvm_exit()` diz ao *pvmd* local que este processo está deixando o PVM. Esta rotina não mata o processo, que pode continuar a executar Tarefas exatamente como qualquer outro processo serial UNIX. Os usuários comumente chamam `pvm_exit()` antes de saírem de seus programas C e antes do comando STOP em programas FORTRAN.

Parâmetros:

`info` código inteiro de *status* retornado pela rotina. Valores menores que zero indicam erro.

pvmfspawn( ) pvm_spawn( )

```
int numt = pvm_spawn(char *task,char **argv,int flag,
char *where, int ntask, int *tids)
call pvmfspawn (task,flag,where, ntask, tids, numt)
```

A rotina `pvm_spawn()` dispara `ntask` cópias do arquivo executável chamado `task` na máquina virtual. `argv` é um apontador para um vetor de argumentos para `task` com o final do vetor especificado por NULL. Se a Tarefa não tem argumentos, então `argv` é NULL.

As máquinas em que os processos PVM serão executados, são estabelecidas pelos argumentos `flag` e `where`. Parâmetros:

`task` *string* de caracteres contendo o nome do arquivo executável a ser iniciado. O executável precisa está residente na máquina em que ele será executado. A localização *default* é `$HOME/pvm3/bin/$PVM_ARCH/filename`.

`argv` ponteiro para um vetor de argumentos para o executável `task`.

`flag` inteiro especificando opções de distribuição.
Em C, `flag` deverá ser a soma de:

Valor	Opção	Significado
0	PvmTaskDefault	PVM escolhe onde distribuir cada processos
1	PvmTaskHost	`where` indica uma máquina particular
2	PvmTaskArch	`where` indica um tipo de arquitetura
4	PvmTaskDebug	inicia processos sob o *debugger*

`where` *string* de caracteres indicando onde executar os processos PVM. Dependendo do valor de `flag`, `where` pode ser um nome de máquina como "ibm1.epm.ornl.gov" ou uma classe de arquitetura PVM tal como "SUN4". Se `flag` é zero, `where` é ignorado e o PVM selecionará a máquina mais apropriada.

`ntask` inteiro especificando o número de cópias do executável a serem iniciadas.

`tids` vetor de inteiros com tamanho `ntask`. No retorno o vetor contém os TID dos processos PVM iniciados através da chamada à `pvm_spawn()`. Se há um erro de inicialização em uma dada Tarefa, então a localização correspondente no vetor conterá o código de erro correspondente.

`numt` inteiro retornado indicando o número de Tarefas distribuídas. Valores menores que zero indicam erro no sistema. Um valor positivo menor que `ntask` indica uma falha parcial. Neste caso o usuário deverá verificar o vetor `tids`.

pvmfkill( ) pvm_kill( )

```
int info = pvm_kill(int tid)
call pvmfkill(tid, info)
```

A rotina `pvm_kill()` mata qualquer outra Tarefa PVM identificada por `tid`. Esta rotina não foi projetada para matar a Tarefa que a chamou, que deverá ser realizada pela chamada à rotina `pvm_exit()` seguido por `exit()`.

A rotina `pvm_kill()` envia um sinal terminador (`SIGTERM`) para o processo PVM identificado por `tid`. Se a chamada à rotina `pvm_kill()` for bem sucedida, `info` será zero.

Parâmetros:

`tid` identificador inteiro do processo PVM a ser encerrado.
`info` código inteiro de *status* retornado pela rotina. Valores menores que zero indicam erro.

pvmfparent( ) **pvm_parent( )**

```
int tid = pvm_parent(void)
call pvmfparent(tid)
```

A rotina `pvm_parent()` retorna o TID do processo que gerou a Tarefa que fez a chamada à rotina. Se o processo não foi criado através da chamada à `pvm_spawn()`, então `tid` é igual a PvmNoParent.

Parâmetros:

`tid` inteiro retornando o identificador da Tarefa pai. Se o processo não foi criado através da chamada à rotina `pvm_spawn()`, então `tid` = PvmNoParent.

A.6.2 - Passagem de Mensagens

Enviar uma mensagem compreende três passos no PVM. Primeiro, um *buffer* de emissão precisa ser inicializado através de uma chamada à rotina `pvm_initsend()` ou `pvm_mkbuf()`. Segundo, a mensagem precisa ser "empacotada" para este *buffer* usando qualquer número e combinação das rotinas `pvm_pk*()`. (em FORTRAN todo empacotamento de mensagens é feito pela rotina `pvmfpack()`). Terceiro, a mensagem é enviada para outro processo através da chamada à rotina `pvm_send()` ou espalhada através da chamada à rotina `pvm_mcast()`.

Uma mensagem é recebida através da chamada a uma rotina receptora bloqueada ou a uma não bloqueada, em seguida "desempacotando" cada um dos itens do pacote para o *buffer* de recepção. As rotinas de recepção podem estabelecer aceitar qualquer mensagem, qualquer mensagem de uma fonte especifica, qualquer mensagem com um rótulo especifico ou somente mensagens com um rótulo de uma dada fonte.

A.6.2.1 - Buffers de Mensagens

pvmfinitsend( ) **pvm_initsend( )**

```
int bufid = pvm_initsend(int encoding)
call pvmfinitsend(encoding, bufid)
```

Se o usuário está usando um único *buffer* de emissão (e este é o caso geral) então `pvm_initsend()` é a única rotina para *buffer* solicitada. Ela é chamada antes do empacotamento de uma nova mensagem em um *buffer*. A rotina `pvm_initsend()` limpa o *buffer* de emissão e cria um novo para empacotar uma nova mensagem. O esquema de codificação usado para este empacotamento é estabelecido por `encoding`. O novo identificador do *buffer* é retornado em `bufid`.

Parâmetros:

`encoding` inteiro indicando o próximo esquema de codificação das mensagens.
As opções em C para `encoding` são:
PvmDataDefault - a codificação XDR é usada como *default* pois o PVM não sabe se o usuário está adicionando uma máquina heterogênea antes desta mensagem ser enviada. Se o usuário sabe que a próxima mensagem será enviada somente para uma máquina que compreende o formato nativo, então podemos usar **PvmDataRaw**.
PvmDataRaw - nenhuma codificação é feita. Mensagens são enviadas no seu formato original. Se o processo de recepção não puder ler neste formato, ele irá retornar uma mensagem de erro durante o desempacotamento.

`bufid` identificador inteiro do *buffer* de mensagens. Valores menores que zero indicam um erro.

No PVM 3 há um *buffer* de emissão ativo e um *buffer* de recepção ativo por processo em qualquer momento. O programador pode criar qualquer número de *buffer* de mensagens e permutar entre eles para o empacotamento e emissão de dados. As rotinas de empacotamento, emissão, recepção e desempacotamento afetam somente o *buffer* ativo.

A.6.2.2 - Empacotando Dados

Cada uma das rotinas seguintes em C empacota um vetor de um determinado tipo de dados para o *buffer* de emissão ativo. Elas podem ser chamadas múltiplas vezes para empacotar dados em uma única mensagem. Assim, uma mensagem pode conter vários vetores, cada um com um tipo diferente de dado. Não há limite para a complexidade das mensagens, mas uma aplicação deverá desempacotar a mensagem exatamente como ela foi empacotada.

Os argumentos para cada uma das rotinas são um apontador para o 1º item a ser empacotado, `nitem` é o número total de itens do vetor a serem empacotados, e `stride` é o

passo a ser usado quando empacotando os itens. Um *stride* igual a 1 significa que os itens estão ocupando posições contíguas de memória serão empacotados.

```
int info=pvm_packf( const char *fmt,...)
int info=pvm_pkbyte(  char   *xp, int nitem, int stride)
int info=pvm_pkcplx(  float    *cp, int nitem, int stride)
int info=pvm_pkdcplx( double *zp, int nitem, int stride)
int info=pvm_pkdouble(double *dp, int nitem, int stride)
int info=pvm_pkfloat( float  *fp, int nitem, int stride)
int info=pvm_pkint(   int    *ip, int nitem, int stride)
int info=pvm_pklong(  long   *ip, int nitem, int stride)
```

Cada uma das rotinas `pvm_pk*()` empacota um vetor de um determinado tipo de dado no *buffer* de emissão ativo. A rotina em FORTRAN `pvmfpack(STRING,...)` espera que `nitem` seja o número de caracteres na *string* e `stride` seja 1.

Se o empacotamento for bem sucedido, `info` será zero. Se algum erro ocorrer, `info` será < 0. Uma variável simples (não um vetor) pode se empacotada estabelecendo `nitem=1` e `stride=1`.

Uma única subrotina FORTRAN substitui todas as funções C acima.

```
call pvmfpack(what, xp, nitem, stride, info)
```

O argumento `xp` é o primeiro item do vetor a ser empacotado. Em FORTRAN o número de caracteres em uma *string* a ser empacotada precisa ser indicada em `nitem`. O inteiro `what` indica o tipo de dado a ser empacotado.

A rotina `pvm_packf()` usa uma expressão no formato printf para indicar o tipo de dados a ser empacotado no *buffer* de emissão.

Parâmetros:

`fmt`	expressão do formato dos dados a ser empacotado.
`nitem`	número total de itens a serem empacotados (não o número de bytes).
`stride`	passo a ser usado quando empacotando os itens.
`xp`	ponteiro para o começo de um bloco de bytes, ou seja, a primeira posição do vetor de dados (itens) a ser empacotado. Pode ser qualquer tipo de dados, mas precisa concordar com o correspondente tipo de dado a ser desempacotado.
`cp`	vetor de número complexos de pelo menos `nitem*stride` itens de tamanho.
`zp`	vetor de número complexos em precisão dupla de pelo menos `nitem×stride` itens de tamanho.
`dp`	vetor de número reais em precisão dupla de pelo menos `nitem×stride` itens de tamanho.
`fp`	vetor de número reais de pelo menos `nitem×stride` itens de tamanho.
`ip`	vetor de número inteiros em precisão dupla de pelo menos `nitem×stride` itens de tamanho.
`jp`	vetor de número inteiros de pelo menos `nitem×stride` itens de tamanho.
`sp`	apontador para uma *string* de caracteres terminada por NULL.
`what`	inteiro indicando o tipo de dados sendo empacotado.

```
STRING     0  REAL4      4  BYTE1      1     COMPLEX8  5      INTEGER2     2
REAL8      6  INTEGER4   3  COMPLEX16  7
```

`info` código inteiro de *status* retornado pela rotina. Valores menores que zero indicam um erro.

A.6.2.3 - Enviando e Recebendo Dados

pvmfsend( ) pvm_send( )

```
int info = pvm_send(int tid,  int msgtag)
call pvmfsend(tid, msgtag, info)
```

A rotina `pvm_send()` rotula uma mensagem armazenada no *buffer* de emissão ativo com um identificador inteiro `msgtag` e a envia imediatamente para o processo identificado por `tid`. `msgtag` é usado para rotular o conteúdo da mensagem. A rotina `pvm_send()` é assíncrona. A computação no processo emissor recomeça assim que a mensagem esteja segura em seu caminho para o processo de recepção. Parâmetros:

`tid` identificador inteiro do processo destino.

`msgtag` inteiro fornecido pelo usuário usado para rotular a mensagem. `msgtag` deverá ser ≥ 0.

`info` código inteiro de *status* retornado pela rotina. Valores menores que zero indicam um erro.

pvmfrecv( ) pvm_recv( )

```
int bufid = pvm_recv(int tid, int msgtag)
call pvmfrecv(tid, msgtag, bufid)
```

A rotina `pvm_recv()` bloqueia o processo até que uma mensagem com rótulo `msgtag` tenha chegado de uma Tarefa com identificador `tid`. Ela então coloca a mensagem em um novo *buffer* de recepção ativo que foi criado. O *buffer* de recepção ativo anterior é esvaziado.

Se `pvm_recv()` for bem sucedida, `bufid` terá o valor do identificador do novo *buffer* de recepção ativo. `pvm_recv()` é bloqueada, que significa que a rotina espera até que a mensagem chegue ao *pvmd* local. Se a mensagem chegar, então `pvm_recv()` retorna imediatamente com a mensagem. Uma vez que `pvm_recv()` tenha retornado, os dados na mensagem podem ser desempacotados para a memória do usuário, usando as rotinas de desempacotamento.

Parâmetros:

`tid` identificador inteiro do processo emissor da mensagem, fornecido pelo usuário. (-1 neste argumento aceitará mensagens de qualquer Tarefa.)

msgtag inteiro usado para rotular o conteúdo da mensagem, fornecido pelo usuário. msgtag precisa ser ≥ 0. Ele permitirá ao programa do usuário distinguir entre diferentes tipos de mensagens. (-1 neste argumento aceitará mensagens com qualquer rótulo.)

bufid inteiro retornando o identificador do novo *buffer* de recepção ativo. Valores menores que zero indicam erro.

A.6.3 - Desempacotando Dados

As seguintes rotinas em C desempacotam (múltiplos) tipos de dados do *buffer* de recepção ativo. Em uma aplicação, elas devem concordar com suas correspondentes rotinas para empacotamento em tipo, número de itens, e *stride*. nitem é o número de itens que serão desempacotados e stride é o passo usado no desempacotamento para a memória do usuário.

```
int info=pvm_unpackf( const char *fmt, ...)
int info=pvm_upkbyte(   char   *xp, int nitem, int stride)
int info=pvm_upkcplx(   float  *cp, int nitem, int stride)
int info=pvm_upkdcplx(  double *zp, int nitem, int stride)
int info=pvm_upkdouble( double *dp, int nitem, int stride)
int info=pvm_upkfloat(  float  *fp, int nitem, int stride)
int info=pvm_upkint(    int    *ip, int nitem, int stride)
int info=pvm_upklong(   long   *ip, int nitem, int stride)
```

Cada uma das rotinas pvm_upk*() desempacota um vetor de um determinado tipo de dados do *buffer* de recepção ativo.

Uma exceção é pvm_upstr() que por definição desempacota uma *string* de caracteres terminada por NULL e assim não necessita dos argumentos nitem e stride.

A rotina pvm_unpackf() usa uma expressão no formato printf para indicar o tipo dos dados a serem desempacotados no *buffer* de recepção. A rotina em FORTRAN pvmfunpack(STRING,...) espera que nitem seja o número de caracteres na *string* e stride seja 1. Se o desempacotamento for bem sucedido, info será zero. Se algum erro ocorrer, info será < 0. Uma variável simples (não um vetor) pode ser desempacotada estabelecendo nitem = 1 e stride = 1.

Uma única subrotina em FORTRAN substitui todas as funções C acima.

```
call pvmfunpack(what, xp, nitem, stride, info)
```

Parâmetros:

fmt expressão do formato dos dados a serem desempacotados.

nitem número total de itens a serem desempacotados (não o número de bytes).

stride o passo a ser usado quando desempacotando os itens para a memória.

xp apontador para o começo de um bloco de bytes, onde serão desempacotados os itens. Pode ser qualquer tipo de dados, mas precisa concordar com o tipo de dado empacotado.

cp vetor de números complexos de pelo menos nitem×stride itens de tamanho.

zp vetor de números complexos em precisão dupla de pelo menos nitem×stride itens de tamanho.

dp vetor de números reais em precisão dupla de pelo menos `nitem×stride` itens de tamanho.

fp vetor de números reais de pelo menos `nitem×stride` itens de tamanho.

ip vetor de números inteiros em precisão dupla de pelo menos `nitem×stride` itens de tamanho.

jp vetor de números inteiros de pelo menos `nitem×stride` itens de tamanho.

sp apontador para uma *string* de caracteres terminada por NULL.

what inteiro indicando o tipo de dados sendo desempacotado.

STRING	0	REAL4	4	BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6	INTEGER4	3	COMPLEX16	7

info um código inteiro de *status* retornado pela rotina. Valores menores que zero indicam erro. [Sunderam 94].

Apêndice B

Algoritmos paralelos utilizados nos testes

Algoritmo B1 - Eliminação de Gauss - Resolve um sistema linear $A\mathbf{x} = \mathbf{b}$ através da Eliminação de Gauss com pivotcamento parcial usando processamento paralelo. Ao término, a matriz A conterá a matriz triangular superior resultante da eliminação. O vetor dos termos independente **b** será atualizado a cada passo da eliminação. Este é o algoritmo para a Tarefa-Mestra.

```
c   Eliminação de Gauss   - Tarefa-Mestra
c   nprocs       = número de Tarefas-Escravas a serem distribuídas
c   nla          = número de linhas da matriz A
c   nca          = número de colunas da matriz A
c   mitid        = identificador da Tarefa-Mestra
c   wtids        = vetor de identificadores das Tarefas-Escravas
c   linhas, extra = usados para determinar a quantidade de linhas
c                   da matriz a serem enviadas a cada  Tarefa-Escrava
c   linha        = posição da linha na matriz A
c   ret          = código de retorno de algumas funções do PVM
c   a            = matriz A
c   b            = vetor dos termos independentes
    call pvmfmytid ( mitid )
    if ( mitid < 0 ) then do
          print ' Tarefa-Mestra incapaz de registrar-se no PVM'
          print ' Tarefa-Mestra saindo.....'
          stop
    else
          print 'Tarefa-Mestra registrada no PVM com o número: ', mitid
    endif
    ln = nla - 1
    linhas = ln / nprocs
    extra = mod ( ln , nprocs )
    col = 1
    tam = nla
    col = 1
    index = 0
    mtfpivo( 1 ) = 1
    for i = 2 until nprocs do
        mtfpivo( i ) = 0
    endfor
    for  i = 1 until nprocs do
       mrecpivo( i ) = 1
    endfor
    for i = 1 until nla do
       npiv( i ) = i
    endfor
    colmax = 0.0D0
    for i = 1 until nla do
            hold = abs( a( npiv( i ) , col ) )
            if ( hold >  colmax ) then do
                    colmax = hold
                    nrow = i
            endif
    endfor
    ipivot = npiv( nrow )
    if (  nrow ≠ 1  ) then do
        npiv( nrow ) = npiv( col )
        npiv( col ) = ipivot
    endif
    call pvmfspawn( 'tarefa_escrava ', pvmdefault , '*' , nprocs , wtids , ret )
    msgmt = 1
    msgwk = 2
    k = 2
    uso = 1
    for  i = 1 until nprocs do
        if ( i   ≤   extra ) then do
            qtdlin = linhas + 1
        else
            qtdlin = linhas
        endif
        jpivot = npiv( k )
        call pvmfinitsend( pvmdefault , ret )
        call pvmfpack( integer4 , i , 1 , 1 , ret )
        call pvmfpack( integer4 , col , 1 , 1 , ret )
        call pvmfpack( integer4 , tam , 1 , 1 , ret )
        call pvmfpack( integer4 , qtdlin , 1 , 1 , ret )
        call pvmfpack( real8 , a( ipivot , 1 ) , tam , nla , ret )
        call pvmfpack( real8 , b( ipivot ) , 1 , 1 , ret )
        call pvmfpack( integer4 , mtfpivo( i ) , 1 , 1 , ret )
        call pvmfpack( integer4 , mrecpivo( i ) , 1 , 1 , ret )
        for j = 1 until qtdlin do
                call pvmfpack( real8 , a( jpivot , 1 ) , tam , nla , ret )
                call pvmfpack( real8 , b( jpivot ) , 1 , 1 , ret )
                call pvmfpack( integer4 , j , 1 , 1 , ret )
                call pvmfpack( integer4 , i , 1 , 1 , ret )
                call pvmfpack( integer4 , uso , 1 , 1 , ret )
                k = k + 1
                jpivot = npiv( k )
        endfor
        call pvmfsend( wtids( i ) , msgmt , ret )
    endfor
    true = 1
    linha = 0
    while ( true = 1)  do
          linha = linha + 1
          ind = 0
          for  k = 1 until  nprocs do
              if ( mrecpivo( k ) ≠ 0 ) then do
                  ind = ind + 1
                  call pvmfrecv( wtids( k ) , msgwk , ret )
                  call pvmfunpack( real8 , coluna( ind ) , 1 , 1 , ret )
                  call pvmfunpack( integer4 , pos( ind ) , 1 , 1 , ret )
                  call pvmfunpack( integer4 , qualtf( ind ) , 1 , 1 , ret )
              endif
          endfor
          colmax = 0.0D0
          for  i = 1 until ind do
              hold = abs( coluna( i ) )
              if ( hold >  colmax)  then do
                        colmax = hold
                        nrow = i
              endif
          endfor
c         Verifica a singularidade. A matriz é  considerada singular se
c         colmax  for equivalente ao zero
          if  ( um + colmax ≠ um )  then do
              go to 73
          else
              go to 250
          endif
c         As variáveis achei(1) e achei(2) informam para as Tarefas-
c         Escravas  em qual delas está  a próxima linha pivô
c         (pivotamento parcial)
     73   continue
          achei( 1 ) = pos( nrow )
          achei( 2 ) = qualtf( nrow )
          for j = 1 until nprocs do
                  call pvmfinitsend( pvmdefault , ret )
                  call pvmfpack( integer4 , achei , 2 , 1 , ret )
                  call pvmfsend( wtids( j ) , msgmt , ret )
```

```
            endfor
        for k = 1 until nprocs do
          if ( mrecpivo( k ) ≠ 0 ) then do
            call pvmfrecv( wtids( k ) , msgwk , ret )
            call pvmfunpack( integer4 , mtfpivo( k ) , 1 , 1,  ret )
            call pvmfunpack( integer4, mrecpivo( k ) , 1 , 1 , ret )
            if ( achei( 2 ) = k ) then do
              call pvmfunpack( integer4 , mcol , 1 , 1 , ret )
              call pvmfunpack( integer4 , mtam , 1 , 1 , ret )
              call pvmfunpack( real8 , mlinhapivo , mtam , 1 , ret )
              call pvmfunpack( real8 , mb , 1 , 1 , ret )
              call pvmfunpack(real8,a(linha,mcol-1), mtam-1,nla, ret)
              call pvmfunpack( real8 , b( linha ) , 1 , 1 , ret )
            endif
          endif
        endfor
c       é a última linha recebida da Tarefa-Mestra
        if ( linha  = ( nla - 1 ) ) then do
          a( nla , nla ) = mlinhapivo( 1 )
          b( nla ) = mb
c         mata os processos escravos
          for i = 1 until nprocs do
              call pvmfkill( wtids( i ) , ret )
          endfor
          go to 120
        endif
        for m = 1 until nprocs do
          if ( mrecpivo( m ) ≠ 0 ) then do
            call pvmfinitsend( pvmdefault , ret )
            call pvmfpack( integer4 , mtfpivo( m ) , 1 , 1 , ret )
            call pvmfpack( integer4 , mcol , 1 , 1 , ret )
            call pvmfpack( integer4 , mtam , 1 , 1 , ret )
            call pvmfpack( real8 , mlinhapivo , mtam , 1, ret )
            call pvmfpack( real8 , mb , 1 , 1 , ret )
            call pvmfsend( wtids( m ) , msgmt , ret )
          endif
        endfor
      endwhile
  120 continue
c     inicio da substituição regressiva.
      x( nla ) = b( nla ) / a( nla , nla )
      for i = nla - 1 until 1 passo - 1
         sum = b( i )
         for j = i + 1 until nla do
              sum = sum - a( i , j ) * x( j )
         endfor
         x( i ) = sum / a( i , i )
      endfor
      print *,' solução do sistema '
      for i = 1 until nla do
         print *,' x ( ' , i , ' ) = ',x( i )
      endfor
c     cálculo do vetor de resíduos
      for i = 1 until nla do
          v( i ) = 0.0D0
      endfor
      for j = 1 until nla do
          for i = 1 until nla do
              v( i ) = v( i ) + aa( i , j ) * x( j )
          endfor
      endfor
      for i = 1 until nla do
          resid( i ) = abs( bb( i ) - v( i ) )
      endfor
      indic = isamax( nla , resid , 1 )
      print *,' resíduo = ', resid( indic )
      go to 260
  250 continue
      print *,' matriz considerada singular '
  260 continue
      call pvmfexit( ret )
      stop
      end
```

Obs.: ISAMAX() é uma rotina do BLAS.

Algoritmo B2 - Eliminação de Gauss - Algoritmo para a Tarefa-Escrava.

```
c     nl      = tamanho de uma linha da matriz A
c     mitid   = identificador da Tarefa-Escrava
c     tidmt   = identificador da Tarefa-Mestra
c     mtotl   = número de linhas a serem alocadas na memória do processo
c               escravo
c     codret  = código de retorno de algumas rotinas do PVM
      call pvmfmytid( mitid )
      msgmt = 1
      msgwk = 2
      call pvmfparent( tidmt )
      print 'identidade da tarefa pai = ', tidmt
      print 'identidade  da Tarefa-Escrava = ', mitid
      call pvmfrecv( tidmt , msgmt , codret )
      call pvmfunpack( integer4 , wtarefa  , 1 , 1 , ret )
      call pvmfunpack( integer4 , wcol , 1 , 1 , codret )
      call pvmfunpack( integer4 , wtam , 1 , 1 , codret )
      call pvmfunpack( integer4 , wqtdlin , 1 , 1 , codret )
      call pvmfunpack( real8 , privet , wtam , 1 , codret )
      call pvmfunpack( real8 , wb1 , 1 , 1 , codret )
      call pvmfunpack( integer4 , wtfpivo , 1 , 1 , codret )
      call pvmfunpack( integer4 , wrecpivo , 1 , 1 , codret )
      for i = 1 until wqtdlin do
          call pvmfunpack( real8 , waa( i ,  1 ) , wtam , mtot , codret )
          call pvmfunpack( real8 , wbb( i ) , 1 , 1 , codret )
          call pvmfunpack( integer4 , wpos( i ) , 1 , 1 , ret )
          call pvmfunpack( integer4 , wtaf( i ) , 1 , 1 , codret )
          call pvmfunpack( integer4 , wuso( i ) , 1 , 1 , codret )
      endfor
      true = 1
      while ( true  = 1 ) do
          wx = 0
          for i = 1 until wqtdlin do
            if ( wuso( i ) ≠ 0 ) then do
                waa( i , wcol ) = waa( i , wcol ) / privet( wcol )
                for j = wcol + 1 until nl do
                   waa( i , j ) = waa(i , j)- (waa( i ,wcol )×privet(j) )
                endfor
                wbb( i ) = wbb( i ) - waa( i , wcol ) * wb1
                wx = wx + 1
            endif
          endfor
          tcol = wcol + 1
          tconta = 0
          for i = 1 until  wqtdlin do
            if( wuso( i ) ≠ 0 ) then do
                tconta = tconta + 1
                tcoluna( tconta ) = waa( i , tcol )
                tpos( tconta ) = wpos( i )
                ttaf( tconta ) = wtaf( i )
            endif
          endfor
          colmax = 0.0D0
          for i = 1 until  tconta do
              hold = abs(  tcoluna( i )  )
              if(  hold  >  colmax ) then do
                   colmax = hold
                   nrow = i
              endid
          endfor
          call pvmfinitsend( pvmdefault , codret )
          call pvmfpack( real8 , tcoluna( nrow ) , 1 , 1 , codret )
          call pvmfpack( integer4 , tpos( nrow ) , 1 , 1 , codret )
          call pvmfpack( integer4 , ttaf( nrow ) , 1 , 1 , ret )
          call pvmfsend( tidmt , msgwk , codret )
c
          call pvmfrecv( tidmt , msgmt , codret )
          call pvmfunpack( integer4 , wachei , 2 , 1 , codret )
          if (  ( wachei( 2 ) = wtarefa ) .and.  (wx = 1 ) ) then do
              wrecpivo = 0
          endif
          call pvmfinitsend( pvmdefault , codret )
          call pvmfpack( integer4 , wtfpivo , 1 , 1 , codret )
          call pvmfpack( integer4 , wrecpivo , 1 , 1 , codret )
```

```
      if( wtarefa = wachei( 2 ) ) then do
         wtam = wtam -1
         wcol = wcol + 1
         call pvmfpack( integer4 , wcol , 1 , 1 , ret )
         call pvmfpack( integer4 , wtam , 1 , 1 , ret )
         call pvmfpack(real8,waa(wachei(1),wcol ),wtam,ntot,ret)
         call pvmfpack( real8 , wbb( wachei( 1 ) ) , 1 , 1 , ret )
         call pvmfpack(real8, privet(wcol-1),wtam +1,1,codret)
         call pvmfpack( real8 , wb1 , 1 , 1 , codret )
         wuso( wachei( 1 ) ) = 0
      endif
      call pvmfsend( tidmt , msgwk , codret )
      if( wrecpivo = 0 )  go to 80
c
      call pvmfrecv( tidmt , msgmt , codret )
      call pvmfunpack( integer4 , wtfpivo , 1 , 1 , codret )
      call pvmfunpack( integer4 , wcol , 1 , 1 , codret )
      call pvmfunpack( integer4 , wtam , 1 , 1 , codret )
      call pvmfunpack( real8 , privet( wcol ) , wtam , 1 , codret )
      call pvmfunpack( real8 , wb1 , 1 , 1 , codret )
   endwhile
80 continue
   call pvmfexit( codret )
   stop
   end
```

Algoritmo B3 - Fatoração LU - Resolve um sistema linear $Ax = b$ através da Fatoração LU usando a Eliminação de Gauss com pivoteamento parcial utilizando processamento paralelo. Este é o algoritmo para a Tarefa-Mestra.

```
c  Fatoração LU   - Tarefa-Mestra
c  nprocs      = número de Tarefas-Escravas a serem distribuídas
c  nla         = número de linhas da matriz A
c  nca         = número de colunas da matriz A
c  mitid       = identificador da Tarefa-Mestra
c  wtids       = vetor de identificadores das Tarefas-Escravas
c  linhas,extra = usados para determinar a quantidade de linhas da matriz A
c                 a serem enviadas a cada  Tarefa-Escrava
c  linha       = posição da linha na matriz A
c  ret         = código de retorno de algumas funções do PVM
c  a           = a matriz A
c  b           = vetor dos termos independentes
   parameter ( nla  = 500 )
   parameter ( nca  = 500 )
   parameter ( nprocs = 5 )
   integer msgmt , msgwk , mcol , tam , mtam , mtfpivo( nprocs )
   integer mrecpivo( nprocs ) , nrow , ret
   integer pos( nla ) , qualtf( nla ) , uso , linha , aux1
   integer qtdlin , mqtdlin , extra , linhas , ind
   integer i , j , k , m , t , col , wcol , mitid , npiv( nla ) , jpivot , mjpiv
   integer ln , wtids( nprocs ) , mudou , achei( 2 ) , quanti
   real*8 a( nla , nca ) , y( nla ), x( nla ) , aux , b( nla ) , resid( nla )
   real*8 aa( nla , nca ) , bb( nla ) , v( nla ) , coluna( nla )
   real*8 mlinhapivo( nla ) , colmax , hold , mb , zero , um
   data zero/0.0D0/ , um /1.0D0/
   call pvmfmytid( mitid )
   if ( mitid  < 0 ) then do
        print ' Tarefa-Mestra incapaz de registrar-se no PVM '
        print ' Tarefa-Mestra saindo.....'
        stop
   else
        print 'Tarefa-Mestra registrada com o número:',mitid
   endif
   ln = nla - 1
   linhas = ln / nprocs
   extra = mod( ln , nprocs )
   col = 1
   tam = nla
   col = 1
   index = 0
   mtfpivo( 1 ) = 1
   for  i = 2 until  nprocs do
      mtfpivo( i ) = 0
   endfor
   for i = 1 until  nprocs do
      mrecpivo( i ) = 1
   endfor
   for i = 1 until  nla do
      npiv( i ) = i
   enfor
   colmax = 0.0D0
c  pesquisa pelo maior valor absoluto na primeira coluna (pivoteamento
c  parcial)
   for i = 1 until nla do
        hold = abs( a( npiv ( i ) , col ) )
        if ( hold >  colmax) then do
           colmax = hold
           nrow = i
        endif
   endfor
   ipivot = npiv( nrow )
   if( nrow  = 1) go to 15
      npiv( nrow ) = npiv( col )
      npiv( col ) = ipivot
15 continue
   call pvmfspawn(' tarefa_escrava',pvmdefault,'*', nprocs,wtids,ret)
```

```
      msgmt = 1
      msgwk = 2
      k = 2
      uso = 1
      for i = 1 until nprocs do
          if ( i  ≤ extra ) then do
            qtdlin = linhas + 1
          else
            qtdlin = linhas
          endif
          jpivot = npiv( k )
          call pvmfinitsend( pvmdefault , ret )
          call pvmfpack( integer4 , i , 1 , 1 , ret )
          call pvmfpack( integer4 , col , 1 , 1 , ret )
          call pvmfpack( integer4 , tam , 1 , 1 , ret )
          call pvmfpack( integer4 , qtdlin , 1 , 1 , ret )
          call pvmfpack( real8 , a( ipivot , 1 ) , tam , nla , ret )
          call pvmfpack( real8 , b( ipivot ) , 1 , 1 , ret )
          call pvmfpack( integer4 , mtfpivo( i ) , 1 , 1 , ret )
          call pvmfpack( integer4 , mrecpivo( i ) , 1 , 1 , ret )
          for  j = 1 until qtdlin do
              call pvmfpack( real8 , a( jpivot , 1 ) , tam , nla , ret )
              call pvmfpack( real8 , b( jpivot ) , 1 , 1 , ret )
              call pvmfpack( integer4 , j , 1 , 1  , ret )
              call pvmfpack( integer4 , i , 1 , 1  , ret )
              call pvmfpack( integer4 , uso , 1 , 1 , ret )
              k = k + 1
              jpivot =  npiv( k )
          endfor
          call pvmfsend( wtids( i ) , msgmt , ret )
      endfor
      true  = 1
      linha = 0
      while ( true  =  1 ) do
          linha = linha + 1
          ind = 0
          for k = 1 until nprocs do
            if ( mrecpivo( k ) ≠  0 ) then do
              ind = ind + 1
              call pvmfrecv( wtids( k ) , msgwk , ret )
              call pvmfunpack( real8 , coluna( ind ) , 1 , 1 , ret )
              call pvmfunpack( integer4 , pos( ind ) , 1 , 1 , ret )
              call pvmfunpack( integer4 , qualtf( ind ) , 1 , 1  , ret )
            endif
          endfor
          colmax = 0.0D0
          for i = 1 until ind do
              hold = abs( coluna( i ) )
              if ( hold  > colmax ) then do
                  colmax = hold
                  nrow = i
              endif
          endfor
c         Verifica a singularidade. A matriz é  considerada singular se
c         colmax for equivalente ao zero
          if  ( um + colmax ≠ um ) then do
              go to 73
          else
              go to 250
          endif
c         As variáveis achei(1) e achei(2) informam para as Tarefas-
c         Escravas  em qual delas está  a próxima linha pivô
c         (pivotamento  parcial)
 73       continue
          achei( 1 ) = pos( nrow )
          achei( 2 ) = qualtf( nrow )
          for  j = 1 until  nprocs do
            call pvmfinitsend( pvmdefault , ret )
            call pvmfpack( integer4 , achei , 2 , 1  , ret )
            call pvmfsend( wtids( j ) , msgmt , ret )
          endfor
          for  k = 1 until  nprocs  do
              if ( mrecpivo( k ) ≠  0)  then do
                  call pvmfrecv( wtids( k ) , msgwk , ret )
                  call pvmfunpack( integer4 , mtfpivo( k ) , 1 , 1 , ret )
                  call pvmfunpack( integer4 , mrecpivo( k ) , 1 , 1 , ret )
                  if ( achei( 2 ) =  k ) then do
                      call pvmfunpack( integer4 , mcol , 1 , 1 , ret )
                      call pvmfunpack( integer4 , mtam , 1 , 1 , ret )
                      call pvmfunpack( real8 , mlinhapivo , nla , 1 , ret )
                      call pvmfunpack( real8 , mb , 1 , 1 , ret )
                      call pvmfunpack( real8 , a( linha , 1 ) , nla , nla , ret )
                      call pvmfunpack( real8 , b( linha ) , 1 , 1 , ret )
                  endif
              endif
          endfor
c         é a última linha pivô
          if ( linha  =  ( nla - 1 ) ) then do
              for i = 1 until nla do
                  a( nla , i ) = mlinhapivo( i )
              endfor
              b( nla ) = mb
c             mata os processos escravos
              for  i = 1 until nprocs do
                  call pvmfkill( wtids( i ) , ret )
              endfor
              go to 120
          endif
          for m = 1 until nprocs do
            if ( mrecpivo( m )  ≠  0) then do
                call pvmfinitsend( pvmdefault , ret )
                call pvmfpack( integer4 , mtfpivo( m ) , 1 , 1 , ret )
                call pvmfpack( integer4 , mcol , 1 , 1 , ret )
                call pvmfpack( integer4 , mtam , 1 , 1 , ret )
                call pvmfpack( real8 , mlinhapivo , nla , 1 , ret )
                call pvmfpack( real8 , mb , 1 , 1 , ret )
                call pvmfsend( wtids( m ) , msgmt , ret )
            endif
          endfor
      endwhile
120   continue
c     Início da substituição progressiva. O resultado é armazenado em y
c     Resolve a equação Ly = b
      y( 1 ) = b( 1 )
      for  k = 2 until nla do
        aux = 0.0D0
        for  i = 1 until k-1 do
            aux = aux + ( a ( k , i ) x y( i )  )
        endfor
        y( k ) = b( k ) - aux
      endfor
c     inicio da substituição regressiva.
      x( nla ) = y( nla ) / a( nla , nla )
      for i = nla - 1 until 1 passo -1 do
        sum = y( i )
        for j = i + 1 until nla do
            sum = sum - a( i , j )  x  x( j )
        enfor
        x( i ) = sum /  a( i , i )
      enfor
c     cálculo do vetor de resíduos
      for i = 1 until nla do
        v( i ) =  0.0D0
      endfor
      for  j = 1 until  nla do
        for  i = 1 until nla do
            v( i ) = v( i )  +  aa( i , j ) x x( j )
        endfor
      endfor
      for  i = 1 until nla do
        resid( i ) = abs(  bb( i ) - v( i ) )
      endfor
      indic = isamax( nla , resid , 1 )
      print *,' residuo = ' , resid( indic )
      go to 260
250   continue
      print *,' matriz considerada singular '
260   continue
      call pvmfexit( ret )
      stop
      end
```

Obs.: ISAMAX() é uma rotina do BLAS.

Algoritmo B4 - Fatoração LU - Algoritmo para a Tarefa-Escrava.

```
c  Fatoração LU - Tarefa-Escrava
c  nl      = tamanho de uma linha da matriz A
c  mitid   = identificador da Tarefa-Escrava
c  tidmt   = identificador da Tarefa-Mestra
c  ntotl   = número de linhas a serem alocadas na memória do
c            processo escravo
c  codret  = código de retorno de algumas rotinas do PVM
   include 'fpvm3.h'
   parameter( nl = 500 )
   parameter( nprocs = 5)
   parameter( ntot = ( ( nl - 1 ) / nprocs ) + 1 )
   integer i , codret , mitid , tidmt , wcol , wlinha , wqtdlin , wtfpivo
   integer wrecpivo , wtarefa , tcol , tconta , wx , nrow
   integer tpos( ntot ) , ttaf( ntot ) , wachei( 2 ) , wpos( ntot )
   integer ln , msgmt , msgwk , wtam , wnpiv( ntot ) , wuso( ntot )
   integer wtaf( ntot )
   real*8 privet( nl ) , waa( ntot , nl ) , tcoluna ( ntot ) , wb1 , wbb( ntot )
   real*8 colmax , hold
   call pvmfmytid( mitid )
   msgmt = 1
   msgwk = 2
   call pvmfparent( tidmt )
   print 'identidade da tarefa pai = ', tidmt
   print 'identidade da Tarefa-Escrava = ', mitid
   call pvmfrecv( tidmt , msgmt , codret )
   call pvmfunpack( integer4 , wtarefa , 1 , 1 , ret )
   call pvmfunpack( integer4 , wcol , 1 , 1 , codret )
   call pvmfunpack( integer4 , wtam , 1 , 1 , codret )
   call pvmfunpack( integer4 , wqtdlin , 1 , 1 , codret )
   call pvmfunpack( real8 , privet , wtam , 1 , codret )
   call pvmfunpack( real8 , wb1 , 1 , 1 , ret )
   call pvmfunpack( integer4 , wtfpivo , 1 , 1 , codret )
   call pvmfunpack( integer4 , wrecpivo , 1 , 1 , codret )
   for  i = 1 until wqtdlin do
        call pvmfunpack( real8 , waa( i , 1 ) , wtam , ntot , codret )
        call pvmfunpack( real8 , wbb( i ) , 1 , 1 , ret )
        call pvmfunpack( integer4 , wpos( i ) , 1 , 1 , ret )
        call pvmfunpack( integer4 , wtaf( i ) , 1 , 1 , codret )
        call pvmfunpack( integer4 , wuso( i ) , 1 , 1 , codret )
   endfor
   true = 1
   while ( true = 1 ) do
      wx = 0
      for i = 1 until wqtdlin do
          if ( wuso( i ) ≠ 0 ) then do
             waa( i , wcol ) = waa( i , wcol ) / privet( wcol )
             for j = wcol + 1 until nl do
                 waa(i ,j ) = waa(i , j)- ( waa(i , wcol )× privet( j ) )
             endfor
             wx = wx + 1
          endif
      endfor
      tcol = wcol + 1
      tconta = 0
      for  i = 1 until  wqtdlin do
          if ( wuso( i ) ≠ 0 ) then do
                tconta = tconta + 1
                tcoluna( tconta ) = waa( i , tcol )
                tpos( tconta ) = wpos( i )
                ttaf( tconta ) = wtaf( i )
          endif
      endfor
      colmax = 0.0D0
      for i = 1 until tconta do
            hold = abs( tcoluna( i ) )
            if ( hold > colmax ) then do
                colmax = hold
                nrow = i
            endif
      endfor
      call pvmfinitsend( pvmdefault , codret )
      call pvmfpack( real8 , tcoluna( nrow ) , 1 , 1 ,  codret )
      call pvmfpack( integer4 , tpos( nrow ) , 1 , 1 ,  codret )
      call pvmfpack( integer4 , ttaf( nrow ) , 1 , 1 , ret )
      call pvmfsend( tidmt , msgwk , codret )
c
      call pvmfrecv( tidmt , msgmt , codret )
      call pvmfunpack( integer4 , wachei , 2 , 1 , codret )
      if ( ( wachei(2) = wtarefa) .and. (wx = 1) ) then do
            wrecpivo = 0
      endif
      call pvmfinitsend( pvmdefault , codret )
      call pvmfpack( integer4 , wtfpivo , 1 , 1 , codret )
      call pvmfpack( integer4 , wrecpivo , 1 , 1 , codret )
      if ( wtarefa = wachei( 2 ) ) then do
          wtam = wtam - 1
          wcol = wcol + 1
          call pvmfpack( integer4 , wcol , 1 , 1 , ret )
          call pvmfpack( integer4 , wtam , 1 , 1 , ret )
          call pvmfpack(real8,waa(wachei(1),1) ,nl ,ntot, ret)
          call pvmfpack( real8 , wbb( wachei( 1 ) ),1,1,ret)
          call pvmfpack( real8 , privet , nl , 1 , codret )
          call pvmfpack( real8 , wb1 , 1 , 1 , ret )
          wuso( wachei( 1 ) ) = 0
      endif
      call pvmfsend( tidmt , msgwk , codret )
c
      if( wrecpivo = 0 ) go to 80
      call pvmfrecv( tidmt , msgmt , codret )
      call pvmfunpack( integer4 , wtfpivo , 1 , 1 , codret )
      call pvmfunpack( integer4 , wcol , 1 , 1 , codret )
      call pvmfunpack( integer4 , wtam , 1 , 1 , codret )
      call pvmfunpack( real8 , privet , nl , 1 , codret )
      call pvmfunpack( real8 , wb1 , 1 , 1 , ret )
   endwhile
80 continue
   call pvmfexit( codret )
   stop
   end
```

Algoritmo B5 - Método iterativo de Gauss-Jacobi - Resolve um sistema linear $Ax = b$ através do método iterativo de Gauss-Jacobi. A matriz A deve ser estritamente diagonal dominante. Este é o algoritmo para a Tarefa-Mestra.

```
c   Método Iterativo de Gauss-Jacobi - Tarefa-Mestra
c   nprocs      = número de Tarefas-Escravas a serem distribuídas
c   nla         = número de linhas da matriz A
c   nca         = número de colunas da matriz A
c   mitid       = identificador da Tarefa-Mestra
c   wtids       = vetor de identificadores das Tarefas-Escravas
c   linhas,extra = usados para determinar a quantidade de linhas da
c                  matriz A a serem enviadas a cada Tarefa-Escrava
c   linha       = posição da linha na matriz A
c   ret         = código de retorno de algumas funções do PVM
c   a           = matriz A diagonal dominante estrita
c   b           = vetor dos termos independentes
c   x           = vetor de aproximação inicial
c   eps         = precisão requerida
c   maxit       = número máximo de iterações declarado
c   ite         = variável de controle
    include 'fpvm3.h'
    parameter( nprocs = 5 )
    parameter( nla = 2000 )
    parameter( nca = 2000 )
    integer linhas , extra , linha , msgmt , msgwk , i , j , h , ite , maxit
    integer ret , wtids( nprocs ) , bufid , tidmet , mitid , qtdlin
    integer mlinha , mqtdlin , index
    real*8 a( nla , nca ) , b( nla ) , xold( nla ) , x( nla ) , eps , v( nla )
    real*8  resid( nla )
    real*8 maior , aux , sum , vetor( nla x nca ) , wa( nla , nca )
    maxit = 50
    eps  = 1.0D-12
c   registra a Tarefa-Mestra no pvm
    call pvmfmytid( mitid )
    If ( mitid  <  0 ) then do
          print ' Tarefa-Mestra incapaz de registrar-se no PVM '
          print ' Tarefa-Mestra saindo..... '
          stop
    else
         print ' Tarefa-Mestra registrada com o número: ', mitid
    endif
c   inicialização do vetor  de aproximação inicial
    for  i = 1 until nla do
        x( i ) = 0.0D0
    endfor
c
    for i = 1 until nla do
       xold( i ) = x( i )
    endfor
c   determina a quantidade de linhas a serem enviadas a cada Tarefa-
c   Escrava
    linhas = nla / nprocs
    extra = mod ( nla , nprocs )
    linha = 1
c   distribuição das Tarefas-Escravas na máquina virtual
    call pvmfspawn(' tarefa_escrava ',pvmdefault ,'* ',nprocs,wtids,ret)
    for  i = 1 until nprocs do
       If ( i  ≤  extra ) then do
            qtdlin = linhas + 1
       else
           qtdlin = linhas
       endif
c      envia dados para as Tarefas-Escravas
c      linha   = é a posição na matriz.
c      qtdlin  = quantidade de linhas da matriz A, que serão enviadas para
c                cada Tarefa-Escrava
c      b       = qtdlin elementos do vetor independente b
c      xold    = vetor de aproximação inicial
c      a       = as qtdlin linhas da matriz A
       msgmt = 1
       call pvmfinitsend( pvmdefault , bufid )
       call pvmfpack( integer4 , linha , 1 , 1 , ret )
       call pvmfpack( integer4 , qtdlin , 1 , 1 , ret )
       call pvmfpack( real8 , b( linha ) , qtdlin , 1 , ret )
       call pvmfpack( real8 , xold , nla , 1 , ret )
       for j = 1 until qtdlin do
          call pvmfpack( real8 , a( linha , 1 ) , nla , nla , ret )
          linha = linha + 1
       endfor
       call pvmfsend( wtids( i ) , msgmt , ret )
    endfor
c   espera pelos resultados das Tarefas-Escravas
c   mlinha     = posição inicial no vetor x
c   mqtdlin    = número de elementos a serem desempacotados em x  a
c                partir da posição x(mlinha)
c   x(mlinha)  = os mqtdlin elementos do novo vetor iteração
    ite = 0
    while( ite ≤ maxit ) do
          ite = ite + 1
          msgwk = 2
          for  i = 1 until nprocs do
             call pvmfrecv( wtids( i ) , msgwk , bufid )
             call pvmfunpack( integer4 , mlinha , 1 , 1 , ret )
             call pvmfunpack( integer4 , mqtdlin , 1 , 1 , ret )
             call pvmfunpack( real8 , x( mlinha ) , mqtdlin , 1 , ret )
          endfor
c         faz o teste de parada
          aux = x( 1 ) -  xold( 1 )
          maior = abs( aux )
          for  i = 2 until nla do
             aux = x( i ) - xold( i )
             aux = abs(  aux )
             If ( aux  >  maior ) then do
                maior = aux
             endif
          endfor
          if (  maior  ≤  eps ) then do
             go to 160
          endif
          for  i = 1 until  nla do
             xold( i ) = x( i )
          endfor
c         envia o novo vetor iteração para as Tarefas-Escravas
          for i = 1 until nprocs do
             call pvmfinitsend( pvmdefault , bufid )
             call pvmfpack( real8 , xold , nla , 1 , ret )
             call pvmfsend( wtids( i ) , msgmt , ret )
          endfor
    endwhile
160 continue
    print ' solução: '
    for i = 1 until  nla do
       print  ' x( ',i,') = ', x( i )
    endfor
c   mata os processos escravos
    for i = 1 until nprocs do
       call pvmfkill( wtids( i ) , ret )
    endfor
c   cálculo do vetor de resíduos
    for i = 1 until nla do
       v( i ) = 0.0D0
    endfor
c
    for  i =  1 until  nla do
       aux = 0.0D0
       for j = 1 until nla do
          aux = aux + a( i , j ) * x( j )
       endfor
       v( i ) = aux
    endfor
c
    for  i = 1 until nla do
     resid( i ) = abs(  b( i ) - v( i ) )
    endfor
    indic = isamax( nla , resid , 1 )
    print ' resíduo = ', resid( indic )
```

```
      print ' número de iterações máximo declarado = ', maxit
      print ' número de iterações usadas =', ite
      call pvmfexit( ret )
      stop
      end
```

Algoritmo B6 - Método Iterativo de Gauss-Jacobi - Algoritmo para a Tarefa-Escrava.

```
c     Método iterativo de Gauss-Jacobi - Tarefa-Escrava
c     nl        = tamanho de uma linha da matriz A
c     mitid     = identificador da Tarefa-Escrava
c     tidmt     = identificador da Tarefa-Mestra
c     ntotl     = número de linha a serem alocadas na memória do
c                 processo escravo
c     codret    = código de retorno de algumas rotinas do PVM
      include 'fpvm3.h'
c
      parameter( nl = 2000 )
      parameter( nprocs = 5 )
      parameter( ntotl = ( ( nl - 1) / nprocs ) + 1 )
      integer codret , mitid , tidmt , contr
      integer msgmt , msgwk , i , j , wlinha , wqtdlin , true
      real*8 wa( ntotl , nl ) , wb( ntotl )
      real*8 wx( ntotl ) , wxold( nl ) , soma
c     registro da Tarefa-Escrava no PVM
      call pvmfmytid( mitid )
      if (mitid  <  0) then do
          print ' Tarefa-Escrava incapaz de registrar-se no PVM '
          print  ' Tarefa-Escrava saindo..... '
          stop
      else
          print  ' Tarefa-Escrava registrada com o número: ',mitid
      endif
c     recebe uma mensagem da Tarefa-Mestra  contendo:
c     wlinha   = indica a posição na matriz A, indicando a primeira
c                linha enviada pela Tarefa-Mestra.
c     wqtdlin = quantidade de linhas enviadas a Tarefa-Escrava, a
c                partir da linha wlinha
c     wb       = contém wqtdlin elementos do vetor independente,
c                começando pelo elemento na posição  b(wlinha).
c     wxold    = o vetor iteração, contendo nla elementos
c     wa       = matriz que irá armazenar as wqtdlin linhas da matriz A.
      msgmt = 1
      msgwk = 2
      call pvmfparent( tidmt )
      call pvmfrecv( tidmt , msgmt , codret )
      call pvmfunpack( integer4 , wlinha , 1 , 1 , codret )
      call pvmfunpack( integer4 , wqtdlin , 1 , 1 , codret )
      call pvmfunpack( real8 , wb , wqtdlin , 1 , codret )
      call pvmfunpack( real8 , wxold , nl , 1 , codret )
      contr = wlinha
      for i = 1 until wqtdlin do
         call pvmfunpack( real8 , wa( i , 1 ) , nl  , ntotl , codret )
      endfor
c     cálculo das wqtdlin componentes do novo vetor iteração
      true = 1
      while ( true  =  1) do
            contr = wlinha
            for  i = 1 until wqtdlin do
                 soma = 0.0D0
                 for j = 1 until nl do
                     if( contr ≠ j ) then do
                        soma = soma + ( wa( i , j ) x wxold( j ) )
                     endif
                 endfor
                 wx( i ) = ( wb( i ) – soma ) / wa( i , contr )
                 contr = contr + 1
            endfor
c           envia o resultado da operação de volta à Tarefa-Mestra
c           wlinha  = indica a posição do vetor x onde deve começar
c                     o desempacotamento dos  wqtdlin elementos do
c                     novo vetor  iteração
c           wqtdlin = indica a quantidade de elementos da nova iteração
c                     que serão enviados para a  Tarefa-Mestra
c           wx      = os wqtdlin elementos que farão parte do novo vetor
c                     iteração
            call pvmfinitsend( pvmdefault , codret )
            call pvmfpack( integer4 , wlinha , 1 , 1 , codret )
            call pvmfpack( integer4 , wqtdlin , 1 , 1 , codret )
            call pvmfpack( real8 , wx , wqtdlin , 1 , codret )
            call pvmfsend( tidmt , msgwk , codret )
c
            call pvmfrecv( tidmt , msgmt , codret )
            call pvmfunpack( real8 , wxold , nl , 1 , codret )
      endwhile
      call pvmfexit( codret )
      stop
      end
```

Algoritmo B7 - Gradientes Conjugados - Resolve um sistema linear $Ax = b$ (a matriz A simétrica positiva definida) através do método iterativo dos Gradientes Conjugados usando processamento paralelo. Este é o algoritmo para a Tarefa-Mestra.

```
c
c  versão na linguagem FORTRAN da  Tarefa-Mestra para o ambiente  pvm.
c  A  Tarefa-Mestra é responsável pela criação das  Tarefas-Escravas que vão
c  trabalhar na resolução do problema, coordenação da entrada dos dados
c  iniciais para cada tarefa  escrava e pela coleta dos seus resultados  parciais.
c  A  Tarefa-Mestra distribui nprocs Tarefas-Escravas na máquina virtual.
c     nprocs        = número de Tarefas-Escravas a serem distribuidas
c     nla           = número de linhas da matriz A
c     nca           = número de colunas da matriz A
c     mitid         = identificador da Tarefa-Mestra
c     wtids         = vetor de identificadores das Tarefas-Escravas
c     linhas , extra = usados para determinar a quantidade de linhas da
c                      matriz A  serem enviadas a  cada Tarefa-Escrava
c     linha         = posição da linha na matriz A
c     ret           = código de retorno de algumas funções
c     a             = matriz A
c     b             = vetor dos termos independentes
c     x             = vetor de aproximação inicial
c     eps           = precisão requerida
c     maxit         = número máximo de iterações declarado
c     ite           = variável de controle
c
c  cada Tarefa-Escrava recebe:
c        (nla / nprocs) linhas da matriz A
c        (nla / nprocs) elementos do vetor independente b
c        e o vetor de aproximação inicial x
c
      include 'fpvm3.h'
      parameter( nprocs = 2 )
      parameter( nla = 4 )
      parameter( nca = 4 )
      integer linhas , extra , linha , msgmt , msgwk , i , j , h , ite , maxit
      integer ret , wtids( nprocs ) , bufid , tidmet , mitid , qtdlin
      integer mlinha , mqtdlin , date_time(8)
      real*8 a( nla , nca ) , r( nla ) , b( nla ) , v( nla ) , x( nla )
      real*8 rsd( nla ) , z( nla ) , soma , t , delta , c , d
      delta = 1.0D–12
      maxit = 50
c     gera a matriz simétrica positiva definida
      for  i = 1  until  nla do
           soma = 0.0D0
           for  j = 1until  nla  do
                if ( j  < i ) then do
                        a( i , j ) = a( j , i )
                else
                        a( i , j ) = dble( float( i + j ) )
                endif
                soma  = soma + dabs( a( i , j ) )
           enfor
           a( i , i ) = soma + soma
      endfor
c     gera o vetor independente de modo que a solução do sistema linear seja
```

```
c   (1,1,1,..,1)
    for  i = 1 until  nla  do
            soma = 0.0D0
            for  j = 1 until  nla  do
                    soma = soma + A( i , j )
            endfor
            b( i ) = soma
     endfor
    for  i = 1 until nla do
         x( i ) = 0.0D0
    endfor
    call resid( nla , A , x , b , r )
    for  i = 1 until nla do
         v( i ) = r( i )
    endfor
    c = prod( nla , r , r )
    linhas = nla / nprocs
    extra = mod( nla , nprocs )
    linha = 1
c
c   distribui as Tarefas-Escravas
c
    call pvmfspawn('./tarefa_escrava' ,pvmdefault , '*',nprocs,wtids,ret)
    for  i = 1  until nprocs do
            if ( i  ≤  extra ) then do
                  qtdlin = linhas + 1
            else
                  qtdlin = linhas
            endif
            msgmt = 1
            call pvmfinitsend( pvmdefault , bufid )
            call pvmfpack( integer4 , linha , 1 , 1 , ret )
            call pvmfpack( integer4 , qtdlin , 1 , 1 , ret )
            call pvmfpack( real8 , v ,  nla , 1 , ret )
            for  j = 1 until qtdlin do
               call pvmfpack( real8 , A( linha , 1 ) , nla , nla , ret )
               linha = linha + 1
            endfor
            call pvmfsend(wtids( i ) , msgmt , ret )
     endfor
c
for k = 1  until  maxit  do
      if ( sqrt(   prod( nla , v , v )  ) < delta) go to 100
      msgwk = 2
c
c     recebe das Tarefas-Escravas vetores que vão compor o vetor z
c
      for  i = 1 until nprocs do
           call pvmfrecv( wtids( i ) , msgwk , bufid )
           call pvmfunpack( integer4 , mlinha , 1 , 1 , ret )
           call pvmfunpack( integer4 , mqtdlin , 1 , 1 , ret )
           call pvmfunpack( real8 , z( mlinha ) , mqtdlin , 1 , ret )
      endfor
c
      t = c / prod( nla , v , z )
      for  i = 1 until nla do
           x( i ) = x( i ) + t × v( i )
      endfor
      for  i = 1 until nla do
         r( i ) = r( i ) - t × z( i )
      endfor
c
      d = prod( nla , r , r )
c
      for  i = 1 until nla do
         v( i ) = r( i ) + (d / c) × v( i )
      endfor
c
      c = d
c     envia o novo vetor  v para as Tarefas-Escravas
c
      for  i = 1  until nprocs do
           call pvmfinitsend( pvmdefault , bufid )
           call pvmfpack( real8 , v , nla , 1 , ret )
           call pvmfsend( wtids( i ) , msgmt , ret )
      endfor
      print ' iteracao = ',k
c
endfor
100 continue
c   mata as Tarefas-Escravas
for  i = 1 until nprocs do
     call pvmfkill( wtids( i ) , ret )
endfor
print ,'a solucao é :'
for  i = 1  until  nla do
     print 'x(' , i , ') = ', x( i )
endfor
print   ' o vetor residuo é:'
for  i = 1  until  nla do
      print 'r(' , i , ' ) = ' , r( i )
endfor
call pvmfexit(ret)
stop
c
function prod( nla , x , y )
c
c   computa o produto vetorial
c
    real*8 x( nla ) , y( nla ) , soma
    soma  =  0.0D0
    for  i = 1 until nla do
         soma  =  soma + x( i ) × y( i )
    endfor
    prod  =  soma
    return
    end
c
    subroutine resid( nla , A , x , b , r )
c
c   computa o resíduo   r = b – Ax
c
    integer i
    real*8 a( nla , nla ) , x( nla ) , b( nla ) , r( nla )
    call mult( nla , A , x , r )
    for  i = 1 until nla do
        r( i ) = b( i ) – r( i )
    endfor
c
    return
    end
c
    subroutine mult( nla , A , x , y )
c
c   computa o produto matriz A pelo  x  vetor
c
    real*8 a( nla , nla ) , x( nla ) , y( nla ) , soma
    for  i = 1 until nla do
         soma  =  0.0d0
         for j = 1 until nla do
                soma  =  soma + A( i , j ) × x( j )
           endfor
           y( i ) =  soma
    endfor
c
    return
    end
```

Algoritmo B8- Método iterativo dos Gradientes Conjugados - Este é o algoritmo para a Tarefa-Escrava.

```
c   Método dos Gradientes Conjugados - Tarefa-Escrava
c   nprocs      = número de Tarefas-Escravas a serem distribuidas
c
c   descrição: recebe da Tarefa-Mestra wqtdlin linhas da matriz A  e
c   wqtdlin elementos do vetor v  e devolve à Tarefa-Mestra wqtdlin
c   elementos do novo vetor z
c
c   nl      = tamanho de uma linha da matriz A
c   mitid   = identificador da Tarefa-Escrava
c   tidmt   = identificador da Tarefa-Mestra
c   codret = código de retorno de algumas rotinas
    include 'fpvm3.h'
c
    parameter(nl = 2000)
    parameter(nprocs = 5)
    parameter(ntotl = ( (nl – 1) / nprocs) + 1 )
    integer codret , mitid , tidmt , contr
    integer msgmt , msgwk , i , j , wlinha , wqtdlin , true
    real*8  wa( ntotl , nl )
    real*8 wy( ntotl ) , wv( nl ) , soma
    call pvmfmytid( mitid )
    if( mitid  <  0) then do
       print ' Tarefa-Escrava incapaz de registrar-se no PVM'
       print ' Tarefa-Escrava  saindo.....'
    else
       print   ' Tarefa-Escrava registrada com o número:',mitid
    endif
    msgmt = 1
    msgwk = 2
    call pvmfparent( tidmt )
    call pvmfrecv( tidmt , msgmt , codret )
    call pvmfunpack( integer4 , wlinha , 1 , 1 , codret )
    call pvmfunpack( integer4 , wqtdlin , 1 , 1 , codret )
    call pvmfunpack( real8 , wv , nl , 1 , codret )
    for i  = 1 until wqtdlin do
           call pvmfunpack( real8 , wa( i , 1 ) , nl , ntotl , codret )
    endfor
c
c   cálculo das wqtdlin componentes do próximo vetor z
c
    true  = 1
    while ( true  =  1) do
              for  i  = 1 until  wqtdlin  do
                 soma = 0.0D0
                 for  j  = 1 until   nl   do
                    soma = soma + ( wa( i , j ) x  wv( j )  )
                 endfor
                 wy( i ) = soma
           endfor
c
c  envia o resultado da operacao de volta a Tarefa-Mestra
c
      call pvmfinitsend( pvmdefault , codret)
      call pvmfpack( integer4 , wlinha , 1 , 1  , codret )
      call pvmfpack( integer4 , wqtdlin , 1 , 1 ,  codret )
      call pvmfpack( real8 , wy , wqtdlin , 1 , codret )
      call pvmfsend( tidmt ,  msgwk , codret )
c
c    recebe o novo vetor  v
c
      call pvmfrecv( tidmt , msgmt , codret )
      call pvmfunpack( real8 , wv , nl , 1 , codret )
endwhile
call pvmfexit(codret)
stop
end
```

Algoritmo B9 - Eliminação de Gauss - Resolve um sistema linear $Ax = b$ através da Eliminação de Gauss com pivoteamento parcial e **escalamento** usando processamento paralelo. Ao término, a matriz A conterá a matriz triangular superior resultante da eliminação. O vetor independente será atualizado a cada passo da eliminação. Este é o algoritmo para a Tarefa-Mestra.

```
c   Eliminação de Gauss    - Tarefa-Mestra
c   nprocs      = número de Tarefas-Escravas a serem distribuidas
c   nla         = número de linhas da matriz A
c   nca         = número de colunas da matriz A
c   mitid       = identificador da Tarefa-Mestra
c   wtids       = vetor de identificadores das Tarefas-Escravas
c   linhas, extra = usados para determinar a quantidade de linhas da
c                   matriz a serem enviadas a cada  Tarefa-Escrava
c   linha       = posição da linha na matriz A
c   ret         = código de retorno de algumas funções do PVM
c   a           = matriz A
c   b           = vetor dos termos independentes
    call pvmfmytid ( mitid )
    if ( mitid  < 0  ) then do
         print ' Tarefa-Mestra incapaz de registrar-se  no PVM'
         print ' Tarefa-Mestra saindo.....'
         stop
    else
         print 'Tarefa-Mestra registrada no PVM com o número: ', mitid
    endif
    ln = nla – 1
    linhas = ln / nprocs
    extra = mod ( ln , nprocs )
    col = 1
    tam = nla
    col = 1
    index = 0
    mtfpivo( 1 ) = 1
    for i = 2 until nprocs do
       mtfpivo( i ) = 0
    endfor
    for  i = 1 until nprocs do
       mrecpivo( i ) = 1
    endfor
    for i = 1 until nla do
       npiv( i ) = i
       d(i) = 0                                                    *
       for j = 1 until nla do                                      *
          d(i) = DMAX1( d(i), DABS( a( i , j ) ) )                 *
       endfor                                                      *
       if ( d(i) =  0 ) then do                                    *
          d(i) = 1.0D0                                             *
       endif                                                       *
    endfor
    colmax = 0.0D0
    for i = 1 until nla do
          hold = abs(  a( npiv( i ) , col ) / d( npiv(i))   ) *
          if ( hold  >  colmax ) then do
             colmax = hold
             nrow = i
          endif
    endfor
    ipivot = npiv( nrow )
    if (  nrow  ≠  1  ) then do
        npiv( nrow ) = npiv( col )
        npiv( col ) = ipivot
    endif
    call pvmfspawn('tarefa_escrava',pvmdefault,'*',nprocs,wtids,ret)
    msgmt = 1
    msgwk = 2
    k = 2
    uso = 1
```

```
      for i = 1 until nprocs do
          if ( i   ≤   extra ) then do
              qtdlin = linhas + 1
          else
              qtdlin = linhas
          endif
          jpivot = npiv( k )
          call pvmfinitsend( pvmdefault , ret )
          call pvmfpack( integer4 , i , 1 , 1 , ret )
          call pvmfpack( integer4 , col , 1 , 1 , ret )
          call pvmfpack( integer4 , tam , 1 , 1 , ret )
          call pvmfpack( integer4 , qtdlin , 1 , 1 , ret )
          call pvmfpack( real8 , a( ipivot , 1 ) , tam , nla , ret )
          call pvmfpack( real8 , b( ipivot ) , 1 , 1 , ret )
          call pvmfpack( integer4 , mtfpivo( i ) , 1 , 1 , ret )
          call pvmfpack( integer4 , mrecpivo( i ) , 1 , 1 , ret )
          for j = 1 until qtdlin do
              call pvmfpack( real8 , a( jpivot , 1 ) , tam , nla , ret )
              call pvmfpack( real8 , b( jpivot ) , 1 , 1 , ret )
              call pvmfpack( real8 , d( jpivot ) , 1 , 1 , ret )
              call pvmfpack( integer4 , j , 1 , 1 , ret )
              call pvmfpack( integer4 , i , 1 , 1 , ret )
              call pvmfpack( integer4 , uso , 1 , 1 , ret )
              k = k + 1
              jpivot = npiv( k )
          endfor
          call pvmfsend( wtids( i ) , msgmt , ret )
      endfor
      true = 1
      linha = 0
      while ( true = 1 ) do
          linha = linha + 1
          ind = 0
          for k = 1 until nprocs do
            if ( mrecpivo( k ) ≠ 0 ) then do
                ind = ind + 1
                call pvmfrecv( wtids( k ) , msgwk , ret )
                call pvmfunpack( real8 , coluna( ind ) , 1 , 1 , ret )
                call pvmfunpack( real8 , dd( ind ) , 1 , 1 , ret )
                call pvmfunpack( integer4 , pos( ind ) , 1 , 1 , ret )
                call pvmfunpack( integer4,qualtf(ind),1,1, ret)
            endif
          endfor
          colmax = 0.0D0
          for i = 1 until ind do
              hold = abs( coluna( i ) / dd(i) )
              if ( hold >  colmax) then do
                  colmax = hold
                  nrow = i
              endif
          endfor
c     Verifica a singularidade. A matriz é considerada singular se colmax
c     for equivalente ao zero
          if ( um + colmax ≠ um ) then do
              go to 73
          else
              go to 250
          endif
c     As variáveis achei(1) e achei(2) informam para as
c     Tarefas-Escravas em qual delas está a próxima
c     linha pivô (pivotamento parcial)
       73 continue
          achei( 1 ) = pos( nrow )
          achei( 2 ) = qualtf( nrow )
          for j = 1 until nprocs do
              call pvmfinitsend( pvmdefault , ret )
              call pvmfpack( integer4 , achei , 2 , 1 , ret )
              call pvmfsend( wtids( j ) , msgmt , ret )
          endfor
          for k = 1 until nprocs do
            if ( mrecpivo( k ) ≠ 0 ) then do
              call pvmfrecv( wtids( k ) , msgwk , ret )
              call pvmfunpack( integer4 , mtfpivo( k ) , 1 , 1,  ret )
              call pvmfunpack( integer4, mrecpivo( k ) , 1 , 1 , ret )
                if ( achei( 2 ) = k ) then do
                  call pvmfunpack( integer4 , mcol , 1 , 1 , ret )
                  call pvmfunpack( integer4 , mtam , 1 , 1 , ret )
                  call pvmfunpack( real8 , mlinhapivo , mtam , 1 , ret )
                  call pvmfunpack( real8 , mb , 1 , 1 , ret )
                  call pvmfunpack(real8,a(linha,mcol -1),mtam+1,nla,ret )
                  call pvmfunpack( real8 , b( linha ) , 1 , 1 , ret )
                endif
            endif
          endfor
c         é a última linha recebida da Tarefa-Mestra
          if ( linha  = ( nla – 1 ) ) then do
              a( nla , nla ) = mlinhapivo( 1 )
              b( nla ) = mb
c             mata os processos escravos
              for i = 1 until nprocs do
                  call pvmfkill( wtids( i ) , ret )
              endfor
              go to 120
          endif
          for m = 1 until nprocs do
              if ( mrecpivo( m ) ≠ 0 ) then do
                  call pvmfinitsend( pvmdefault , ret )
                  call pvmfpack( integer4 , mtfpivo( m ) , 1 , 1 , ret )
                  call pvmfpack( integer4 , mcol , 1 , 1 , ret )
                  call pvmfpack( integer4 , mtam , 1 , 1 , ret )
                  call pvmfpack( real8 , mlinhapivo , mtam , 1, ret )
                  call pvmfpack( real8 , mb , 1 , 1 , ret )
                  call pvmfsend( wtids( m ) , msgmt , ret )
              endif
          endfor
      endwhile
  120 continue
c     inicio da substituição regressiva.
      x( nla ) = b( nla ) / a( nla , nla )
      for i = nla – 1 until 1 passo – 1
          sum = b( i )
          for j = i + 1 until nla do
              sum = sum – a( i , j ) * x( j )
          endfor
          x( i ) = sum / a( i , i )
      endfor
      print *,' solução do sistema '
      for i = 1 until nla do
          print *,' x ( ' , i , ' ) = ',x( i )
      endfor
c     cálculo do vetor de resíduos
      for i = 1 until nla do
          v( i ) = 0.0D0
      endfor
      for j = 1 until nla do
          for i = 1 until nla do
              v( i ) = v( i ) + aa( i , j ) * x( j )
          endfor
      endfor
c
      for i = 1 until nla do
          resid( i ) = abs( bb( i ) – v( i ) )
      endfor
      indic = isamax( nla , resid , 1 )
      print *,' residuo = ', resid( indic )
      go to 260
  250 continue
      print *,' matriz considerada singular '
  260 continue
      call pvmfexit( ret )
      stop
      end
```

Obs.: ISAMAX() é uma rotina do BLAS.

* - linha acrescentada ao código da Eliminação de Gauss com piviteamento

Algoritmo B10 - Eliminação de Gauss - Algoritmo para a Tarefa-Escrava da Eliminação de Gauss com pivoteamento e escalamento.

```
c   nl      = tamanho de uma linha da matriz A
c   mitid   = identificador da Tarefa-Escrava
c   tidmt   = identificador da Tarefa-Mestra
c   mtot    = número de linhas a serem alocadas na memória do processo
c             escravo
c   codret  = código de retorno de algumas rotinas do PVM
    call pvmfmytid( mitid )
    msgmt = 1
    msgwk = 2
    call pvmfparent( tidmt )
    print 'identidade da tarefa pai  = ', tidmt
    print 'identidade  da Tarefa-Escrava = ', mitid
    call pvmfrecv( tidmt , msgmt , codret )
    call pvmfunpack( integer4 , wtarefa , 1 , 1 , ret )
    call pvmfunpack( integer4 , wcol , 1 , 1 , codret )
    call pvmfunpack( integer4 , wtam , 1 , 1 , codret )
    call pvmfunpack( integer4 , wqtdlin , 1 , 1 , codret )
    call pvmfunpack( real8 , privet , wtam , 1 , codret )
    call pvmfunpack( real8 , wb1 , 1 , 1 , codret )
    call pvmfunpack( integer4 , wtfpivo , 1 , 1 , codret )
    call pvmfunpack( integer4 , wrecpivo , 1 , 1 , codret )
    for  i = 1 until wqtdlin do
       call pvmfunpack( real8 ,waa( i,1),wtam,ntot,codret)
       call pvmfunpack( real8 , wbb( i ) , 1 , 1 , codret )
       call pvmfunpack( real8 , wd( i ) , 1 , 1 , codret )      *
       call pvmfunpack( integer4 , wpos( i ) , 1 , 1 , ret )
       call pvmfunpack( integer4 , wtaf( i ) , 1 , 1 , codret )
       call pvmfunpack( integer4 , wuso( i ) , 1 , 1 , codret )
    endfor
    true = 1
    while ( true  = 1 ) do
       wx = 0
       for i = 1 until wqtdlin do
        if ( wuso( i )  ≠  0 ) then do
          waa( i , wcol ) = waa( i , wcol ) / privet( wcol )
           for  j = wcol + 1 until nl do
                waa( i , j ) = waa(i , j) – ( waa(i , wcol ) × privet( j) )
           endfor
          wbb( i ) = wbb( i ) – waa( i , wcol ) × wb1
          wx = wx + 1
         endif
        endfor
        tcol = wcol + 1
        tconta = 0
        for i = 1 until  wqtdlin do
          if( wuso( i ) ≠  0 ) then do
              tconta = tconta + 1
              tcoluna( tconta ) = waa( i , tcol )
                td( tconta ) = wd( i )                    *
              tpos( tconta ) = wpos( i )
              ttaf( tconta ) = wtaf( i )
           endif
         endfor
c
         colmax = 0.0D0
         for  i = 1 until  tconta do
             hold = abs(  tcoluna( i ) / td(i)  )         *
             if(  hold  >  colmax ) then do
                  colmax = hold
                  nrow = i
             endid
         endfor
         call pvmfinitsend( pvmdefault , codret )
         call pvmfpack( real8 , tcoluna( nrow ) , 1 , 1 , codret )
         call pvmfpack( real8 , td( nrow ) , 1 , 1 , codret )      *
         call pvmfpack( integer4 , tpos( nrow ) , 1 , 1 , codret )
         call pvmfpack( integer4 , ttaf( nrow ) , 1 , 1 , ret )
         call pvmfsend( tidmt , msgwk , codret )
c
         call pvmfrecv( tidmt , msgmt , codret )
         call pvmfunpack( integer4 , wachei , 2 , 1 , codret )
         if( (  wachei( 2 ) = wtarefa ) .and. ( wx = 1 )    ) then do
             wrecpivo = 0
         endif
c
         call pvmfinitsend( pvmdefault , codret )
         call pvmfpack( integer4 , wtfpivo , 1 , 1 , codret )
         call pvmfpack( integer4 , wrecpivo , 1 , 1 , codret )
         if( wtarefa = wachei( 2 ) ) then do
             wtam = wtam – 1
             wcol = wcol + 1
             call pvmfpack( integer4 , wcol , 1 , 1 , ret )
             call pvmfpack( integer4 , wtam , 1 , 1 , ret )
             call pvmfpack(real8,waa(wachei(1) ,wcol) ,wtam ,ntot,ret)
             call pvmfpack( real8 , wbb( wachei( 1 ) ) , 1 , 1 , ret )
             call pvmfpack(real8,privet(wcol–1),wtam–1,1,codret )
             call pvmfpack( real8 , wb1 , 1 , 1 , codret )
             wuso(  wachei( 1 )  ) = 0
         endif
         call pvmfsend( tidmt , msgwk , codret )
c
         if( wrecpivo = 0 )   go to 80
c
         call pvmfrecv( tidmt , msgmt , codret )
         call pvmfunpack( integer4 , wtfpivo , 1 , 1 , codret )
         call pvmfunpack( integer4 , wcol , 1 , 1 , codret )
         call pvmfunpack( integer4 , wtam , 1 , 1 , codret )
         call pvmfunpack( real8 , privet( wcol ) , wtam , 1 , codret )
         call pvmfunpack( real8 , wb1 , 1 , 1 , codret )
       endwhile
 80    continue
       call pvmfexit( codret )
       stop
       end
```

* - Linha de código acrescentada ao código da Tarefa-Escrava da Eliminação de Gauss com pivoteamento.