**Algoritmos Paralelos para Solução
de Sistemas Lineares**

Henrique Mongelli

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Routo Terada**

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da CAPES

- São Paulo, abril de 1995 -

Algoritmos Paralelos para Solução de Sistemas Lineares

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Henrique Mongelli
e aprovada pela comissão julgadora.

São Paulo, 01 de junho de 1995.

Banca examinadora:

- Prof. Dr. Routo Terada (orientador) - IME-USP
- Prof. Dr. Siang Wun Song - IME-USP
- Prof. Dr. Jairo Panetta - IEA-CTA

*à minha esposa Magda
e ao meu filho que está a caminho*

Resumo

A solução de Sistemas Lineares é muito importante em diversas áreas científicas. Com o desenvolvimento da computação paralela na última década, este problema tem sido amplamente estudado pelo seu paralelismo inerente. Neste trabalho, apresentamos alguns conceitos básicos de paralelismo e de sistemas lineares nos dois primeiros capítulos. A seguir, descrevemos e discutimos alguns algoritmos: fatoração LU (Gauss, Gauss com pivotação, Gauss-jordan), fatoração QR (rotações de Givens e decomposição de Cholesky) e algoritmo de Csaniky.

Implementamos estes algoritmos em linguagem C e através de contagem do número de operações elementares (adição, subtração, multiplicação, divisão e comparação) obtivemos os números de operações máximo e médio por processador. De posse destes dados comparamos os desempenhos destes algoritmos. Além disso, obtivemos a carga de utilização dos processadores e verificamos o erro numérico na execução dos mesmos.

Abstract

Solving linear equation systems is very important in several scientific areas. With parallel computing development, this problem has been largely researched because of its inherent parallelism. In this work, we present some basic parallelism and linear systems concepts, in the first two chapters. In the three following chapters, we describe and discuss some parallel linear system solvers: LU factorization (Gauss, Gauss with pivotation, Gauss-Jordan), QR factorization (Givens rotations and Cholesky's decomposition) and Csaniky's algorithm. The last one presents a very important theoretical result.

These algorithms had been implemented in C language and by counting elementary operations (addition, subtraction, multiplication, division and comparison) we got the maximum and average numbers of operations per processor and we compared their performances. Besides, we got the load factor of processors and verified the rounding errors in these algorithms.

Agradecimentos

Ao meu orientador, Prof. Routo Terada, pela dedicação e paciência.

Aos meus pais, Orlando e Alzira, e meus irmãos pelo carinho e apoio.

À Profa. Yoshiko Wakabayashi pelo incentivo.

À Luiza Furlani pela amizade.

A Fábio Martinez, Flávio Miyazawa e Marcelo Lauretto pela ajuda e amizade.

Aos colegas do Departamento de Ciências Exatas/CEUC/UFMS.

A todos que contribuíram, direta ou indiretamente, no desenvolvimento deste trabalho.

Conteúdo

1	Introdução e Conceitos Básicos	1
1.1	Introdução	1
1.2	A necessidade de Computação Paralela.	3
1.3	Modelos de Computação	4
1.4	Modelo Memória Compartilhada	5
1.5	Modelo Combinacional ou Circuito Booleano	6
1.6	Modelo de Rede	8
1.6.1	Lista Linear e Anel	9
1.6.2	Grade (<i>Mesh</i>)	9
1.6.3	Hipercubo (n -cubo binário)	9
1.7	Complexidade	11
1.8	Medidas de Desempenho	11
1.9	Notação dos Algoritmos	13
1.10	Algoritmo <i>Fan-in</i> Associativo	15
1.11	Conclusão	18
2	Sistemas Lineares	19
2.1	Introdução	19
2.2	Matrizes	19
2.2.1	Conceitos Gerais	20
2.2.2	Operações com Matrizes	21
2.2.3	Matrizes Inversíveis	21

2.2.4	Operações Elementares	22
2.3	Sistemas Lineares	23
2.4	Sistemas Triangulares	24
2.4.1	Substituições Recorrentes	24
2.4.2	Algoritmo de Sameh e Brent	25
2.5	Algoritmos Básicos com Matrizes.	31
2.5.1	Produto Interno	32
2.5.2	Produto de Matriz por Vetor.	32
2.5.3	Produto de Matrizes	33
2.5.4	Potências de Matrizes	33
2.6	Conclusão	34
3	Fatoração LU	35
3.1	Introdução	35
3.2	A Fatoração LU	35
3.3	As Matrizes L e U	36
3.4	O Algoritmo da Eliminação de Gauss	38
3.5	Eliminação Gaussiana com Pivotação	40
3.6	Algoritmo de Gauss-Jordan	41
3.7	Fatoração LU em Outros Modelos	42
3.8	Implementação	43
3.8.1	Algoritmo de Gauss	44
3.8.2	Algoritmo de Gauss com pivotação	46
3.8.3	Algoritmo de Gauss-Jordan	48
3.9	Implementação na PVM	49
3.9.1	Algoritmo de Gauss	49
3.9.2	Algoritmo de Gauss com Pivotação	53
3.9.3	Algoritmo de Gauss-Jordan	57
3.9.4	Resultados	59
3.10	Conclusão	60

4	Fatoração QR	61
4.1	Introdução	61
4.2	Matrizes Ortogonais e Rotações	61
4.3	Rotações de Givens	62
4.4	Esquema de Eliminação de Sameh e Kuck	64
4.5	Outros Esquemas de Eliminação	67
4.6	Implementação do Algoritmo QR -Givens	68
4.7	Decomposição de Cholesky	71
4.8	Fatoração QR Usando Decomposição de Cholesky	74
4.9	Implementação do Algoritmo QR -Cholesky	76
4.10	Conclusão	80
5	Algoritmo de Csanky	81
5.1	Introdução	81
5.2	Autovalores e Autovetores	81
5.3	Método de Leverrier	83
5.4	Algoritmo de Csanky	84
5.5	Implementação	86
5.6	Conclusão	97
6	Resultados	99
6.1	Introdução	99
6.2	Número de Operações	100
6.2.1	Total de Operações - Máximo	100
6.2.2	Total de Operações - Média	102
6.2.3	Adições - Máximo	102
6.2.4	Adições - Média	104
6.2.5	Subtrações - Máximo	104
6.2.6	Subtrações - Média	105
6.2.7	Multiplicações - Máximo	105

6.2.8	Multiplicações - Média	107
6.2.9	Divisão - Máximo	107
6.2.10	Divisão - Média	108
6.2.11	Raízes Quadradas - Máximo	108
6.2.12	Raízes Quadradas - Média	110
6.2.13	Comparações - Máximo	110
6.2.14	Comparações - Média	110
6.3	Utilização dos Processadores	112
6.3.1	Porcentagem de Processadores com Carga Nula	112
6.3.2	Porcentagem de Processadores abaixo da Média	116
6.4	Erros	121
6.5	Ganho	122
6.6	Conclusão	124
7	Conclusões	127
A	PVM	131
A.1	Conceitos Básicos	131
A.2	Biblioteca de Comandos	132
B	Sistemas Lineares para Verificação dos Erros	135
C	Tabelas	137

Capítulo 1

Introdução e Conceitos Básicos

1.1 Introdução

Algoritmos paralelos para resolver problemas de Álgebra Linear foram bastante estudados durante as duas últimas décadas contribuindo para o aperfeiçoamento e desenvolvimento dos computadores paralelos. Entre estes algoritmos, os algoritmos para a solução de sistemas lineares foram amplamente pesquisados pela sua utilidade em diversas áreas científicas.

Neste trabalho, estudamos alguns algoritmos paralelos para a solução de sistemas lineares densos. Nosso objetivo principal é comparar seus desempenhos através de contagem do número de operações elementares por processador. Para isto utilizamos variáveis inteiras multidimensionais para contagem do número de operações. Cada elemento destas variáveis corresponde à contagem de operações em um determinado processador. Implementando os algoritmos em linguagem C e obtivemos alguns resultados: número máximo de operações, número médio de operações por processador, porcentagem de processadores com carga zero, porcentagem de processadores com carga abaixo da média e os erros na solução, que serviram para esta comparação. Não computamos o tempo de leitura dos dados e saída dos resultados, pois, na análise teórica, consideramos os dados inicialmente na memória. Utilizamos um modelo de memória compartilhada CREW. Não abordamos as estratégias de processamento distribuído [41] e sistólico [37]. Nas listagens das implementações transcritas para este trabalho marcamos com um asterisco, à esquerda, os pontos do programa onde foram feitas contagens de operações ou foi cronometrado o tempo.

No primeiro capítulo apresentamos a importância dos computadores paralelos e um breve histórico do seu desenvolvimento. Apresentamos os modelos de computação paralela mais utilizados. Definimos medidas de tempo teórico e as classes *NC* e *AC*. Descrevemos a sintaxe utilizada nos algoritmos deste texto. Mostramos um algoritmo para

a determinação de um valor através de operações binárias associativas que nos fornece um resultado teórico muito importante, pois apresenta uma cota inferior para este tipo de operação (Teorema 1). Neste texto, ao nos referirmos a algoritmos estaremos considerando algoritmos paralelos. Quando tratarmos de algoritmos sequenciais isto será explicitado.

O segundo capítulo traz alguns conceitos básicos de matrizes e sistemas lineares. São mostrados dois algoritmos para a solução de sistemas triangulares e apresentamos alguns algoritmos para as operações com matrizes mais utilizadas neste trabalho. Consideramos sistemas lineares cujas matrizes dos coeficientes fossem quadradas e inversíveis, isto é, consideramos sistemas compatíveis determinados.

No terceiro capítulo, apresentamos o algoritmo da eliminação de Gauss, que é uma versão menos formal do algoritmo da fatoração LU e duas variantes deste algoritmo: o algoritmo de Gauss com pivotação e o algoritmo de Gauss-Jordan, mostrando seus aspectos teóricos. Fazemos uma breve discussão do algoritmo de Gauss com pivotação em um modelo de memória distribuída. São mostradas as implementações através das quais obtivemos os números de operações. Estes algoritmos foram implementados na PVM (*Parallel Virtual Machine*), que teve um caráter apenas ilustrativo.

Os algoritmos que utilizam a decomposição QR são discutidos no capítulo 4. Estudamos duas formas de obtermos esta fatoração: através de rotações de Givens e através da decomposição de Cholesky. Fizemos uma apresentação teórica destes algoritmos e, logo a seguir a cada uma delas, foram mostradas suas implementações.

O algoritmo paralelo de Csanky para a inversão de matrizes, que traz um importante resultado teórico - a inclusão deste problema na classe NC^2 - é apresentado no capítulo 5. Apesar de ser o melhor algoritmo conhecido para este problema, ele é numericamente instável e utiliza um número elevado de processadores. Descrevemos este algoritmo teoricamente e apresentamos sua implementação, que é a mais complexa entre os algoritmos estudados.

No capítulo 6, apresentamos os resultados obtidos na execução das implementações. Para cada algoritmo determinamos o número total de operações máximo e médio por processador, que estão mostrados em tabelas e gráficos. Estes valores também foram determinados para cada operação elementar (adição, subtração, multiplicação, divisão e comparação) e para as raízes quadradas que aparecem nos algoritmos da fatoração QR e que não entraram no cômputo do número total de operações. As porcentagens de processadores com carga nula e com carga abaixo da média estão apresentados em tabelas. Determinamos os erros para estes algoritmos através de sistemas que são utilizados como padrão para testes com sistemas lineares, e estão apresentados no apêndice B. Estes erros são mostrados em tabelas.

As implementações dos algoritmos estão disponíveis no endereço @brucutu.ime.usp.br no diretório /home/posmac/mongelli/pub.

1.2 A necessidade de Computação Paralela.

Os computadores paralelos começaram a surgir com grande força na década de 80 pelas necessidades das aplicações, utilizando o computador. Além disso, o aumento da velocidade, diminuição do tamanho e o barateamento dos microprocessadores contribuiu para o desenvolvimento das máquinas paralelas.

Mas se conseguimos microprocessadores mais rápidos, mais baratos e menores, por que utilizarmos computadores paralelos? A resposta para isto está nas leis físicas que limitam a velocidade de comunicação entre os componentes dos computadores, fazendo com que todo o ganho obtido no processamento se perca nas comunicações entre eles. A aproximação dos componentes poderia parecer uma solução; porém, existe um limite a partir do qual estes acabam interferindo entre si.

A idéia de computação paralela não é recente. Quando o primeiro computador digital foi criado, já se sabia do ganho obtido com a divisão de tarefas. Na década de 70, os supercomputadores já adotavam o *pipeline* em sua arquitetura, alguns adotavam componentes especializados que liberavam o processador para outras tarefas. Estes exemplos não caracterizavam computadores paralelos, apesar de neles haver ganho de tempo.

O conceito de computador paralelo dado em Já.Já [26] está descrita a seguir:

Um computador paralelo é simplesmente uma coleção de processadores, tipicamente do mesmo tipo, interconectados de uma certa maneira a permitir a coordenação de suas atividades e a troca de dados.

A velocidade de computadores paralelos é medida em *flops* (*floating point operations per second* - operações de ponto flutuante por segundo). Os mais potentes computadores paralelos atualmente atingem um pico¹ de $2\ Tflops = (10^{12}\ flops)^2$.

O uso de computadores paralelos faz-se necessário em diversos problemas onde o volume de dados e cálculos é grande e precisa-se de rapidez na obtenção das respostas. Tais problemas podem ser encontrados na previsão do tempo, na prospecção de petróleo, na dinâmica dos fluídos, na computação gráfica, entre outros.

Com a computação paralela a criação de algoritmos modifica-se sensivelmente. As estratégias utilizadas em algoritmos sequenciais para resolver um problema não servem totalmente para a criação de algoritmos paralelos para o mesmo problema. Os recursos disponíveis são maiores e deve-se aproveitá-los ao máximo. Na criação dos algoritmos leva-se em conta também a natureza dos problemas. Alguns são inerentemente paralelizáveis; outros apresentam um paralelismo não tão transparente, exigindo estratégias mais elaboradas na montagem dos algoritmos; e existem aqueles em que o grau de paralelismo

¹A velocidade de pico é a velocidade obtida se todos os processadores estivessem trabalhando no máximo de sua capacidade. É obtida pela soma das velocidades de pico de cada um deles.

²CM-5 (*Connection Machine*) com 16000 processadores.

é muito baixo ou inexistente. Além disso, no desenvolvimento dos algoritmos temos que considerar a arquitetura paralela a ser utilizada, pois isto afeta o desempenho dos algoritmos.

1.3 Modelos de Computação

Qualquer computador sequencial ou paralelo executa uma sequência de instruções sobre um conjunto de dados. Uma sequência de instruções diz ao computador o que fazer em cada passo. O conjunto de dados é então utilizado e modificado por estas instruções. Baseado nesta idéia as máquinas podem ser agrupadas segundo a classificação de Flynn [18].

- **SISD** (Single Instruction stream, Single Data stream)
- **MISD** (Multiple Instruction stream, Single Data stream)
- **SIMD** (Single Instruction stream, Multiple Data stream)
- **MIMD** (Multiple Instruction stream, Multiple Data stream)

O modelo SISD mostra o esquema de funcionamento de uma máquina sequencial onde uma sequência única de dados é processada por uma sequência única de instruções.

O modelo MISD é inexistente na prática.

Os modelos SIMD e MIMD representam computadores paralelos. Em ambos, a sequência de dados é múltipla, sendo que no primeiro a sequência de instruções é única e no segundo é múltipla.

Nos modelos em paralelo podemos ter dois tipos de memória para armazenamento dos dados.

- memória compartilhada (*shared memory*)
- memória não-compartilhada (*non-shared memory*)

A memória compartilhada não necessita de sincronização global porque existe uma maior transparência para o programador simular qualquer modelo de memória não compartilhada e a comunicação entre os processadores é feita através de leitura e escrita na memória compartilhada. Com este tipo de memória é possível construir um modelo de máquina paralela fácil de utilizar.

Na memória não-compartilhada a comunicação entre os processadores é feita através de troca de mensagens, a sincronização somente é necessária no tráfego dos dados e tem-se a vantagem de se explorar a topologia, para solucionar uma classe de problemas de maneira eficiente.

Figura 1.1: Memória compartilhada com p processadores.

Dessa forma, podemos ter os seguintes modelos em paralelo:

- modelo grafo orientado acíclico
serve para representar certas computações paralelas de modo simples e não está relacionado com arquiteturas particulares.
- modelo de rede
incorpora a topologia de interconexão e captura as características da comunicação
- modelo de memória compartilhada
modelo na qual um número de processadores acessa uma memória global

Nas próximas sessões trataremos mais detalhadamente dos modelos descritos acima.

1.4 Modelo Memória Compartilhada

Definição 1 O modelo memória compartilhada é um modelo mais simples e geral para se estudar computação paralela. Consiste num conjunto de processadores idênticos para propósitos gerais P_i , $1 \leq i \leq p$, cada um com sua memória local. O modelo de memória compartilhada SIMD é também conhecido como **PRAM** (Parallel Random Access Machine).

A memória local de cada processador armazena os programas e dados intermediários para seus cálculos. Qualquer comunicação entre processadores é feita através de leitura e escrita na memória global. Cada um dos processadores conhece seu índice.

Vamos supor que os dados estarão armazenados nas n primeiras posições da memória compartilhada. O número de processadores deve ser uma função do tamanho do problema.

Em um dado passo, cada processador pode ler ou escrever na memória compartilhada ou efetuar algum cálculo aritmético ou lógico. Dependendo do modo como os dados serão lidos ou escritos na memória compartilhada, temos três possibilidades de modelos a partir do mais forte.

- **CRCW** (*Concurrent Read, Concurrent Write*)

É permitida simultaneidade na leitura e escrita no mesmo espaço de memória por diferentes processadores. Para administrar os possíveis conflitos temos os critérios:

modelo *CRCW* com escrita *prioritária*

o processador com índice menor tem prioridade para escrever.

modelo *CRCW* com escrita *comum*

é necessário que todos os processadores em conflito escrevam o mesmo valor.

modelo *CRCW* com escrita *arbitrária*

um dos processadores deve ter sucesso na escrita, não importando qual deles.

- **CREW** (*Concurrent Read, Exclusive Write*)

É permitida leitura simultânea de uma mesma célula de memória por mais de um processador. Não é permitida a escrita simultânea em uma mesma célula por mais de um processador.

- **EREW** (*Exclusive Read, Exclusive Write*)

Não é permitido leitura ou escrita simultâneas em uma mesma posição de memória.

Existem equivalências em complexidade entre alguns dos modelos acima, sendo possível simular em um deles o comportamento do outro[28, 44].

1.5 Modelo Combinacional ou Circuito Booleano

Este modelo tem grande utilidade na obtenção de resultados teóricos quanto à complexidade de algoritmos paralelos. Com ele podemos montar classes de complexidade e correlacionar estas classes [28, 44].

Definição 2 Um circuito booleano B com n entradas e m saídas é um grafo finito orientado acíclico com nós rotulados segundo a regra: o circuito B possui k nós de entrada com grau de entrada zero com rótulos $x_1, x_2, \dots, x_k$; todos os outros nós com grau de entrada zero são rotulados com as constantes 0 ou 1. Os nós de operação possuem grau de entrada 1 ou 2. Os nós de operação com rótulo **NÃO** têm grau de entrada um e fazem a operação **NOT**. Os nós de operação de grau de entrada dois são rotulados **E** ou **OU** e fazem as operações **AND** ou **OR**, respectivamente. Não há qualquer restrição

quanto ao grau de saída destes nós. Exatamente m nós rotulados $y_1, y_2, \dots, y_m$ são nós de saída. Todo nó de entrada possui um caminho dele até algum nó de saída.

Definição 3 O tamanho s_n de um circuito booleano é o número de arestas do grafo. A profundidade d_n de um circuito booleano é o maior caminho de um nó de entrada a um nó de saída.

O tamanho do circuito está relacionado com o conteúdo do *hardware* e a profundidade como o tempo de processamento.

Definição 4 Uma família de circuitos é **log-espaco uniforme** se dado um inteiro n é possível gerar um n -ésimo circuito em espaco logarítmico³ usando um algoritmo determinístico seqüencial.

A partir dessas definições podemos definir algumas classes de algoritmos e desta forma classificar alguns problemas, levando-se em conta a profundidade dos circuitos booleanos que os resolvem.

Definição 5 A classe NC^k para $k \geq 0$ é a coleção dos problemas cujas soluções podem ser determinadas por circuitos log-espaco uniformes com tamanho polinomial em n e profundidade $\log^k n$.

Definição 6 A classe NC é a união $NC = \bigcup_{k \geq 0} NC^k$.

Se na definição das classes NC e NC^k pudermos ter nós de operação com rótulos **E** ou **OU** com grau de entrada de tamanho qualquer, então teremos as classes AC e AC^k ($k \geq 0$) definidas de forma análoga.

É possível, a partir de um circuito AC^k , gerarmos um circuito NC^{k+1} . Mais ainda, $AC = NC$. Podemos também mostrar equivalências, por exemplo, entre as classes AC^k e NC^k e os modelos de memória compartilhada $CRCW^k$, onde $CRCW^k$ é a família dos problemas solucionáveis por modelos $CRCW$ gerados em espaco logarítmico uniforme, e de complexidade de espaco polinomial e tempo $\mathcal{O}(\log^k n)$ [28, 44].

Exemplos de problemas em NC^1 .

- soma ou multiplicação de dois números de n -bits.
- soma de n números de n -bits.

³O logaritmo utilizado em todo este texto é o logaritmo de base 2. Usaremos a notação $\log$ para representá-lo.

- multiplicação de matrizes de inteiros ou booleanas.
- ordenação de n números de n -bits.

Exemplos de problemas em NC^2 .

- divisão de dois números, cada um com n -bits.
- computação sobre matrizes e potência A^n .
- inversão de matrizes.
- computação de determinante.
- ordenação topológica de grafo acíclico.
- custo mínimo para multiplicar sequência de matrizes.

Podemos considerar os nós de operações rotulados com operações aritméticas binárias obtendo um circuito de operadores aritméticos. Este tipo de circuito pode ser utilizado como modelo de computação paralela e é muito útil na determinação de cotas inferiores da complexidade de algoritmos paralelos, que utilizam operações aritméticas, como veremos na seção 1.10.

1.6 Modelo de Rede

O modelo de rede está associado a uma arquitetura de interconexão de processadores. Não trataremos detalhadamente das arquiteturas, apenas descreveremos brevemente as mais utilizadas.

Uma rede é considerada como um grafo $G(V, A)$, onde V é o conjunto de vértices do grafo G e A é o conjunto de arestas de G . Cada $i \in V$ representa um processador e cada aresta $(i, j) \in A$ representa um canal de comunicação (bidirecional).

Cada processador possui sua memória local, não existe memória compartilhada.

O desempenho destes modelos é dado em função dos parâmetros diâmetro (máxima distância entre qualquer par de vértices) e grau máximo, que devem ser pequenos.

Figura 1.2: Modelo de lista linear com p processadores Figura 1.3: Modelo de anel com p processadores

1.6.1 Lista Linear e Anel

Lista Linear

Consiste de uma sequência de p processadores $P_1, P_2, \dots, P_p$ conectados linearmente, isto é, P_i está conectado a P_{i-1} e P_{i+1} , se existirem. Apresenta diâmetro $p - 1$.

Anel

É uma lista linear onde o último processador está conectado ao primeiro. Seu diâmetro é $\lfloor \frac{p}{2} \rfloor$.

1.6.2 Grade (*Mesh*)

É uma versão bidimensional da lista linear. Consiste de $p = n^2$ processadores conectados numa grade $n \times n$ de tal forma que um processador P_{ij} está conectado aos processadores $P_{i\pm 1j}$ e $P_{ij\pm 1}$, se existirem. Tem diâmetro $2\sqrt{p} - 2$.

1.6.3 Hipercubo (n -cubo binário)

Um hipercubo de dimensão d consiste de $p = 2^d$ processadores conectados como descrito a seguir.

Figura 1.4: Modelo de grade com n^2 processadores

Figura 1.5: Modelo de hipercubo com a representação binária da identificação de seus processadores.

Consideremos os p processadores rotulados como $P_0, P_1, \dots, P_{p-1}$.

Seja a representação binária de i , $0 \leq i \leq p-1$ dada por

$$i = i_{d-1}i_{d-2} \dots i_2i_1i_0$$

Seja $i^{(j)} = i_{d-1}i_{d-2} \dots \bar{i}_j \dots i_2i_1i_0$ onde $\bar{i}_j = 1 - i_j$.

O processador P_i está ligado ao processador $P_{i^{(j)}}$, $0 \leq j \leq d-1$, isto é, dois processadores estão ligados se e somente se seus índices diferem em exatamente um bit.

Um hipercubo tem grau máximo $d(= \log p)$ e diâmetro $d(= \log p)$.

Os modelos grade e hipercubo são os que aparecem com mais frequência na arquitetura de máquinas existentes e em desenvolvimento. Existem outros modelos que são menos utilizados como: *shuffle exchange*, *butterfly* e grafo de Bruijn.

1.7 Complexidade

A complexidade de algoritmos paralelos pode ser determinada como nos algoritmos sequenciais, através do crescimento assintótico do número de operações elementares efetuadas em cada processador. Consideraremos como operações elementares as operações aritméticas ou lógicas. Usamos as mesmas notações de algoritmos sequenciais para a determinação dos tempos teóricos, onde n é o tamanho da instância do problema.

- $T(n) = \mathcal{O}(f(n))$ se existem constantes inteiras positivas c, n_0 tal que $T(n) \leq c \cdot f(n), \forall n \geq n_0$.
- $T(n) = \Omega(f(n))$ se existem constantes inteiras positivas c, n_0 tal que $T(n) \geq c \cdot f(n), \forall n \geq n_0$.
- $T(n) = \Theta(f(n))$ se $T(n) = \mathcal{O}(f(n))$ e $T(n) = \Omega(f(n))$ ou, equivalentemente, se existem constantes inteiras positivas c_1, c_2 e n_0 tais que $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \forall n \geq n_0$.

1.8 Medidas de Desempenho

Existem várias formas de se medir o desempenho de computadores paralelos e algoritmos paralelos. Usualmente podemos utilizar o desempenho de pico das máquinas medido em

FLOPS (*floating point operations per second* - operações de ponto flutuante por segundo). Porém, na prática, estas medidas não são realísticas, uma vez que não utilizamos toda a capacidade dos processadores. Uma maneira de mostrar este desempenho é a utilização dos conceitos a seguir, que utilizam o tempo de um algoritmo seqüencial como padrão para comparação [2, 26, 28, 31].

Dado um problema $\mathcal{P}$ com entrada de tamanho n .

- **Ganho ou Speed-up ($S_p(n)$)**

O ganho compara a complexidade de um algoritmo paralelo usando p processadores ($T_p(n)$) com a complexidade do melhor algoritmo seqüencial que resolve $\mathcal{P}$ ($T(n)$). Talvez esta seja a melhor medida conhecida da eficácia de algoritmos paralelos com respeito ao melhor algoritmo seqüencial

$$S_p(n) = \frac{T(n)}{T_p(n)}.$$

$S_p(n)$ é uma função do tamanho n do problema e do número de processadores.

Na definição do ganho o algoritmo paralelo pode ser bem diferente do algoritmo seqüencial.

- **Eficiência do processador ($E_p(n)$)**

$$E_p(n) = \frac{T(n)}{pT_p(n)} = \frac{S_p(n)}{p}$$

$E_p(n)$ mede a utilização efetiva dos p processadores, $E_p(n) \leq 1$.

Quando $E_p(n) \approx 1$ todos os processadores são úteis.

Um algoritmo paralelo, usando $p(> 1)$ processadores, deve ser tão rápido quanto um algoritmo seqüencial, assim

$$S_p(n) \geq 1.$$

Algumas vezes, podemos estar interessados em comparar o desempenho do algoritmo paralelo numa máquina seqüencial com o mesmo algoritmo usando p processadores. O ganho é dado por

$$S_p^*(n) = \frac{T_1(n)}{T_p(n)}$$

onde $T_1(n)$ é a complexidade do algoritmo paralelo usando um processador.

A eficiência é então dada por

$$E_p^*(n) = \frac{S_p^*(n)}{p}.$$

Por um raciocínio análogo, $S_p^*(n) \geq 1$.

Temos que $S_p^*(n) \leq p$, uma vez que um passo do algoritmo paralelo usando p processadores leva no máximo p passos, quando implementado sequencialmente. Isto implica $T_1(n) \leq pT_p(n)$.

Assim,

$$1 \leq S_p^*(n) \leq p \text{ e } 0 \leq E_p^*(n) \leq 1.$$

Obviamente, $T(n) \leq T_1(n)$, pela escolha de $T(n)$, logo $S_p(n) \leq S_p^*(n)$. Se $S_p(n) = \Omega(p)$ o algoritmo é dito ser de ganho ótimo.

1.9 Notação dos Algoritmos

Os algoritmos serão escritos, usando uma descrição de alto nível e combinando comandos em português com estruturas de programação conhecidas.

As variáveis serão utilizadas como nas linguagens de programação. Daremos preferência a letras maiúsculas para variáveis indexadas e minúsculas para variáveis de controle. Não serão feitas declarações de variáveis.

Os blocos de comando serão destacados pela indentação utilizada, sem se fazer uso de delimitadores.

- Comando *Calcular*

O comando *Calcular* executa um procedimento (ou função) previamente conhecido ou descrito. Sua forma geral é dada por:

Calcular argumento,

onde argumento é uma atribuição ou simplesmente o nome da variável de atribuição (quando ficar claro a operação ou procedimento que está sendo efetuado).

Exemplos.

Calcular $C = AB$

Calcula o produto de A por B , colocando o resultado em C .

Calcular A^j

Calcula o valor de A elevado à j -ésima potência colocando o resultado em A^j .

- Laços de controle seqüenciais

Os laços de controle seqüenciais serão feitos pelo comando

para var=início até fim passo incr faça

bloco de comandos

onde

var é uma variável de controle

início é o valor inicial do laço

fim é o valor final do laço

incr é o incremento a ser utilizado

bloco de comandos é a seqüência de comandos a serem executados dentro do laço

O funcionamento do laço de controle seqüencial acima é o mesmo dos laços de controle de linguagens estruturadas como C ou Pascal.

Os comandos que serão executados em paralelo serão seguidos pela expressão *em paralelo* logo após sua descrição normal. Dessa forma, temos as seguintes descrições:

- Comando *Calcular em paralelo*

Sua forma geral é dada por:

Calcular argumento em paralelo

Este comando funciona basicamente da mesma forma que o respectivo seqüencial, com a diferença de que o argumento vai ser executado por um procedimento ou função paralelo previamente conhecido ou descrito.

- Laços de controle em paralelo

O laço de controle⁴ em paralelo é descrito por:

para controle faça em paralelo

bloco de comandos

onde

⁴Este comando corresponde ao comando *forall* da literatura.

controle é um intervalo descrito por:

$$\underline{\text{início}} \left(\begin{array}{c} \leq \\ \text{ou} \\ < \end{array} \right) \underline{\text{var}} \left(\begin{array}{c} \leq \\ \text{ou} \\ < \end{array} \right) \underline{\text{fim}}$$

ou

$$\underline{\text{var}} \in \{\underline{\text{início}}, \dots, \underline{\text{fim}}\}$$

var é uma variável de controle

início é o valor inicial do laço

fim é o valor final do laço

bloco de comandos é a seqüência de comandos a serem executados dentro do laço

Os comandos do bloco de comandos são executados em paralelo por processadores separados para cada valor descrito em controle.

1.10 Algoritmo *Fan-in* Associativo

O algoritmo descrito nesta seção é muito utilizado na determinação de valores a partir de operações binárias associativas. A idéia deste algoritmo oferece uma cota inferior para algoritmos que envolvam este cálculo [5, 23, 28].

Vamos descrever inicialmente este algoritmo, usando a adição e sua propriedade associativa. É baseado em um esquema de divisão e conquista. Dados n valores $x_1, x_2, \dots, x_n$ a adição destes n valores pode ser obtida pela expressão

$$\sum_{i=1}^n x_i = \sum_{i=1}^{\lceil n/2 \rceil} x_i + \sum_{i=\lceil n/2 \rceil+1}^n x_i \quad (1.1)$$

Podemos exprimir cada um dos termos do lado direito da equação 1.1 como a soma de $n/4$ termos, e assim sucessivamente. Este algoritmo, seqüencialmente, necessita de $(n-1)$ operações. Na figura 1.6 temos um grafo representando o funcionamento deste algoritmo para oito valores.

Consideremos a seguir o algoritmo *fan-in* associativo em paralelo para $n = 2^k$ e $k \geq 1$. O número de processadores é pelo menos $n/2$. O modelo utilizado é de memória compartilhada e os dados estão, inicialmente, armazenados em um vetor A, a resposta é colocada numa variável S.

Figura 1.6: Grafo de passos do algoritmo *fan-in* associativo para a soma de 8 valores. Os nós de um mesmo nível podem ser obtidos simultaneamente.

Algoritmo *fan-in*

Entrada: um vetor A com $n = 2^k$, $k \geq 0$, elementos.

Saída: S contendo a soma dos elementos deste vetor.

1. Para $j = 1$ até $\log n$ faça
 - Para $1 \leq i \leq \frac{n}{2^j}$ faça em paralelo
 - Calcular $A_i = A_{2i-1} + A_{2i}$
2. $S = A_1$

Este algoritmo leva $\mathcal{O}(\log n)$ passos usando no máximo $\lceil n/2 \rceil$ processadores.

Como dissemos anteriormente, a idéia do algoritmo *fan-in* associativo pode ser estendido a qualquer operação que seja associativa.

O teorema enunciado a seguir traz um resultado simples, mas de muita importância, obtido por Munro e Patterson [33].

Teorema 1 *Seja $N \geq 1$ o número de operações binárias necessárias para se determinar um valor único V numa máquina seqüencial. Se $t = T_p(N)$ é o menor tempo gasto para determinar-se V , utilizando uma máquina paralela com p processadores, então*

$$t \geq \left\lceil \frac{N + 1 - 2^{\lceil \log p \rceil}}{p} \right\rceil + \lceil \log p \rceil, \text{ se } N \geq 2^{\lceil \log p \rceil},$$

e

$$t \geq \lceil \log(N + 1) \rceil, \text{ caso contrário.}$$

Demonstração.

Na sequência de cálculos para a determinação de V , no máximo um processador está sendo utilizado no tempo t . Como cada operação utilizada no cálculo de V é uma operação binária, no tempo $t - 1$ estarão sendo utilizados no máximo dois processadores. Desta maneira, observa-se que para $j \geq 0$, no tempo $t - j$, no máximo $\min(p, 2^j)$ processadores estarão sendo utilizados para a determinação de V .

Usando p processadores, o número total de operações que podem ser feitas em tempo $\lceil \log p \rceil$ é dado por

$$\sum_{i=1}^{\lceil \log p \rceil} 2^{i-1} = 2^{\lceil \log p \rceil} - 1.$$

Temos dois casos a analisar.

Caso 1.

Se $N > 2^{\lceil \log p \rceil} - 1$, então $t > \lceil \log p \rceil$. Dessa forma, durante as primeiras $\lceil \log p \rceil$ unidades de tempo, o número de operações é dado por

$$1 + 2 + 2^2 + \dots + 2^{\lceil \log p \rceil - 1} = 2^{\lceil \log p \rceil} - 1.$$

Como o número máximo de operações realizadas num dado instante é limitado por p , um total de $p(t - \lceil \log p \rceil)$ operações são realizadas nas $t - \lceil \log p \rceil$ unidades de tempo restantes.

Assim,

$$N \leq p(t - \lceil \log p \rceil) + 2^{\lceil \log p \rceil} - 1.$$

Daí, segue-se que

$$t \geq \left\lceil \frac{N + 1 - 2^{\lceil \log p \rceil}}{p} \right\rceil + \lceil \log p \rceil.$$

Caso 2.

$$t \leq \lceil \log p \rceil \text{ ou } N \leq 2^{\lceil \log p \rceil} - 1.$$

Como o número de operações que podem ser realizadas em t unidades de tempo é

$$\sum_{i=0}^{t-1} 2^i = 2^t - 1, \text{ segue-se que}$$

$$N \leq 2^t - 1, \text{ donde}$$

$$t \geq \lceil \log(N + 1) \rceil.$$

□

Este teorema nos fornece uma classe de algoritmos ótimos para o cálculo de

$$x_1 \circ x_2 \circ \cdots \circ x_N$$

onde, $\circ$ é uma operação aritmética binária associativa. A classe é a dos algoritmos *fan-in* associativos. O teorema implica que qualquer cálculo usando p processadores que envolva adições, subtrações, multiplicações ou divisões produzindo um valor único V , utilizando $\Omega(N)$ operações numa máquina seqüencial necessita de $\Omega(\frac{N}{p} + \log p)$ passos usando p processadores. Este resultado é válido para operações aritméticas binárias pois Valliant [46] descreve um algoritmo para determinação do máximo de uma lista de N números usando somente comparações e p processadores, em um modelo CREW, em $\Theta(\log \log p)$ passos.

1.11 Conclusão

Neste capítulo mostramos a necessidade da computação paralela. Descrevemos os modelos de memória compartilhada, de rede e grafo orientado acíclico. Nos algoritmos deste trabalho será utilizado um modelo de memória compartilhada CREW. O modelo grafo orientado acíclico nos possibilita a determinação das classes de complexidade NC e AC .

Definimos as medidas de complexidade através do comportamento assintótico do número de operações elementares. Definimos ganho e eficiência que são as medidas de desempenho de algoritmos paralelos mais utilizadas. Descrevemos a sintaxe dos algoritmos a ser utilizada ao longo deste texto.

O algoritmo *fan-in* associativo foi descrito e determinamos sua complexidade de tempo e número de processadores. Mostramos que para a determinação de um valor V , utilizando somente operações binárias associativas, pode ser realizada em $\Omega(\frac{N}{p} + \log p)$ passos utilizando p processadores. Este é um resultado teórico muito importante pois oferece uma cota inferior para este tipo de problema.

Capítulo 2

Sistemas Lineares

2.1 Introdução

Sistemas lineares estão presentes em muitas aplicações práticas de diversas áreas como Engenharia, Física, Química, etc., na solução de equações diferenciais, problemas de otimização e análise de várias estruturas discretas. Devido ao grande volume de dados e cálculos envolvidos nos problemas práticos tem-se buscado algoritmos mais eficientes e estáveis.

Algoritmos paralelos para solução de problemas que envolvam cálculos com matrizes têm sido amplamente estudados, não só pela sua estrutura, presente em muitas outras aplicações, mas pelas contribuições que seu estudo vem trazendo para o desenvolvimento da computação paralela.

Os algoritmos paralelos para solução de sistemas lineares que, pela sua própria natureza, também envolvem cálculos com matrizes têm sido amplamente pesquisados. Algoritmos clássicos puderam ser adaptados para o processamento paralelo e outros algoritmos surgiram buscando mais eficiência na solução deste problema.

Neste capítulo, introduziremos alguns conceitos básicos de sistemas lineares e matrizes, algoritmos paralelos para a solução de sistemas triangulares e alguns algoritmos paralelos básicos envolvendo cálculos com matrizes.

2.2 Matrizes

Nesta seção, definiremos alguns conceitos e operações envolvendo matrizes.

2.2.1 Conceitos Gerais

Definição 7 Dados $m \geq 1$ e $n \geq 1$, números inteiros. Uma matriz $m \times n$ real é um arranjo de números reais

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}.$$

Diremos que esta é uma matriz de ordem $m \times n$.

Uma matriz costuma ser representada por letras latinas maiúsculas. Podemos representar uma matriz A em relação ao seu termo geral $a_{i,j}$

$$A = (a_{i,j}) = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}, \text{ com } a_{i,j} \in \mathbb{R}, 1 \leq i \leq m \text{ e } 1 \leq j \leq n \quad (2.1)$$

Definição 8 Uma matriz onde o número de linhas é igual ao número de colunas ($m = n$) é denominada matriz quadrada de ordem n (ou m).

O conjunto das matrizes de ordem $m \times n$ forma um espaço vetorial real, denominado espaço das matrizes de ordem $m \times n$, representado por $M_{m \times n}(\mathbb{R})$. No caso de matrizes quadradas, simplesmente dizemos espaço das matrizes de ordem n , representado por $M_n(\mathbb{R})$.

Definição 9 Considerando 2.1 as m seqüências horizontais

$$A^{(1)} = (a_{1,1} \ a_{1,2} \ \cdots \ a_{1,n}), \dots, A^{(m)} = (a_{m,1} \ a_{m,2} \ \cdots \ a_{m,n})$$

são denominadas linhas de A . Analogamente as n seqüências verticais

$$A_{(1)} = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{pmatrix}, \dots, A_{(n)} = \begin{pmatrix} a_{1,n} \\ a_{2,n} \\ \vdots \\ a_{m,n} \end{pmatrix}$$

são denominadas colunas de A .

2.2.2 Operações com Matrizes

Algumas operações básicas com matrizes são dadas a seguir.

- Adição de matrizes

Dadas A e B matrizes reais de ordem $m \times n$. A matriz C de ordem $m \times n$, que é o resultado da adição de A e B , é dada por

$$C = A + B \text{ onde } c_{i,j} = a_{i,j} + b_{i,j}, \text{ com } 1 \leq i \leq m \text{ e } 1 \leq j \leq n.$$

- Produto de matriz por escalar

Dados A uma matriz real de ordem $m \times n$ e $\alpha \in \mathbb{R}$. A matriz C de ordem $m \times n$ que é o produto da matriz A pelo escalar α é dada por

$$C = \alpha A \text{ onde } c_{i,j} = \alpha a_{i,j}, \text{ com } 1 \leq i \leq m \text{ e } 1 \leq j \leq n.$$

- Produto de matrizes

Dadas A e B matrizes reais de ordens $m \times n$ e $n \times p$, respectivamente. A matriz C de ordem $m \times p$, que é o resultado do produto de A e B , é dada por

$$C = AB \text{ onde } c_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j}, \text{ com } 1 \leq i \leq m \text{ e } 1 \leq j \leq p.$$

- Transposição

Dada uma matriz real A de ordem $m \times n$. A transposta C de A é uma matriz $n \times m$ dada por

$$C = A^t \text{ onde } c_{i,j} = a_{j,i}, \text{ com } 1 \leq i \leq n \text{ e } 1 \leq j \leq m.$$

2.2.3 Matrizes Inversíveis

Definição 10 Uma matriz quadrada de ordem n onde

$$a_{i,j} = \begin{cases} 0, & \text{se } i \neq j \\ 1, & \text{se } i = j \end{cases}$$

é denominada **matriz identidade de ordem n** , representada por I_n ou, simplesmente, I .

A matriz identidade é o elemento neutro do produto de matrizes (mantendo-se as respectivas correspondências nas dimensões das matrizes envolvidas).

Definição 11 Uma matriz A de ordem n se diz **invertível** se, e somente se, existe uma matriz B de ordem n , tal que $AB = BA = I_n$. A matriz B , caso exista, é única e é denominada **inversa** de A , indicada por A^{-1} .

O conjunto das matrizes quadradas de ordem n invertíveis é um subespaço vetorial de $M_n(\mathbb{R})$.

As matrizes invertíveis apresentam propriedades que são facilmente demonstradas [8, 24].

Propriedade 1 Dada uma matriz real A de ordem n invertível. A sua inversa A^{-1} também é invertível e vale $(A^{-1})^{-1} = A$.

Propriedade 2 Se A e B são matrizes quadradas reais invertíveis de ordem n . O produto AB também é invertível e sua inversa é dada por $(AB)^{-1} = B^{-1}A^{-1}$.

Propriedade 3 Uma matriz quadrada real que tenha uma linha (ou coluna) formada somente por zeros não é invertível.

2.2.4 Operações Elementares

Nesta seção, enumeraremos algumas operações que podem ser realizadas com as linhas de uma matriz. Essas operações são importantes para a resolução de sistemas lineares.

Definição 12 Dada uma matriz $m \times n$ qualquer, são **operações elementares** sobre essa matriz :

- i) permutar duas linhas dessa matriz
- ii) multiplicar qualquer linha por um escalar não nulo
- iii) adicionar a uma linha dessa matriz uma outra linha multiplicada por um escalar

Definição 13 Uma operação elementar aplicada a uma matriz identidade I produz uma matriz elementar correspondente E .

Propriedade 4 Multiplicar, à esquerda, uma matriz elementar por uma matriz A , corresponde a aplicar a operação elementar sobre a matriz A .

Propriedade 5 Uma matriz elementar E é invertível.

Pela propriedade 5, podemos ter uma seqüência de matrizes elementares $E_1, E_2, \dots, E_k$ de tal forma que $E = E_k \cdot E_{k-1} \cdot \dots \cdot E_2 \cdot E_1$ é uma matriz invertível. Esta matriz E é obtida aplicando-se as operações elementares associadas a $E_1, E_2, \dots, E_k$, nesta ordem, à matriz identidade.

2.3 Sistemas Lineares

Definição 14 Dados os números reais $\alpha_1, \alpha_2, \dots, \alpha_n, \beta$ ($n \geq 1$), a equação

$$\alpha_1 x_1 + \alpha_2 x_2 + \dots + \alpha_n x_n = \beta \quad (2.2)$$

onde os x_i são variáveis em $\mathbb{R}$ é chamada equação linear sobre $\mathbb{R}$ nas incógnitas $x_1, x_2, \dots, x_n$.

Definição 15 Uma solução da equação 2.2 é uma n -upla de números reais $(b_1, b_2, \dots, b_n)$ que satisfaz essa equação.

Definição 16 Um sistema de m equações lineares com n incógnitas ($m, n \geq 1$) é um conjunto de m equações lineares a n incógnitas, consideradas simultaneamente.

Um sistema de equações lineares pode ser escrito da seguinte forma:

$$S : \begin{cases} \alpha_{1,1}x_1 + \alpha_{1,2}x_2 + \dots + \alpha_{1,n}x_n = \beta_1 \\ \alpha_{2,1}x_1 + \alpha_{2,2}x_2 + \dots + \alpha_{2,n}x_n = \beta_2 \\ \vdots \\ \alpha_{m,1}x_1 + \alpha_{m,2}x_2 + \dots + \alpha_{m,n}x_n = \beta_m \end{cases} \quad (2.3)$$

Definição 17 Uma solução do sistema acima é uma n -upla de números reais $(b_1, b_2, \dots, b_n)$ que satisfaça cada uma das equações de S .

Definição 18 Quanto às soluções, um sistema de equações lineares S pode ser: **incompatível**, quando o sistema S não admite soluções; **compatível determinado**, quando o sistema S admite uma única solução; e **compatível indeterminado**, quando o sistema S admite mais de uma solução.

Definição 19 Dois sistemas de equações lineares S_1 e S_2 são ditos **equivalentes**, quando qualquer solução de um for solução do outro e vice-versa.

No nosso estudo consideraremos somente sistemas de equações lineares com n equações e n incógnitas. Nos referiremos a ele dizendo sistema de equações lineares de ordem n .

Considerando

$$A = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \dots & \alpha_{1,n} \\ \alpha_{2,1} & \alpha_{2,2} & \dots & \alpha_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1} & \alpha_{n,2} & \dots & \alpha_{n,n} \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \text{ e } b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

O sistema S em 2.3 para $m = n$ pode, então, ser escrito na forma matricial $Ax = b$, onde A recebe o nome de matriz dos coeficientes do sistema S e b o vetor de termos independentes.

Na determinação da solução de sistemas lineares podemos utilizar métodos diretos ou iterativos. Nos métodos diretos podemos proceder, basicamente, de duas formas:

- Determinando sistemas lineares equivalentes mais simples, por exemplo triangulares, equivalentes a S (neste caso a solução é determinada resolvendo-se o sistema triangular equivalente); ou
- Calculando a inversa da matriz A , se A for inversível. Nessa forma, a solução é dada resolvendo-se o produto $A^{-1}b$.

Neste trabalho, estaremos tratando apenas da solução de sistemas lineares através de métodos diretos.

2.4 Sistemas Triangulares

Os sistemas triangulares são sistemas de equações extremamente simples. Como veremos algoritmos para solução de sistemas lineares que recaem em sistemas triangulares ou os utilizam em cálculos intermediários, nesta seção estudaremos algoritmos para este tipo de sistemas.

Definição 20 *Um sistema de equações lineares com n equações e n incógnitas é dito triangular se a matriz $A = (a_{i,j})$ dos coeficientes é tal que $a_{i,j} = 0$ para $i > j$ ou $a_{i,j} = 0$ para $i < j$. No primeiro caso, o sistema é dito triangular inferior, e no segundo triangular superior.*

2.4.1 Substituições Recorrentes

Apresentaremos aqui um algoritmo que é clássico na resolução de sistemas triangulares, através de substituições recorrentes (*Back Substitutions*). O algoritmo para substituições recorrentes pode ser implementado em paralelo de maneira simples e está descrito a seguir. Este algoritmo resolve sistemas triangulares superiores. No caso de um sistema triangular inferior basta adaptar os laços. Para facilitar, consideramos o vetor b como a $(n+1)$ -ésima coluna de A [28].

Algoritmo Substituições Recorrentes

Entrada: a matriz estendida dos coeficientes de um sistema triangular $Ax = b$.

Saída: um vetor x de ordem n com a solução do sistema.

1. Para $k = n$ até 2 com passo -1 faça

Para $1 \leq i \leq k - 1$ faça em paralelo

Calcular $a_{i,n+1} = a_{i,n+1} - a_{i,k}a_{k,n+1}$

2. Para $1 \leq k \leq n$ faça em paralelo

Calcular $x_k = a_{k,n+1}$

A idéia do algoritmo está em se resolver a n -ésima equação, obtendo x_n . Com o resultado de x_n substituímos este valor nas $(n - 1)$ -ésimas equações em paralelo, sendo necessários 2 passos e $(n - 1)$ processadores. A seguir resolvemos a $(n - 1)$ -ésima equação, obtendo x_{n-1} e substituímos este valor nas $(n - 2)$ -ésimas equações simultaneamente, novamente em 2 passos, agora utilizando $(n - 2)$ dos $(n - 1)$ processadores da iteração anterior. E assim sucessivamente até chegarmos à primeira equação, quando usaremos apenas 1 processador dos $(n - 1)$ inicialmente necessários e apenas uma divisão. Dessa forma, o algoritmo traz a solução do sistema em tempo $\mathcal{O}(n)$ utilizando $\mathcal{O}(n)$ processadores.

2.4.2 Algoritmo de Sameh e Brent

Este algoritmo foi obtido em [28, 38].

Consideremos um sistema de equações lineares $n \times n$, na forma matricial, $Ax = b$, com A triangular inferior e tal que $a_{i,i} = 1, i = 1, \dots, n$.

Seja $e_j = (0, 0, \dots, 1, 0, \dots, 0)^t$ um vetor de n posições com a unidade na posição j e as outras posições nulas e seja $a_i = (0, 0, \dots, 0, a_{i+1,i}, a_{i+2,i}, \dots, a_{n,i})^t$ um vetor com as primeiras i posições nulas.

Para $i = 1, \dots, n$ definimos

$$A_i = I + a_i e_i^t = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & a_{i+1,i} & 1 & \cdots & 0 \\ 0 & 0 & \cdots & a_{i+2,i} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,i} & 0 & \cdots & 1 \end{bmatrix}$$

Analogamente, definimos

$$\overline{A}_i = I - a_i e_i^t$$

A partir disto temos propriedades que são imediatas.

Propriedade 6 $A_i^{-1} = \overline{A}_i$, para $i = 1, \dots, n$.

Propriedade 7

$$A_i A_{i+1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & a_{i+1,i} & 1 & \cdots & 0 \\ 0 & 0 & \cdots & a_{i+2,i} & a_{i+2,i+1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,i} & a_{n,i+1} & \cdots & 1 \end{bmatrix}$$

para $i = 1, \dots, n-1$.

Propriedade 8

$$A = A_1 \cdot A_2 \cdot \cdots \cdot A_n$$

e

$$A^{-1} = \overline{A}_n \cdot \overline{A}_{n-1} \cdot \cdots \cdot \overline{A}_2 \cdot \overline{A}_1$$

A solução do sistema $Ax = b$ pode então ser dada por

$$x = \overline{A}_n \cdot \overline{A}_{n-1} \cdot \cdots \cdot \overline{A}_2 \cdot \overline{A}_1 \cdot b$$

Podemos então obter a solução deste sistema a partir da expressão acima, usando um algoritmo *fan-in* associativo descrito a seguir.

Algoritmo Sameh e Brent

Entrada: $\overline{A}_i^{(0)} = \overline{A}_i$, para $i = 1, \dots, n$, $b^{(0)} = b$, obtidos do sistema $Ax = b$.

Saída: um vetor com a solução do sistema.

1. Para $j = 0$ até $k-2$ faça

 Para $i \in \{1, 2, \dots, 2^{k-j-1} - 1\}$ faça em paralelo

 Calcular $\overline{A}_i^{(j+1)} = \overline{A}_{2i+1}^{(j)} \overline{A}_{2i}^{(j)}$ em paralelo

 Calcular $b^{(j+1)} = \overline{A}_1^{(j)} b^{(j)}$ em paralelo

2. Calcular $b^{(k)} = \overline{A}_1^{(k-1)} b^{(k-1)}$ em paralelo

Figura 2.1: Árvore de execução do algoritmo de Sameh e Brent para uma matriz de ordem $n = 7$.

O paralelismo presente neste algoritmo não é tão claro. A seqüência a seguir mostra este paralelismo. Esta descrição também é necessária para a determinação do tempo de processamento e número de processadores. A figura 2.1 ilustra o funcionamento deste algoritmo para $n = 7$.

Consideremos $n = 2^k - 1, k \geq 1$.

Numa dada iteração j temos

$$\bar{A}_i^{(j)} = \begin{bmatrix} I_i^{(j)} & 0 & 0 \\ 0 & L_i^{(j)} & 0 \\ 0 & S_i^{(j)} & \bar{I}_i^{(j)} \end{bmatrix}$$

$I_i^{(j)}$ é uma matriz identidade de ordem $i2^j - 1$

$\bar{I}_i^{(j)}$ é uma matriz identidade de ordem $2^k - (i + 1)2^j$

$L_i^{(j)}$ é uma matriz triangular de ordem 2^j

$S_i^{(j)}$ é uma matriz de ordem $(2^k - (i + 1)2^j) \times 2^j$

Podemos particionar $S_i^{(j)}$ da seguinte forma

$$S_i^{(j)} = \begin{bmatrix} U_i^{(j)} \\ V_i^{(j)} \end{bmatrix}$$

onde $U_i^{(j)}$ é uma matriz quadrada de ordem 2^j e $V_i^{(j)}$ é uma matriz de ordem $(2^k - (i+2)2^j) \times 2^j$.

Quando $j = 0$ temos

$$L_i^{(0)} = [1]$$

$$S_i^{(0)} = \begin{bmatrix} -a_{i+1,i} \\ -a_{i+2,i} \\ \vdots \\ -a_{n,i} \end{bmatrix}$$

onde $U_i^{(0)} = [-a_{i+1,i}]$.

$I_i^{(0)}$ e $\bar{I}_i^{(0)}$ são matrizes de ordem $i-1$ e $2^k - (i+1) = n-i$.

A partir disto, teremos

$$\bar{A}_i^{(j+1)} = \bar{A}_{2i+1}^{(j)} \bar{A}_{2i}^{(j)}$$

onde

$$L_i^{(j+1)} = \begin{bmatrix} L_{2i}^{(j)} & 0 \\ L_{2i+1}^{(j)} U_{2i}^{(j)} & L_{2i+1}^{(j)} \end{bmatrix}$$

e

$$S_i^{(j+1)} = \begin{bmatrix} S_{2i+1}^{(j)} U_{2i}^{(j)} + V_{2i}^{(j)} & S_{2i+1}^{(j)} \end{bmatrix}$$

Analogamente, $b^{(j)}$ pode ser particionado da seguinte forma

$$b^{(j)} = \begin{bmatrix} g_1^{(j)} \\ g_2^{(j)} \\ g_3^{(j)} \end{bmatrix}$$

onde $g_1^{(j)}$ e $g_2^{(j)}$ são vetores coluna de tamanho $2^j - 1$ e 2^j , respectivamente. Assim, o tamanho de $g_3^{(j)}$ é $2^k - 2^{j+1}$.

Temos então,

$$b^{(j+1)} = \begin{bmatrix} g_1^{(j+1)} \\ g_2^{(j+1)} \\ g_3^{(j+1)} \end{bmatrix} = \bar{A}_1^{(j)} b^{(j)}$$

onde

$$g_1^{(j+1)} = g_1^{(j)}$$

$$\begin{aligned} g_2^{(j+1)} &= L_1^{(j)} g_2^{(j)} \\ g_3^{(j+1)} &= S_1^{(j)} g_2^{(j)} + g_3^{(j)}. \end{aligned}$$

A descrição anterior é útil para demonstrar o teorema a seguir, que traz a complexidade de tempo e o número de processadores para este algoritmo. Esta complexidade não é tão simples de se obter quanto a das substituições recorrentes, em vista das economias no número de operações obtidas pelos produtos de matrizes envolvendo matrizes triangulares.

Teorema 2 *Um sistema triangular de equações lineares de ordem n pode ser resolvido, utilizando-se o algoritmo de Sameh e Brent, em $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^3)$ processadores.*

Demonstração.

Sejam $t_i^{(j)}$ e $p_i^{(j)}$ o tempo mínimo e o número de processadores necessários para calcular $\overline{A}_i^{(j+1)}$. Sejam, também, $r^{(j)}$ e $q^{(j)}$ o tempo mínimo e o número de processadores necessários para calcular $b^{(j+1)}$.

Seja

$$T^{(j)} = \max \{r^{(j)}, t_i^{(j)} \text{ para } i \in \{1, 2, \dots, 2^{k-j-1} - 1\}\}$$

o tempo necessário para se calcular o laço interno do algoritmo para um dado j .

$$p^{(j)} = q^{(j)} + \sum_{i=1}^{2^{k-j-1}-1} p_i^{(j)}$$

é o número de processadores necessários para se calcular o laço interno em tempo $T^{(j)}$.

$$T = r^{k-1} + \sum_{j=0}^{k-2} T^{(j)} \quad (2.4)$$

e

$$P = \max \{p^{(j)}, j = 0, \dots, k-1\} \quad (2.5)$$

o tempo mínimo e o número de processadores necessários para o algoritmo todo.

Iremos analisar separadamente os cálculos de $\overline{A}_i^{(j+1)}$ e $b^{(j+1)}$.

Cálculo de $\overline{A}_i^{(j+1)}$.

$\overline{A}_i^{(j+1)}$ é dado pelo produto $\overline{A}_{2i+1}^{(j)} \overline{A}_{2i}^{(j)}$. As duas matrizes que são os fatores deste produto são matrizes de bloco dadas por 2.4.2. O produto é então obtido como em 2.4.2. Dessa forma na determinação deste produto não é necessário executar-se todas as operações envolvidas em tal produto, bastando concentrarmo-nos na obtenção dos novos blocos, economizando cálculos e processadores. Assim, o cálculo de $\overline{A}_i^{(j+1)}$ se reduz a

(i) Cálculo de $L_{2i+1}^{(j)} U_{2i}^{(j)}$.

Nesta matriz produto, cada coluna é obtida através de 2^j produtos internos de vetores de tamanho 2^j . Podemos obter cada um desses produtos internos em $j + 1$ passos (1 multiplicação e j adições usando a idéia do algoritmo *fan-in* associativo). Como $L_{2i+1}^{(j)}$ é triangular inferior, o número de processadores necessários para o cálculo de cada coluna dessa matriz produto é $2^{j-1}(2^j + 1)$. Daí, são necessários $2^{2j-1}(2^j + 1)$ processadores para o cálculo deste produto.

(ii) Cálculo de $S_{2i+1}^{(j)} U_{2i}^{(j)}$.

O cálculo de cada um dos $2^{(k+j)} - (i + 1)2^{2j+1}$ elementos desta matriz produto envolve o produto interno de vetores de tamanho 2^j . O tempo necessário é $j + 1$ passos e o número de processadores necessários é $(2^{(k+j)} - (i + 1)2^{2j+1})2^j$.

(iii) Cálculo de $S_{2i+1}^{(j)} U_{2i}^{(j)} + V_{2i}^{(j)}$.

Usando o mesmo número de processadores do item anterior, essa matriz soma pode ser obtida em apenas um passo.

Os itens (i) e (ii) podem ser feitos simultaneamente, uma vez que eles são independentes. Já o item (iii) depende da conclusão do cálculo do item (ii). Daí, de (i) e combinando (ii) e (iii), segue-se que

$$t_i^{(j)} = j + 2. \quad (2.6)$$

De (i) e (ii) (alguns dos processadores usados na determinação de (i) e (ii) são usados, novamente, para determinar a soma do item (iii)), segue-se que

$$\begin{aligned} p_i^{(j)} &= 2^{2j-1}(2^j + 1) + (2^{k+j} - (i + 1)2^{2j+1})2^j = \\ &= 2^{2j}(2^k + \frac{1}{2}) - 2^{3j}(2i + \frac{3}{2}). \end{aligned}$$

Cálculo de $b^{(j+1)}$.

Como a matriz $A_1^{(j)}$ é uma matriz em blocos e nos cálculos para determinação de $b^{(j+1)}$ usamos produtos por matrizes triangulares e por matrizes de ordem menor que n , o número de operações e o número de processadores envolvidos é menor do que se, simplesmente, fizéssemos o produto sem utilizar estas propriedades. Dessa forma, o cálculo de $b^{(j+1)}$ pode ser reduzido a

(iv) Cálculo de $L_1^{(j)} g_2^{(j)}$.

Cada elemento deste vetor pode ser obtido como produto interno de vetores de tamanho 2^j . O cálculo pode ser completado em $j + 1$ passos usando $2^{j-1}(2^j + 1)$ processadores.

(v) Cálculo de $S_1^{(j)} g_2^{(j)} + g_3^{(j)}$.

Cada elemento deste vetor pode ser obtido em $j + 2$ passos usando $(2^k - 2^{j+1})2^j$ processadores.

De (iv) e (v) segue-se que

$$r^{(j)} = (j + 2) \quad (2.7)$$

e

$$\begin{aligned} q^{(j)} &= 2^{j-1}(2^j + 1) + (2^k - 2^{j+1})2^j = \\ &= 2^j(2^k + \frac{1}{2}) - \frac{3}{2}2^{2j}. \end{aligned}$$

De 2.4, 2.6 e 2.7, obtemos T que é dado por

$$T = \sum_{j=0}^{k-1} j + 2 = \frac{k(k+3)}{2} = \mathcal{O}(\log^2 n).$$

Consideremos

$$\begin{aligned} p^{(j)} &= q^{(j)} + \sum_{i=1}^{2^{k-j-1}-1} p_i^{(j)} = \\ &= 2^j(2^k + \frac{1}{2}) - \frac{3}{2}2^{2j} + \sum_{i=1}^{2^{k-j-1}-1} 2^{2j}(2^k + \frac{1}{2}) - 2^{3j}(2i + \frac{3}{2}) = \\ &= \frac{3}{2}2^{3j} - \frac{2^{2j}}{4}(5 \cdot 2^k + 8) + \frac{2^j}{4}(2^{2k} + 5 \cdot 2^k + 2). \end{aligned}$$

Considerando $s = 2^j$ obtemos a seguinte função

$$f(s) = \frac{3}{2}s^3 - \frac{s^2}{4}(5 \cdot 2^k + 8) + \frac{s}{4}(2^{2k} + 5 \cdot 2^k + 2).$$

Pode-se demonstrar que o máximo desta função (para $n \geq 8$) ocorre, aproximadamente, em $s = \frac{2^k}{8}$. De 2.5, temos que

$$P = \frac{15}{1024}2^{3k} + \frac{1}{8}2^{2k} + \frac{1}{16}2^k = \mathcal{O}(n^3).$$

□

2.5 Algoritmos Básicos com Matrizes.

Nesta seção, apresentaremos alguns algoritmos paralelos básicos de manipulação de matrizes.

2.5.1 Produto Interno

Dados dois vetores $x = (x_1, x_2, \dots, x_n)$ e $y = (y_1, y_2, \dots, y_n)$ de $n = 2^k$ ($k \geq 0$) posições, o produto interno de x e y é dado por $xy = \sum_{i=1}^n x_i y_i$. O algoritmo para determiná-lo é dado a seguir.

Algoritmo Produto interno

Entrada: vetores $x = (x_1, x_2, \dots, x_n)$ e $y = (y_1, y_2, \dots, y_n)$, $n = 2^k$ ($k \geq 0$)

Saída: $P = xy$.

1. Para $1 \leq i \leq n$ faça em paralelo
 Calcular $p_i = x_i y_i$
2. Para $i = 1$ até k faça
 Para $1 \leq j \leq 2^{k-i}$ faça em paralelo
 Calcular $p_j = p_{2j-1} + p_{2j}$
3. $P = p_1$

O item 1 é realizado em 1 passo utilizando n processadores. O item 2 pode ser concluído em $k = \log n$ passos usando um máximo de $\frac{n}{2}$ processadores. Finalmente, o passo 3 é feito em 1 passo usando 1 processador. Assim, o algoritmo leva um total de $\mathcal{O}(\log n)$ passos, usando $\mathcal{O}(n)$ processadores.

2.5.2 Produto de Matriz por Vetor

Dados uma matriz quadrada $A = (a_{i,j})$ de ordem n (poderia ser uma matriz de ordem $m \times n$) e um vetor $x = (x_1, x_2, \dots, x_n)$ de n posições ($n = 2^k$). O vetor P resultante do produto de A por x é dado por $P_i = \sum_{j=1}^n a_{i,j} x_j$, $i = 1, \dots, n$. A solução pode ser encontrada pelo algoritmo

Algoritmo Produto matriz por vetor

Entrada: uma matriz $A = (a_{i,j})$ e um vetor $x = (x_1, x_2, \dots, x_n)$

Saída: $P = Ax$.

1. Para $1 \leq i \leq n$ e $1 \leq j \leq n$ faça em paralelo
 Calcular $c_{i,j} = a_{i,j} x_j$
2. Para $l = 1$ até k faça
 Para $1 \leq i \leq n$ e $1 \leq j \leq 2^{k-l}$ faça em paralelo
 Calcular $c_{i,j} = c_{i,2j-1} + c_{i,2j}$
3. Para $1 \leq i \leq n$ faça em paralelo
 $P_i = c_{i,1}$

O item 1 pode ser realizado em 1 passo usando n^2 processadores, com isso todos os produtos necessários serão feitos neste passo. O item 2 leva $\log n$ passos usando um máximo de $\frac{n^2}{2}$ processadores. O vetor P é então obtido no item 3 com mais 1 passo e n processadores. Dessa forma, este algoritmo é concluído em $\mathcal{O}(\log n)$ passos utilizando $\mathcal{O}(n^2)$ processadores.

2.5.3 Produto de Matrizes

Baseado na idéia do algoritmo *fan-in* associativo utilizado nos dois casos anteriores, podemos construir um algoritmo de produto de matrizes.

Dadas duas matrizes quadradas de ordem $n = 2^k$, $k \geq 0$, $A = (a_{i,j})$ e $B = (b_{i,j})$, vamos construir uma matriz $P = (p_{i,j})$ também de ordem n , tal que $P = AB$. Utilizamos uma variável auxiliar tridimensional C .

Algoritmo Produto matriz por vetor

Entrada: as matrizes $A = (a_{i,j})$ e $B = (b_{i,j})$ de ordem n .

Saída: $P = AB$.

1. Para $1 \leq i \leq n$, $1 \leq j \leq n$ e $1 \leq g \leq n$ faça em paralelo
 Calcular $c_{i,j,g} = a_{i,k}b_{k,j}$
2. Para $l = 1$ até k faça
 Para $1 \leq i \leq n$, $1 \leq j \leq n$ e $1 \leq g \leq 2^{k-i}$ faça em paralelo
 Calcular $c_{i,j,g} = c_{i,j,2g-1} + c_{i,j,2g}$
3. Para $1 \leq i \leq n$ e $1 \leq j \leq n$ faça em paralelo
 $P_{i,j} = c_{i,j,1}$

A etapa 1 é feita em 1 passo usando n^3 processadores. Ao final desta etapa todos os produtos que deveriam ser feitos já estão calculados, restando apenas as somas. Na etapa 2, as somas são realizadas em $\log n$ passos, usando $n^3/2$ processadores. Finalizando os cálculos, a matriz P é obtida em mais 1 passo utilizando n^2 processadores. Concluimos, então, que o algoritmo todo leva $\mathcal{O}(\log n)$ passos, utilizando $\mathcal{O}(n^3)$ processadores.

Como vimos, estes três problemas estão em NC^1 .

2.5.4 Potências de Matrizes

Dada uma matriz quadrada A de ordem n , $2^{k-1} < n \leq 2^k$ e $k \geq 1$, as n potências de A podem ser obtidas em paralelo usando a idéia do algoritmo *fan-in* associativo.

Em algumas circunstâncias, como veremos mais adiante, queremos calcular uma série de potências de matrizes. Isto é, dada A uma matriz quadrada de ordem n , calcular

$A^j, j = 1, \dots, n$. O algoritmo abaixo calcula esta sequência de potências. Novamente a idéia utilizada é a do algoritmo *fan-in* associativo.

Algoritmo potências de matrizes

Entrada: uma matriz quadrada A de ordem $n = 2^k, k \geq 0$.

Saída: uma sequência de potências $A^j, j = 1, \dots, n$, da matriz A .

1. Para $i = 1$ até k faça

Para $j \in \{2^{i-1} + 1, 2^{i-1} + 2, \dots, \min(2^i, n)\}$ faça em paralelo

Calcular $A^j = A^{2^{i-1}} A^{j-2^{i-1}}$ em paralelo

Em cada etapa i , 2^{i-1} potências podem ser calculadas independentemente a partir das matrizes calculadas nas etapas anteriores. Assim, usando $2^{i-1}n^3$ processadores estas potências podem ser calculadas em $\mathcal{O}(\log n)$ passos. Na k -ésima etapa são usados $[\min(2^k, n) - 2^{k-1}]n^3$ processadores. Como $n \leq 2^k$, o algoritmo leva ao todo $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^4)$ processadores, na realidade são necessários $n^4/2$ processadores.

2.6 Conclusão

Neste capítulo, pudemos ver alguns conceitos básicos de matrizes e sistemas lineares e algumas operações com matrizes. Descrevemos e discutimos os algoritmos de substituições recorrentes e de Sameh e Brent para a solução de sistemas triangulares, que serão muito úteis nos próximos capítulos. O algoritmo de substituições recorrentes é simples de ser descrito, e sua complexidade de tempo e o número de processadores necessários são óbvios e naturalmente obtidos. No algoritmo de Sameh e Brent detalhamos seu funcionamento. Sua complexidade e número de processadores necessários foram cuidadosamente obtidos (teorema 2) por não serem triviais, este algoritmo traz a solução em $\mathcal{O}(\log^2 n)$ passos utilizando $\mathcal{O}(n^3)$ processadores. Foram apresentados algoritmos paralelos para operações básicas com matrizes: produto interno, produto de vetor por matriz, produto de matrizes e sequência de potências de matrizes. Os três primeiros são obtidos em $\mathcal{O}(\log n)$ passos, utilizando, respectivamente, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ e $\mathcal{O}(n^3)$ processadores, e o último leva tempo $\mathcal{O}(\log^2 n)$ passos com $\mathcal{O}(n^4)$ processadores.

Capítulo 3

Fatoração LU

3.1 Introdução

Neste capítulo, estudaremos um algoritmo para solução de sistemas lineares que, atualmente, ainda é amplamente utilizado. A descrição do algoritmo paralelo não é difícil em vista do paralelismo transparente que ele possui. Inicialmente, apresentaremos a idéia do algoritmo e sua descrição. Estudaremos alguns variantes do algoritmo e alguns detalhes de implementação. Neste capítulo utilizamos as referências [5, 23, 28, 31].

3.2 A Fatoração LU

Dado um sistema de equações lineares na forma matricial $Ax = b$ com A quadrada de ordem n . Estamos interessados em, a partir do sistema original, recair em sistemas cuja estrutura da matriz dos coeficientes seja mais simples e dos quais conheçamos algoritmos para determinar sua solução.

No caso, desejamos obter sistemas triangulares, pois, como vimos, os sistemas triangulares são bastante fáceis de serem solucionados.

A idéia principal da solução deste sistema está em particionarmos a matriz A num produto de matrizes triangulares, $A = LU$ com L triangular inferior e U triangular superior.

A partir daí, a solução do sistema $Ax = b$ pode ser dada resolvendo-se os sistemas triangulares

$$Ly = b \quad \text{e} \tag{3.1}$$

$$Ux = y \tag{3.2}$$

nesta ordem.

Considerando que tal fatoração pode ser feita, como veremos adiante, temos

$$Ax = (LU)x = L(Ux) = Ly = b.$$

Portanto, resolvendo os sistemas 3.1 e 3.2 obtemos a solução do sistema $Ax = b$.

3.3 As Matrizes L e U

A determinação das matrizes L e U reside na possibilidade de anularmos posições de um vetor .

Seja $w = (w_1, w_2, \dots, w_n)^t$ e definamos uma matriz

$$M_i = I - me_i^t$$

onde

$e_i = (0, 0, \dots, 1, 0, \dots, 0)^t$ com 1 na posição i e 0 nas restantes,

I é a matriz identidade de ordem n e

o vetor m é dado por

$$m = (0, 0, \dots, m_{i+1}, m_{i+2}, \dots, m_n)^t, \text{ com}$$

$$m_j = \frac{w_j}{w_i}, j = i + 1, \dots, n.$$

$$M_i = \begin{bmatrix} 1 & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & \dots & 0 \\ 0 & 0 & \dots & -m_{i+1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -m_n & \dots & 1 \end{bmatrix}$$

O produto $M_i w$ é dado por

$$M_i w = Iw - me_i^t w = (w_1, w_2, \dots, w_i, 0, \dots, 0)^t,$$

uma vez que

$$Iw = w,$$

$$me_i^t = \begin{bmatrix} 0 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 0 & \cdots & m_{i+1} & \cdots & 0 \\ 0 & 0 & \cdots & m_{i+2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & m_n & \cdots & 0 \end{bmatrix},$$

onde a única coluna não nula é a i -ésima, e

$$me_i^t w = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ m_{i+1}w_i \\ m_{i+2}w_i \\ \vdots \\ m_n w_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ w_{i+1} \\ w_{i+2} \\ \vdots \\ w_n \end{bmatrix}, \text{ pois } m_j w_i = \frac{w_j}{w_i} w_i, \text{ para } j = i+1, \dots, n.$$

Se considerarmos o vetor w como sendo uma das colunas da matriz A , teremos

$$A' = M_i A = A - me_i^t A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,i} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,i} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,i} & \cdots & a_{i,n} \\ a'_{i+1,1} & a'_{i+1,2} & \cdots & 0 & \cdots & a'_{i+1,n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a'_{n,1} & a'_{n,2} & \cdots & 0 & \cdots & a'_{n,n} \end{bmatrix}.$$

Dessa forma, se multiplicarmos a matriz A por uma seqüência de matrizes $M_1, M_2, \dots, M_{n-1}$, obteremos

$$A^{(i)} x = b^{(i)}$$

onde $A^{(0)} = A$ e $b^{(0)} = b$, e para $1 \leq i \leq n-1$

$$A^{(i)} = M_i A^{(i-1)}$$

$$b^{(i)} = M_i b^{(i-1)}.$$

Ao final desta seqüência de produtos, a matriz $A^{(n-1)}$ será uma matriz triangular superior. Teremos, então,

$$U = A^{(n-1)} \text{ e } y = b^{(n-1)}.$$

Claramente $M_i, 1 \leq i \leq n-1$ é uma matriz inversível e

$$M_i^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ 0 & 0 & \cdots & m_{i+1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & m_n & \cdots & 1 \end{bmatrix}$$

Como queremos A fatorada no produto $A = LU$ e $U = M_{n-1} \cdots M_2 M_1 A$, segue-se que $L = M_1^{-1} \cdot M_2^{-1} \cdots M_{n-1}^{-1}$.

Dessa forma, determinamos as matrizes L e U .

3.4 O Algoritmo da Eliminação de Gauss

Podemos representar um sistema de equações na sua forma matricial $Ax = b$. Assim, as operações elementares definidas na seção 2.2.4 podem ser estendidas às equações deste sistema. Como uma matriz elementar E é inversível, o sistema obtido após a operação elementar é equivalente ao sistema original. $Ax = b \Leftrightarrow EAx = Eb$.

Exemplo.

Dado o sistema

$$S : \begin{cases} 3x_1 + 4x_2 = 9 \\ 6x_1 + x_2 = 4 \end{cases}$$

somando-se à segunda equação a primeira equação multiplicada por -2 obtemos

$$S' : \begin{cases} 3x_1 + 4x_2 = 9 \\ \quad \quad + -5x_2 = -14 \end{cases}$$

que é um sistema triangular superior. Como S' foi obtido de S através de uma operação elementar, S e S' são equivalentes. Para determinarmos a solução de S basta resolver o sistema S' . Dessa forma, $x_1 = -\frac{11}{15}$ e $x_2 = \frac{14}{5}$.

O exemplo nos dá uma noção de como um algoritmo deve proceder. O algoritmo de eliminação de Gauss, ou, simplesmente, algoritmo de Gauss, trabalha, justamente, como no exemplo. Através de operações elementares sobre as equações, obtemos um sistema triangular superior equivalente. A seguir descreveremos a idéia contida no algoritmo.

Primeiramente, vamos anular as posições abaixo do elemento $a_{1,1}$. Para isto utilizaremos uma operação elementar para cada uma das $(n-1)$ linhas abaixo da primeira.

Em seguida repetiremos o processo para o elemento $a_{2,2}$ e as $(n - 2)$ linhas abaixo deste elemento. Continuamos dessa maneira até obtermos o sistema triangular superior. Devemos notar que as posições previamente anuladas de uma coluna não são alteradas na etapa seguinte, esta propriedade vale sempre e é importante pois caracteriza este tipo de procedimento.

Baseado no que foi descrito, consideremos $E_{i,j}$ a matriz elementar correspondente à operação elementar que anula o elemento na posição i, j .

Seja a matriz

$$F_j = E_{n,j} \cdot E_{n-1,j} \cdots \cdots E_{j+2,j} \cdot E_{j+1,j}, j = 1, \dots, n-1.$$

O produto $F_j A$ tem os elementos abaixo de $a_{j,j}$, $j = 1, \dots, n-1$, anulados. Essa matriz F_j é justamente a matriz M_j descrita na seção 3.3.

Seja $F = F_{n-1} \cdot F_{n-2} \cdots \cdots F_2 \cdot F_1$, FA é a matriz U e Fb é o vetor y descritos na seção anterior.

Se anexarmos à matriz A uma $(n + 1)$ -ésima coluna formada pelas componentes do vetor b , o produto FA fornecerá na $(n + 1)$ -ésima coluna a solução do sistema.

A fatoração LU é uma descrição algébrica mais formal do algoritmo de eliminação de Gauss. Este algoritmo paralelo está descrito a seguir.

Algoritmo de Gauss

Entrada: matriz dos coeficientes A estendida.

Saída: solução do sistema $Ax = b$ contido na $(n + 1)$ -ésima coluna de A .

1. Para $k = 1$ até $n - 1$ faça

Para $j = linha + 1, \dots, n + 1$ faça em paralelo

Calcular $a_{k,j} = a_{k,j} / a_{k,k}$

Para $i = k + 1, \dots, n$ e $j = linha + 1, \dots, n + 1$ faça em paralelo

Calcular $a_{i,j} = a_{i,j} - a_{i,k} \cdot a_{k,j}$

2. Calcular $a_{n,n+1} = a_{n,n+1} / a_{n,n}$

No item 1 do algoritmo, o elemento $a_{k,j}$ é denominado *multiplicador*.

Este algoritmo leva tempo $\mathcal{O}(n^3)$ seqüencialmente. O algoritmo paralelo leva $\mathcal{O}(n)$ passos usando $\mathcal{O}(n^2)$ processadores.

O ganho deste algoritmo é dado por

$$S_p(n) = \frac{\mathcal{O}(n^3)}{\mathcal{O}(n)} = \mathcal{O}(n^2),$$

a eficiência de processador é

$$E_p(n) \approx 1.$$

Como $S_p(n) = \Omega(p)$, este algoritmo tem ganho ótimo.

Apesar de ser um algoritmo de ganho ótimo como descrito acima, ele apresenta alguns problemas de estabilidade como veremos na próxima seção.

3.5 Eliminação Gaussiana com Pivotação

Os elementos $a_{ii}, i = 1, \dots, n-1$, da matriz dos coeficientes cujos elementos abaixo serão anulados são chamados de *pivôs*. Pelo que vimos até agora, para podermos proceder o algoritmo de Gauss, o pivô deve ser um elemento não nulo. Mais ainda, para garantirmos uma certa estabilidade no algoritmo estes pivôs não podem ser muito pequenos em relação aos demais valores abaixo deles. Ao aplicarmos o algoritmo de Gauss em sistemas com essa característica podemos incorrer em erros na solução do problema.

Para contornar este problema podemos trocar duas linhas da matriz, quando necessário. Isto não afeta o resultado, uma vez que a permutação de linhas é uma operação elementar. A este processo de permutar linhas damos o nome de *pivotação*.

Devemos evitar que os multiplicadores sejam números grandes em relação aos demais números abaixo dos pivôs, isto significa que os pivôs não devem ser números muito pequenos em relação aos demais. Para garantir isto devemos escolher um pivô de maneira que ele seja o maior possível, em valor absoluto, dentre as posições envolvidas. Este processo de se procurar o maior pivô, em valor absoluto, para depois proceder a permutação recebe o nome de *pivotação parcial*. Logo a seguir damos a descrição do algoritmo da eliminação gaussiana com pivotação.

Algoritmo de Gauss com pivotação

Entrada: a matriz A dos coeficientes estendida.

Saída: solução do sistema $Ax = b$ na $(n+1)$ -ésima coluna de A .

1. Para $k = 1$ até $n-1$ faça
 - Calcular $imax$ em paralelo
 - Trocar os elementos das linhas $imax$ e k em paralelo
 - Para $j = k+1, \dots, n+1$ faça em paralelo
 - Calcular $a_{k,j} = a_{k,j}/a_{k,k}$
 - Para $i = k+1, \dots, n$ e $j = k+1, \dots, n+1$ faça em paralelo
 - Calcular $a_{i,j} = a_{i,j} - a_{i,k} \cdot a_{k,j}$
2. Calcular $a_{n,n+1} = a_{n,n+1}/a_{n,n}$

O índice da linha que contém o máximo é armazenada em $imax$. Se utilizarmos o algoritmo padrão para determinação do máximo de uma sequência de n valores, que leva tempo $\mathcal{O}(n)$, o algoritmo acima levará tempo $\mathcal{O}(n^2)$ com $\mathcal{O}(n^2)$ processadores. Neste caso o ganho será $S_p(n) = \mathcal{O}(n)$ e a eficiência $E_p(n) = \frac{\mathcal{O}(n)}{\mathcal{O}(n^2)} = \mathcal{O}(\frac{1}{n})$. Para n grande, $E_p(n) \approx 0$. Teremos um algoritmo ruim.

Podemos utilizar a idéia do algoritmo *fan-in* associativo para determinação do pivô, que leva tempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ processadores. Assim, o algoritmo todo levará tempo $\mathcal{O}(n \log n)$ utilizando $\mathcal{O}(n^2)$ processadores. Dessa forma, o ganho será $S_p(n) = \frac{\mathcal{O}(n^3)}{\mathcal{O}(n \log n)} = \mathcal{O}(\frac{n^2}{\log n})$ e a eficiência $E_p(n) = \mathcal{O}(\frac{1}{\log n})$. Temos, para n grande, $E_p(n) \approx 0$ também, mas $E_p(n)$ tende a zero “mais devagar” do que no algoritmo padrão.

3.6 Algoritmo de Gauss-Jordan

O algoritmo de Gauss-Jordan é uma variante do algoritmo de eliminação gaussiana. A idéia básica continua sendo a mesma de se anular convenientemente posições. A diferença reside em que os zeros não aparecerão somente abaixo da diagonal principal, obtendo um sistema triangular superior, mas aparecerão, também, acima da diagonal principal. Ao final, teremos chegado a uma matriz diagonal. A solução do sistema é obtida bastando resolver as divisões que restaram.

O algoritmo paralelo é dado a seguir.

Algoritmo de Gauss-Jordan

Entrada: matriz dos coeficientes A estendida.

Saída: solução do sistema $Ax = b$ na $(n + 1)$ -ésima coluna de A .

1. Para $k = 1$ até $n - 1$ faça

 Para $j = k + 1, \dots, n + 1$ faça em paralelo

 Calcular $a_{k,j} = a_{k,j} / a_{k,k}$

 Para $i = k + 1, \dots, n$ e $j = k + 1, \dots, n + 1$ faça em paralelo

 Calcular $a_{i,j} = a_{i,j} - a_{i,k} \cdot a_{k,j}$

 Para $i = 1, \dots, k - 1$ e $j = k + 1, \dots, n + 1$ faça em paralelo

 Calcular $a_{i,j} = a_{i,j} - a_{i,k} \cdot a_{k,j}$

O ganho e a eficiência deste algoritmo são os mesmos do algoritmo de Gauss. Sequencialmente, o algoritmo de Gauss é melhor, porém, no caso paralelo, a situação se inverte.

3.7 Fatoração LU em Outros Modelos

A eficiência do algoritmo de Gauss com pivotação não é bom para n suficientemente grande. No modelo de memória compartilhada executamos os passos como está descrito no algoritmo paralelo. Entretanto, em um modelo MIMD podemos alterar a sequência de tarefas a serem executadas simultaneamente, como desenvolvido por Lord, Kowalik e Kumar [29], que passamos a descrever.

Uma tarefa é uma unidade indivisível de atividades computacionais. Um conjunto de tarefas $J = \{T_1, T_2, \dots, T_n\}$ e uma relação binária irreflexiva, antissimétrica e transitiva $\prec$ constituem um sistema de tarefas. A relação $\prec$ é denominada precedência. Se $T \prec T'$ então T deve completar sua execução antes de T' começar a sua.

Considerando o algoritmo a seguir.

Algoritmo de Gauss com pivotação

Entrada: a matriz A dos coeficientes estendida.

Saída: solução do sistema $Ax = b$ na $(n + 1)$ -ésima coluna de A .

1. Para $k = 1$ até $n - 1$ faça
 - Calcular $imax$
 - Trocar os elementos das linhas $imax$ e k
 - Para $j = k + 1$ até $n + 1$ faça
 - Calcular $a_{k,j} = a_{k,j}/a_{k,k}$
 - Para $j = k + 1$ até $n + 1$ faça
 - Para $i = k + 1$ até n faça
 - Calcular $a_{i,j} = a_{i,j} - a_{i,k} \cdot a_{k,j}$
2. Calcular $a_{n,n+1} = a_{n,n+1}/a_{n,n}$

Neste algoritmo consideremos uma tarefa como sendo um segmento de instruções que age numa certa coluna j para um certo valor de k . Assim, temos a tarefa T_k^k compreendendo o trecho

Calcular $imax$
 Trocar os elementos das linhas $imax$ e k
 Para $j = k + 1$ até $n + 1$ faça
 Calcular $a_{k,j} = a_{k,j}/a_{k,k}$

e a tarefa T_k^j , com $j > k$, é dada pelo trecho

Para $j = k + 1$ até $n + 1$ faça
 Para $i = k + 1$ até n faça
 Calcular $a_{i,j} = a_{i,j} - a_{i,k} \cdot a_{k,j}$

Estas tarefas serão denotadas por

$$J = \{T_k^j, 1 \leq k \leq j \leq n, k \leq n - 1\}.$$

A precedência dada pelo programa é

$$\prec = \{(T_k^j, T_m^l), j < l \text{ e } k = m, \text{ ou } k < m\}.$$

Dessa forma, $C = (J, \prec)$ é um sistema que representa o programa seqüencial.

A partir disto podemos montar um grafo equivalente a C e observarmos que o maior caminho passa pelos nós

$$T_1^1, T_1^2, T_2^2, T_2^3, T_3^3, \dots, T_{n-1}^{n-1}, T_{n-1}^n$$

com tamanho $n^2 - 1$.

Como o problema não pode ser resolvido em tempo menor que o comprimento deste caminho, temos em esquema utilizando no máximo $p = \lfloor \frac{n}{2} \rfloor$ processadores, onde cada processador j executará a seqüência de tarefas

$$T_1^{2j-1}, T_1^{2j}, T_2^{2j}, T_2^{2j+1}, \dots, T_{n-2(j-1)}^n.$$

Este esquema tem comprimento $n^2 - 1$, que é o tamanho do maior caminho, então este esquema é ótimo para $n/2$ processadores. Obtém-se com isto uma eficiência $E_p = 2/3$.

Assim, vimos que o modelo utilizado influiu no desempenho de um algoritmo se utilizarmos, eficientemente, as características do modelo na elaboração do algoritmo e na divisão de tarefas entre os processadores. Em um modelo de memória distribuída a maneira como os dados serão armazenados também influi na eficiência do algoritmo, um exemplo disto pode ser visto em Geist e Romine [20].

3.8 Implementação

Os algoritmos descritos anteriormente foram implementados em linguagem C. O algoritmo de Gauss é a base dos outros dois que apresentam diferenças apenas em alguns procedimentos. Os procedimentos comuns serão descritos apenas uma vez.

Apesar de na contagem dos passos não considerarmos a entrada de dados, já que supusemos que os dados estariam inicialmente na memória, na execução dos programas teremos que ler os dados que estão em um arquivo.

Os pontos em que foram feitas contagens ou foi marcado o tempo estão marcados com um asterisco no início da linha. A cronometragem envolve a ativação e a desativação do relógio e a adição do tempo ao respectivo acumulador. No caso das contagens apenas adiciona-se o número de operações efetuadas aos contadores de operações e total. Estas implemetações são sequenciais. As operações que podem ser feitas simultaneamente são contadas em uma mesma posição de uma matriz bidimensional que corresponde à utilização de diferentes processadores.

3.8.1 Algoritmo de Gauss

Principal

Neste procedimento e no procedimento *processor* explicitamos a maneira como são feitas as contagens dos números de operações por processador. Utilizamos uma variável bidimensional P , $n \times n$, onde cada posição desta variável corresponde ao contador de operações de um processador. Como critério de escalonamento, utilizamos sempre a posição disponível de menores índices, considerando-se primeiro o índice de coluna, como pode ser observado na determinação dos índices da variável P . Com esta variável asseguramos que operações que são feitas simultaneamente utilizam contadores distintos. Em todas as listagens das implementações deste trabalho utilizamos esta variável P com a finalidade descrita acima, variando-se apenas as dimensões desta. As posições onde esta contagem deve ser incluída estão marcadas com um * à esquerda.¹

```
#define N 300
int k, Dm, linha, P[N][N];
double A[N][N+1], X[N];
void processor();
void solucao();
void impressao();

void main()
{ int i, j;

/** Entrada de dados. */
    entrada();
```

¹Esta variável pode armazenar os tempos de processamento bastando para isto modificar-se seu tipo.

```

for (linha=1;linha<Dm; linha++)
    processor();
A[Dm][Dm+1]=A[Dm][Dm+1]/A[Dm][Dm];
P[1][1]++;
A[Dm][Dm]=1;

for (k=Dm;k>1;k--)
    for (linha=1;linha<=k-1;linha++)
        solucao();

for(k=1;k<=Dm;k++)
    X[k]=A[k][Dm+1];

impressao();
}

```

A variável *Dm* armazena a dimensão da matriz. Na variável *A* está armazenada a matriz dos coeficientes estendida e na variável *X* a solução do sistema.

Procedimento processor

O procedimento a seguir é responsável pela triangularização da matriz dos coeficientes. Os cálculos são estendidos à última coluna, que contém o vetor *b*. Ao final, obteremos a matriz triangularizada. Mas, como estamos interessados apenas em determinar a solução do sistema, os cálculos são feitos apenas nas posições que não foram ou não estão sendo anuladas. Com isto, diminuímos o número de operações. Este procedimento é também utilizado no programa do algoritmo de Gauss com pivotação.

```

void processor()
{
    int i, j;
    float m;

    for (j=linha+1;j<=Dm+1;j++)
    {
        A[linha][j] = A[linha][j]/A[linha][linha];
        P[j-linha][1]++;
    }
}

```

```

for (i=linha+1;linha<=Dm;i++)
{
    m = A[i][linha];
    for (j=linha+1;j<=Dm+1;j++)
    {
        A[i][j] = A[i][j]-m*A[linha][j];
        P[i-linha][j-linha]+=2;
    }
}
}

```

Procedimento solucao

No procedimento a seguir, o sistema triangular superior obtido é resolvido utilizando-se substituições recorrentes.

```

void solucao()
{
* A[linha][Dm+1] = A[linha][Dm+1]-A[linha][k]*A[k][Dm+1];
}

```

Este procedimento apresenta apenas uma linha de comando, pois nesta descrição não estão explicitadas as contagens de tempo e número de operações.

3.8.2 Algoritmo de Gauss com pivotação

A implementação deste algoritmo basicamente acompanha a mesma do algoritmo de Gauss. A única diferença está na chamada do procedimento *pivotacao* antes do procedimento *processor*, que é o mesmo. O trecho

```

for (linha=1;linha<Dm;linha++)
    processor();

```

passa a ser

```

for (linha=1;linha<Dm;linha++)
{
    pivotacao();
    processor();
}

```

além da inclusão da declaração do protótipo do procedimento *pivotacao* no início do programa.

O procedimento *pivotacao* utiliza a idéia do algoritmo *fan-in* associativo. Inicialmente, a lista de dados de onde se vai determinar o máximo é copiada para o vetor *M* e os respectivos índices de linha para o vetor *I*. A troca de linhas não precisa ser feita explicitamente, pode-se apenas proceder-se uma troca de índices. Como estamos utilizando algoritmos paralelos a troca explícita pode ser feita em tempo constante e igual a um, usando-se *n* processadores. Neste algoritmo, além das operações aritméticas, estamos considerando as comparações como operações elementares.

O algoritmo *fan-in* associativo é aplicado em vetores de *n* posições onde *n* é uma potência de dois. No nosso caso, não é possível que todas as chamadas ao procedimento *pivotacao* envolvam vetores com tamanhos que sejam potências de dois. Poderíamos estender o tamanho do vetor em questão a uma potência de dois, completando as novas posições com zero, o que não alteraria o resultado do procedimento. Porém, preferimos incluir mais uma posição, com conteúdo nulo, a cada iteração do laço *for g*.

Procedimento *pivotacao*

```
void pivotacao()
{
    int i,j,h,g,Aux, Aux1,I[N];
    double Aux2, M[N];

    for (i=linha;i<=Dm;i++)
    {
        M[i-linha+1]=A[i][linha];
        I[i-linha+1] = i;
    }

    Aux = (int)ceil(log(Dm-linha+1)/log(2.0));
    Aux1 = 1;
    for (h=1;h<=Aux;h++)
    {
        Aux1 = Aux1 *2;
        for (g=1;g<=ceil((float)(Dm-linha+1)/Aux1);g++)
        {
            *   if (fabs(M[2*g-1]) <= fabs(M[2*g]))
                {
                    M[g] = M[2*g];
                    I[g] = I[2*g];
                }
        }
    }
}
```

```

    }
    else
    {
        M[g] = M[2*g-1];
        I[g] = I[2*g-1];
    }
}
M[g] = 0.0;
}

/* Troca as linhas I[1] e linha. */
for (j=1;j<=Dm+1;j++)
{
    Aux2 = A[I[1]][j];
    A[I[1]][j] = A[linha][j];
    A[linha][j] = Aux2;
}
}

```

3.8.3 Algoritmo de Gauss-Jordan

A diferença entre este algoritmo e o algoritmo de Gauss está no procedimento *processor* a seguir. Além disso, o procedimento *solucao* não é necessário.

```

void processor()
{
    int i,j;
    float m;

    for (j = linha+1;j<=Dm+1; j++)
*   A[linha][j] = A[linha][j]/A[linha][linha];

    for (i = linha + 1; i<= Dm;i++)
    {
        m = A[i][linha];
        for (j=linha+1;j<=Dm+1;j++)
*   A[i][j] = A[i][j] - m*A[linha][j];
    }

    for (i = linha - 1; i>= 1;i--)
    {

```

```

        m = A[i][linha];
        for (j=linha+1; j<=Dm+1; j++)
*       A[i][j] = A[i][j] - m*A[linha][j];
    }
}

```

Os resultados gerados por estes algoritmos serão apresentados e discutidos no capítulo 6.

3.9 Implementação na PVM

Estes algoritmos foram também implementados na PVM (*Parallel Virtual Machine*) que está descrita no apêndice A. Na PVM, os programas funcionam em uma forma mestre/escravo (*host/slave*). O programa que roda no mestre é responsável pela ativação dos programas que rodam nos escravos e no recebimento dos resultados gerados nestes. A seguir, descreveremos cada implementação. Novamente, as implementações são muito parecidas e descreveremos primeiramente o algoritmo de Gauss e depois as variações dos demais. Nesta implementação, nos preocupamos em cronometrar o tempo dos cálculos, medido em microssegundos. A cronometragem não abrangeu os tempos gastos com recebimento de dados e envio de resultados. O objetivo destas implementações é puramente ilustrativo da utilização da PVM uma vez que os resultados podem ser obtidos mais rapidamente e para matrizes maiores pelas implementações já discutidas na seção anterior. Além disso, a PVM poderia ser melhor explorada e mais útil se utilizássemos um modelo de memória distribuída, o que não é o nosso caso, pois é possível a comunicação direta entre processadores.

3.9.1 Algoritmo de Gauss

A estrutura do programa na PVM é basicamente a mesma que já foi feita. A ativação dos procedimentos é trocada por uma ativação de programas em outros processadores, os dados são enviados a estes processadores e estes enviam o resultado dos cálculos.

Programa gauss

```

#define N 300

int Dm, linha;
double A[N][N+1], X[N];

```

```

void main
{
    int linha, i, j, num, inst, msgtipo;
    double Pivo[6], dados[4], resultados[4];

    /* Entrada de dados. */
    entrada();

    if ((num=enroll("gauss"))<0)
    {
        pvm_perror("enroll");
        exit(1);
    }

    for (linha=1;linha<=Dm;linha++)
        for (i=linha+1;i<=Dm;i++)
            for (j=linha+1;j<=Dm+1;j++)
            {
                inst = initiate("processor","SUN4");
                Pivo[0] = A[linha][linha];
                Pivo[1] = A[linha][j];
                Pivo[2] = A[i][j];
                Pivo[3] = A[i][linha];
                Pivo[4] = i;
                Pivo[5] = j;

                initsend();
                msgtipo = 1;
                pundfloat(Pivo,6);
                snd("processor",inst,msgtipo);
            }

    msgtipo = 2;
    for (linha=1;linha<=Dm;linha++)
        for (i=1;i<=(Dm-linha)*(Dm+1-linha);i++)
        {
            rcv(msgtipo);
            getndfloat(resultado,4);
            A[(int)resultado[2]][(int)resultado[3]]=resultado[1];
            A[linha][(int)resultado[3]]=resultado[0];
        }

```

```

    }

*   A[Dm][Dm+1]=A[Dm][Dm+1]/A[Dm][Dm];

/* Resolve sistema triangular superior. */

for (k=Dm;k>1;k--)
{
    for (linha=1;linha<=k-1;linha++)
    {
        inst = initiate("trisu", "SUN4");
        dados[0] = A[linha][Dm+1];
        dados[1] = A[linha][k];
        dados[2] = A[k][Dm+1];
        dados[3] = linha;

        init send();
        msgtipo = 3;
        putndfloat(dados,4);
        snd("trisu",inst,msgtipo);
    }

    msgtipo = 4;
    for (linha=1;linha<=k-1;linha++)
    {
        rcv(msgtipo);
        getndfloat(resultado,2);
        A[(int)resultado[1]][Dm+1] = resultado[0];
    }
}

impressao();
leave();
}

```

Não chamamos os procedimentos *processor* e *solucao* que são substituídos por programas que rodarão nos escravos. Os dados estão nos vetores *Pivo* e *dados* que serão enviados aos programas *processor* e *trisu*, respectivamente. Os resultados serão devolvidos no vetor *resultados*. Também são enviados ao programa *processor* os índices da posição que está sendo alterada, e ao programa *trisu* é enviada a posição da coluna $Dm + 1$ que está sendo atualizada.

Programa processor

```

void main()
{
    int msgtipo, num, hostnum, bytes;
    double resultado[5], pivo[6];
    char host[15];

    num = enroll("processor");

    /*Recebimento dos dados. */

    msgtipo = 1;
    rcv(msgtipo);
    getndfloat(Pivo,6);
    rcvinfo(&bytes,&msgtipo,host,&hostnum);

    * resultado[0] = Pivo[1]/Pivo[0];
    * resultado[1] = Pivo[2]-Pivo[3]*resultado[0];
    resultado[2] = Pivo[4];
    resultado[3] = Pivo[5];

    /* Envio dos resultados */

    init send();
    putndfloat(resultado,4);
    msgtipo = 2;
    snd(host,hostnum,msgtipo);

    leave();
}

```

Este programa recebe os dados no vetor *Pivo*, executa os cálculos de atualização da posição passada pelo programa *gauss* armazenando no vetor *resultados* e os enviando de volta ao mestre.

Programa trisup

```

void main ()
{
    int msgtipo, num, hostnum, bytes;

```

```

double resultado[2], dados[4];
char host[15];

num = enroll("trisup");

/* Recebimento dos dados. */

msgtipo = 3;
rcv(msgtipo);
getndfloat(dados,4);
rcvinfo(&bytes,&msgtipo,host,&hostnum);

* resultado[0] = dados[0] -dados[1]*dados[2];
  resultado[1] = dados[3];

/* Envio dos resultados. */

init send();
putndfloat(resultado,2);
msgtipo = 4;
snd(host,hostnum,msgtipo);

leave();
}

```

A cada execução deste programa atualiza-se uma posição da coluna $Dm + 1$ da matriz dos coeficientes já triangularizada. Ao final, teremos nesta coluna a solução do sistema dado.

3.9.2 Algoritmo de Gauss com Pivotação

O programa que roda no mestre neste caso apresenta apenas um trecho a mais do que o correspondente do algoritmo de Gauss. Antes do trecho que ativa o programa *processor*, existe um trecho que ativa o programa *pivota* responsável pela busca da posição que tenha o maior valor absoluto; em seguida ativa-se o programa *troca* que faz a troca da linha do pivô e a linha do maior valor absoluto. Dessa forma, logo antes do trecho que ativa o programa *processor* temos o seguinte trecho.

```

/* Ativa o programa pivota. */
for (linha=1;linha<Dm;linha++) {

```

```

for (i=linha;i<=Dm;i++)
{
    M[i-linha+1]=A[i][linha];
    I[i-linha+1]=i;
}
Aux = (int)ceil(log(Dm-linha+1)/log(2.0));
Aux1 = 1;
for (h=1;h<=Aux;h++)
{
    Aux1 *= 2;
    for (g=1;g<=ceil((float)(Dm-linha+1)/Aux1);g++)
    {
        inst = initiate("pivota","SUN4");
        msgtipo = 5;
        initsend();
        putndfloat(&M[2*g-1],1);
        putndfloat(&M[2*g],1);
        putnint(&I[2*g-1],1);
        putnint(&I[2*g],1);
        snd("pivota",inst,msgtipo);
    }
    msgtipo = 6;
    for (g=1;g<=ceil((float)(Dm-linha+1)/Aux1);g++)
    {
        rcv(msgtipo);
        getndfloat(&M[g],1);
        getnint(&I[g],1);
    }
    M[g]=0.0;
}

/* Troca as posicoes. */
for (j=1;j<=Dm+1;j++)
{
    inst = initiate("troca","SUN4");
    Pivo[0] = A[I[1]][j];
    Pivo[1] = A[linha][j];
    Pivo[2] = j;

    initsend();
    msgtipo = 7;

```

```
    putndfloat(Pivo,3);
    snd("troca",inst,msgtipo);
}

msgtipo = 8;
for (j=1;j<=Dm+1;j++)
{
    rcv(msgtipo);
    getndfloat(resultado,3);
    A[I[1]][(int)resultado[2]]=resultado[0];
    A[linha][(int)resultado[2]]=resultado[1];
}
```

Programa pivota

```
void main()
{
    int msgtipo,num,hostnum,bytes,indices[2],indmax;
    double resultado[2],dados[3];
    char host[15];

    num = enroll("pivota");

    /* Recebimento dos dados. */

    msgtipo = 5;
    rcv(msgtipo);
    getndfloat(dados,2);
    getnint(indices,2);
    rcvinfo(&bytes,&msgtipo,host,&hostnum);

    * if (fabs(dados[0]) <= fabs(dados[1]))
    {
        resultado[0] = dados[1];
        indmax = indices[1];
    }
    else
    {
        resultado[0] = dados[0];
        indmax = indices[0];
    }
}
```

```
}

/* Envio dos resultados. */

    init send();
    putndfloat(&resultado[0],1);
    putndint(&indmax,1);
    msgtipo = 6;
    snd(host,hostnum,msgtipo);

    leave();
}
```

Este programa recebe do mestre um par de valores e um par de índices que são armazenados nos vetores *dados* e *índices*, respectivamente. Estes dados são comparados e o maior deles e o respectivo índice são devolvidos ao mestre.

Programa troca

```
void main()
{
    int msgtipo,mynum,hostnum,bytes,inum;
    double resultado[3];
    char host[15];

    num = enroll("troca");

/* Recebimento dos dados. */

    msgtipo = 7;
    rcv(msgtipo);
    getndfloat(resultado,3);
    rcvinfo(&bytes,&msgtipo,host,&hostnum);

/* Envio dos resultados. */

    init send();
    putndfloat(&resultado[1],1);
    putndfloat(&resultado[0],1);
```

```

    putndfloat(&resultado[2],1);
    msgtipo = 8;
    snd(host,hostnum,msgtipo);

    leave();
}

```

O programa recebe dois valores do mestre e os troca de posição. Além disso, ele recebe o índice da coluna, que é mandado de volta juntamente com os dados já trocados.

3.9.3 Algoritmo de Gauss-Jordan

Este programa está baseado no programa da eliminação de Gauss. Usa-se o mesmo programa *processor* no escravo. A diferença está na chamada a este programa que é feita duas vezes uma para a parte abaixo da diagonal principal e outra para a parte que está acima. Ao final, já teremos a solução na coluna $Dm + 1$ da matriz A .

Programa *gjordan*

```

#define N 300

int Dm, linha;
double A[N][N+1], X[N];

void main()
{
    int linha, i, j, num, inst, msgtipo;
    double Pivo[6];

    /* Entrada de dados. */
    entrada();

    if((num=enroll("gjordan"))<0)
    {
        pvm_perror("enroll");
        exit(1);
    }

    for (linha = 1;linha<=Dm;linha++)
    {

```

```

for(i=linha+1;i<=Dm;i++)
  for(j=linha;j<=Dm+1;j++)
  {
    inst = initiate("processor","SUN4");
    Pivo[0]=A[row][row];
    Pivo[1]=A[row][j];
    Pivo[2]=A[i][j];
    Pivo[3]=A[i][row];
    Pivo[4]=i;
    Pivo[5]=j;

    init send();
    msgtipo=1;
    putndfloat(Pivo,6);
    snd("processor",inst,msgtipo);
  }

for(i=linha-1;i>=1;i--)
  for(j=linha;j<=Dm+1;j++)
  {
    inst = initiate("processor","SUN4");
    Pivo[0]=A[linha][linha];
    Pivo[1]=A[linha][j];
    Pivo[2]=A[i][j];
    Pivo[3]=A[i][linha];
    Pivo[4]=i;
    Pivo[5]=j;

    init send();
    msgtipo=1;
    putndfloat(Pivo,6);
    snd("processor",inst,msgtipo);
  }

msgtipo=2;
for(i=1;i<=(Dm-1)*(Dm+2-linha);i++)
{
  rcv(msgtipo);
  getndfloat(resultado,4);
  A[(int)resultado[2]][(int)resultado[3]]=resultado[1];

```

```

        A[linha][(int)resultado[3]]=resultado[0];
    }
}

impressao();
leave();
}

```

3.9.4 Resultados

Na implementação dos algoritmos na PVM medimos o tempo², em microssegundos, de utilização de cada processador efetuando operações elementares. Não levamos em conta os tempos de envio e recebimento de dados. O tempo cronometrado em cada execução do escravo foi enviado juntamente com os outros resultados ao mestre, onde as matrizes de tempo eram atualizadas. Utilizamos 6 máquinas como escravos, mas, para efeito de marcação de tempo, consideramos a existência de n^2 processadores. Os resultados obtidos com a execução estão mostrados na tabela 3.1 onde figuram, também, os tempos dos respectivos programas através das implementações sequenciais da seção 3.8. A ordem máxima utilizada foi 30, esta limitação foi causada pelo tempo total de execução dos programas e pelos erros causados pelo grande volume de comunicações com os escravos.

n	PVM			Implementações		
	Gauss	Gauss(P)	Gauss-Jordan	Gauss	Gauss(P)	Gauss-Jordan
5	3.746	2.221	2.498	551	786	430
10	4.953	3.036	2.582	1.060	1.952	779
15	7.155	8.289	14.086	1.670	3.139	1.162
20	9.098	10.755	14.086	2.214	4.521	2.091
25	18.586	14.292	19.294	2.808	5.956	2.536
30	21.630	17.970	60.568	3.371	7.384	2.804

Tabela 3.1: Tabela do tempo (em microssegundos) máximo de utilização dos processadores pelos algoritmos de Gauss, Gauss com pivotação (Gauss(P)) e Gauss-Jordan utilizando a PVM e as implementações sequenciais.

A marcação do tempo de processamento em frações de segundos apresenta algumas distorções causadas pelo tráfego nos processadores. Por mais que executássemos os programas em períodos do dia em que a demanda seria menor, os resultados não se mostraram representativos. A PVM, pela limitação do tamanho da matriz e pelo tempo de execução

²Para obter os tempos utilizamos a função *gettimeofday*.

elevado do programa no mestre, não apresentou benefícios na obtenção dos resultados. Como optamos por apenas contar operações, o mesmo valor poderia ser obtido utilizando a PVM ou as implementações, pois a diferença dos programas está na “utilização dos processadores”. Nas implementações sequenciais da seção 3.8 ativamos subrotinas e na PVM executamos programas nos escravos. Estas subrotinas e programas dos escravos são basicamente os mesmos, executando a mesma sequência de operações.

3.10 Conclusão

O algoritmo da fatoração LU , ou na sua descrição menos formal e mais conhecida - o algoritmo de Gauss - ainda é amplamente utilizado na sua versão com pivotação parcial. Esta operação diminui a instabilidade deste algoritmo, evitando que utilizemos multiplicadores muito grandes em relação aos elementos de uma coluna (a coluna do pivô) que estão abaixo da diagonal principal. A eficiência deste algoritmo não é muito boa, mas é a melhor eficiência conhecida utilizando memória compartilhada. Em um modelo de memória distribuída, esta pode ser melhorada através de uma divisão de tarefas melhor.

O algoritmo de Gauss-Jordan nos fornece a solução diretamente, sem a necessidade de resolvermos um sistema triangular. No caso sequencial, ele apresenta um tempo maior que o algoritmo de Gauss. Entretanto, no caso de computadores paralelos, ele é o melhor dentre os três. Um inconveniente de sua utilização está na impossibilidade de se evitar a instabilidade, causada por multiplicadores muito grandes ou pelo aparecimento de zeros na diagonal principal, por um processo semelhante à pivotação. Por este motivo o algoritmo paralelo de Gauss com pivotação é preferencialmente utilizado em relação ao algoritmo de Gauss-Jordan. Os três algoritmos utilizam $\mathcal{O}(n^2)$ processadores, Gauss e Gauss-Jordan são algoritmos de complexidade de tempo linear e o algoritmo de Gauss com pivotação traz a solução em $\mathcal{O}(n \log n)$ passos.

A implementação destes algoritmos na PVM é basicamente a mesma das implementações sequenciais da seção 3.8, pois, ao invés de ativarmos procedimentos e funções, ativamos programas em máquinas escravas que realizam as mesmas operações dos procedimentos e funções. O programa que roda no mestre é similar ao procedimento principal das implementações sequenciais. Desta forma, ao contarmos número de operações, a escolha de uma ou outra implementação não influi no resultado. Porém, o tempo total de execução do programa no mestre acaba sendo muito superior ao da correspondente implementação sequencial da seção 3.8; assim como as comunicações se tornam ineficientes para matrizes de ordem acima de 30. A obtenção de tempos em frações de segundo se mostra imprecisa e flutuante, em ambas implementações. Por esses motivos, optamos por uma visão da PVM de caráter ilustrativo.

Capítulo 4

Fatoração QR

4.1 Introdução

Como vimos no capítulo anterior, uma maneira de melhorar a estabilidade e evitar divisões por zero no algoritmo de Gauss está em se utilizar a pivotação parcial. Este procedimento faz com que a eficiência do algoritmo caia, considerando um modelo de memória compartilhada. A decomposição QR é uma alternativa para se evitar a pivotação.

Dada uma matriz quadrada A de ordem n queremos determinar uma matriz Q ortogonal e uma matriz R triangular superior de tal forma que $A = QR$.

Neste capítulo, descreveremos alguns conceitos básicos e dois algoritmos para obter esta decomposição e suas respectivas implementações.

4.2 Matrizes Ortogonais e Rotações

Definição 21 *Uma matriz real de ordem n é ortogonal se, e somente se, sua transposta é igual à sua inversa, isto é,*

$$A \cdot A^t = A^t \cdot A = I_n.$$

As colunas de uma matriz ortogonal formam uma base ortonormal do espaço $\mathbb{R}^n$.

No $\mathbb{R}^2$, um exemplo de matrizes ortogonais é o da família das rotações planas. Dado um vetor $x = (x_1, x_2)$ do $\mathbb{R}^2$, a matriz

$$R = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

efetua uma rotação do vetor x de um ângulo θ no sentido horário.

Figura 4.1: Rotação do vetor x de um ângulo θ no sentido horário por um operador rotação R .

A rotação plana é um operador linear ortogonal do $\mathbb{R}^2$, pois

$$R \cdot R^t = \begin{bmatrix} \cos^2 \theta + \sin^2 \theta & -\cos \theta \sin \theta + \sin \theta \cos \theta \\ -\sin \theta \cos \theta + \sin \theta \cos \theta & \cos^2 \theta + \sin^2 \theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Propriedade 9 *O produto de operadores ortogonais é um operador ortogonal.*

Demonstração. Dados Q_1 e Q_2 operadores ortogonais. O resultado é imediato, basta observarmos que $(Q_1 Q_2)^t = Q_2^t Q_1^t = Q_2^{-1} Q_1^{-1} = (Q_1 Q_2)^{-1}$.

□

Podemos escolher o ângulo de tal maneira que a rotação anule a segunda componente do vetor x , teremos $\theta = -\arctan(\frac{x_2}{x_1})$ (para $x_1 = 0, \theta = \frac{\pi}{2}$).

Baseando-nos nesta idéia e na propriedade 9 podemos obter, a partir de uma matriz dada, uma matriz triangular superior, aplicando uma sequência de rotações conveniente.

4.3 Rotações de Givens

A partir do que foi descrito e considerando

$$R = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

Figura 4.2: Rotação do vetor (x_1, x_2) de um ângulo θ no sentido horário, zerando a segunda componente.

tal que $c = \cos \theta$ e $s = \sin \theta$, θ convenientemente tomado de forma a zerarmos a segunda componente de um vetor $x = (x_1, x_2)$ como na figura 4.2.

Seja $h = \sqrt{x_1^2 + x_2^2}$, a hipotenusa do triângulo ABC na figura 4.2.

Considerando o produto interno canônico, o operador rotação conserva a norma¹, assim, $x'_1 = h$. Os valores de c e s podem ser determinados resolvendo-se o sistema

$$\begin{cases} cx_1 + sx_2 = h \\ -sx_1 + cx_2 = 0 \end{cases}$$

o que nos leva a

$$c = \frac{x_1}{\sqrt{x_1^2 + x_2^2}} \text{ e } s = \frac{x_2}{\sqrt{x_1^2 + x_2^2}}.$$

A rotação de Givens é um operador linear representado pela matriz

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

¹O operador rotação é uma isometria. Maiores detalhes podem ser vistos em Caliolli et alli [8] e Hoffman e Kunze [24]

que coincide com a identidade menos em duas linhas onde teremos uma rotação.

Ao aplicarmos, convenientemente, uma seqüência de rotações de Givens a uma matriz de ordem n , poderemos obter uma matriz triangular superior ao final. A conveniência da aplicação dessas rotações está em não se alterar uma posição que já tenha sido previamente anulada por uma rotação.

Como a matriz original apresenta n linhas podemos efetuar estas operações em paralelo onde no máximo $\lfloor \frac{n}{2} \rfloor$ posições podem ser anuladas simultaneamente, uma vez que a rotação de Givens envolve duas linhas em cada processo de anulação.

A maneira como obteremos a matriz triangular superior em paralelo, através destas rotações, é descrita através de um esquema de eliminação.

4.4 Esquema de Eliminação de Sameh e Kuck

Um dos esquemas de eliminação mais fáceis de se obter é o de Sameh e Kuck [28, 40].

Podemos modificar levemente a definição de decomposição QR sem prejuízo à definição original uma vez que Q é ortogonal. Neste caso, dada uma matriz quadrada A , estamos interessados em determinar uma matriz ortogonal Q tal que $QA = R$, onde R é uma matriz triangular superior. Como foi visto, a aplicação de uma seqüência de rotações de Givens nos fornece esta matriz triangular.

Para determinarmos a solução do sistema $Ax = b$, basta resolvermos o sistema $Rx = y$ onde $y = Qb$.

Consideremos a matriz Q dada por

$$Q = Q_{n-1} \cdot Q_{n-2} \cdot \dots \cdot Q_2 \cdot Q_1$$

onde

$$Q_j = P_{j,j+1}^{(j)} \cdot \dots \cdot P_{n-2,n-1}^{(j)} \cdot P_{n-1,n}^{(j)}, 1 \leq j \leq n-1$$

e $P_{i,i+1}^{(j)}$ é uma rotação de Givens dada por

$$P_{i,i+1}^{(j)} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_i & s_i & \dots & 0 \\ 0 & 0 & \dots & -s_i & c_i & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 \end{bmatrix}$$

com a matriz de rotação imersa nas posições (i, i) , $(i, i + 1)$, $(i + 1, i)$ e $(i + 1, i + 1)$. Esta matriz realiza uma rotação nas linhas i e $i + 1$, zerando o elemento na posição $(i + 1, j)$. Como já havíamos observado, c_i e s_i são dados por

$$\begin{aligned} p_i &= \sqrt{a_{i,j}^2 + a_{i+1,j}^2} \\ c_i &= a_{i,j}/p_i \\ s_i &= a_{i+1,j}/p_i. \end{aligned}$$

Seqüencialmente, um algoritmo utilizando a descrição acima necessita aproximadamente $4n^3/3$ multiplicações e $n^2/2$ raízes quadradas.

Aproveitando a possibilidade de podermos anular posições simultaneamente, vamos descrever o esquema de eliminação de Sameh-Kuck.

*														
14	*													
13	15	*												
12	14	16	*											
11	13	15	17	*										
10	12	14	16	18	*									
9	11	13	15	17	19	*								
8	10	12	14	16	18	20	*							
7	9	11	13	15	17	19	21	*						
6	8	10	12	14	16	18	20	22	*					
5	7	9	11	13	15	17	19	21	23	*				
4	6	8	10	12	14	16	18	20	22	24	*			
3	5	7	9	11	13	15	17	19	21	23	25	*		
2	4	6	8	10	12	14	16	18	20	22	24	26	*	
1	3	5	7	9	11	13	15	17	19	21	23	25	27	*

Figura 4.3: Esquema de eliminação de Sameh e Kuck para $n = 15$.

Este esquema necessita de $l = 2n - 3$ transformações ortogonais, um exemplo para $n = 15$ é dado na figura 4.3, onde cada posição anulada pela k -ésima transformação é representado por k . A k -ésima transformação é dada por $Q_k = \prod_{i,j} P_{i,i+1}^{(j)}$ onde i e j são dados por:

i.

$$k = 1, 2, \dots, n - 1$$

$$i = \{(n - k), (n - k + 1), \dots, (n - 1 - \delta(k))\}$$

$$j = \{1, 2, \dots, \lceil k/2 \rceil\}$$

ii.

$$k = n, \dots, 2n - 3$$

$$i = \{(k - n + 2), (k - n + 4), \dots, (n - 1 - \delta(k))\}$$

$$j = \{(k - n + 2), (k - n + 3), \dots, \lceil k/2 \rceil\}$$

$$\text{onde } \delta(k) = \begin{cases} 0, & \text{se } k \text{ é ímpar.} \\ 1, & \text{se } k \text{ é par.} \end{cases}$$

O algoritmo QR usando rotações de Givens é o seguinte:

Algoritmo QR -Givens

Entrada: uma matriz A de ordem n .

Saída: uma matriz triangular superior equivalente a A .

1. Para $k = 1$ até $n - 1$

Para $i = \{(n - k), (n - k + 1), \dots, (n - 1 - \delta(k))\}$ e

$j = \{1, 2, \dots, \lceil k/2 \rceil\}$ faça em paralelo

Calcular $a_{i,j} = c_i a_{i,j} + s_i a_{i+1,j}$ e

$a_{i+1,j} = -s_i a_{i,j} + c_i a_{i+1,j}$ em paralelo

2. Para $k = n$ até $2n - 3$ faça

Para $i = \{(k - n + 2), (k - n + 4), \dots, (n - 1 - \delta(k))\}$ e

$j = \{(k - n + 2), (k - n + 3), \dots, \lceil k/2 \rceil\}$ faça em paralelo

Calcular $a_{i,j} = c_i a_{i,j} + s_i a_{i+1,j}$ e

$a_{i+1,j} = -s_i a_{i,j} + c_i a_{i+1,j}$ em paralelo

Neste algoritmo, as $l = 2n - 3$ transformações realizam um total de $n(n - 1)/2$ rotações, como no algoritmo seqüencial.

Dada $A = (a_{ij})$, ao procedermos uma rotação $P_{i,i+1}^{(j)}$ que anula a posição $(i + 1, j)$ temos que calcular p_i , c_i e s_i , sendo necessários 3 passos e 1 raiz quadrada usando 2 processadores. Os elementos das linhas i e $i + 1$ são atualizados por $P_{i,i+1}^{(j)}$ usando as expressões

$$\bar{a}_{i,j} = c_i a_{i,j} + s_i a_{i+1,j}$$

$$\bar{a}_{i+1,j} = -s_i a_{i,j} + c_i a_{i+1,j}$$

zerando a posição desejada e envolvendo 4 processadores, onde $\bar{a}_{i,j}$ e $\bar{a}_{i+1,j}$ são os valores atualizados.

Dessa forma, para cada Q_k todas as rotações são determinadas em $T_1 = 3$ passos e 1 raiz quadrada usando $p_1 = 2q$ processadores, onde q é o número de transformações em Q_k . Na seqüência de produtos de Q_k por A , se considerarmos $A_0 = A$, $A_k = Q_k A_{k-1}$ pode

ser determinado em $T_2 = 2$ passos usando $p_2 = 4 \cdot \sum_{i=1}^q (n - t)$ processadores. O tempo total para se obter Q e R é dado por

$$T_p = (2n - 3)(T_1 + T_2) = 5(2n - 3) \text{ passos.}$$

e $(2n - 3)$ raízes quadradas usando

$$p = \max\{p_1, p_2\} = 4 \cdot \sum_{i=1}^{n/2} (n - t) = n(3n - 2)/2 \text{ processadores.}$$

O algoritmo seqüencial leva tempo $T_1 = n(4n^2 - n - 3)/2$ passos e $n(n - 1)/2$ raízes quadradas. O ganho é então $S_p = \frac{T_1}{T_p} = \mathcal{O}(n^2)$ e a eficiência $E_p = \mathcal{O}(1)$.

Aplicando o algoritmo à matriz dos coeficientes estendida obtemos um sistema triangular equivalente ao original que resolvido, usando substituições recorrentes, faz com que o algoritmo todo para solucionar $Ax = b$ leve $\mathcal{O}(n)$ passos usando $\mathcal{O}(n^2)$ processadores. A eficiência deste algoritmo é $E_p \approx 1$.

4.5 Outros Esquemas de Eliminação

O esquema desenvolvido por Sameh e Kuck não é único. O esquema de Cosnard e Robert [14] dado através de um algoritmo que leva $2(n - 1) - \lfloor \log n \rfloor$ passos usando o mesmo número de processadores. É um algoritmo do tipo guloso, onde o máximo possível de elementos são anulados em cada passo. A figura 4.4 apresenta um esquema para $n = 15$ que leva 24 passos.

Este algoritmo ótimo, apesar de indicar uma leve melhora na velocidade em relação ao algoritmo de Sameh e Kuck, não é muito tratável em computadores paralelos, por causa dos cálculos complicados dos índices. Os autores concluem que, embora o algoritmo de Sameh e Kuck não seja o mais rápido, mas assintoticamente ótimo, ele é o melhor.

Em um modelo MIMD, Lord, Kowalik e Kumar [29] apresentam esquemas de eliminação com eficiência $E_p = 4/9$. Uma seqüência de tarefas concorrentes é a chave para este resultado, a descrição é semelhante ao que já vimos na seção 3.7. Dois esquemas COLSWP (*column sweep* - varredura por colunas) e ZIGZAG são apresentados e implementados em um computador paralelo HEP², usando uma variante da linguagem FORTRAN.

²HEP (*Heterogeneous Element Processor*) da Delcor, Inc.

*														
4	*													
3	6	*												
3	5	8	*											
2	5	7	10	*										
2	4	7	9	12	*									
2	4	6	9	11	14	*								
2	4	6	8	10	13	16	*							
1	3	5	8	10	12	15	18	*						
1	3	5	7	9	11	14	17	19	*					
1	3	5	7	9	11	13	16	18	20	*				
1	3	4	6	8	10	12	15	17	19	21	*			
1	2	4	6	8	10	12	14	16	18	20	22	*		
1	2	4	5	7	9	11	13	15	17	19	21	23	*	
1	2	3	5	6	8	10	12	14	16	18	20	22	24	*

Figura 4.4: Esquema de eliminação de Cosnard e Robert para $n = 15$.

4.6 Implementação do Algoritmo QR-Givens

Implementamos este algoritmo em linguagem C nas estações SUN. Os resultados de sua execução podem ser acompanhados no capítulo 6.

A seguir, temos o programa e os procedimentos utilizados.

Principal

```
#define N 300

int linha, Dm;
double A[N][N+1], X[N];

void Prod(int, int, int, int);
void Pivo1();
void solucao(int);
void impressao();

void main()
{
    int i, j, k, h, cont;
```

```

double S,Soma;

/* Entrada de dados. */
entrada();

/* Primeiras Dm-1 eliminacoes simultaneas. */
for (k=1;k<=Dm-1;k++)
{
    for (j=1;j<=(int)ceil((double)k/2)-1;j++)
        Prod(Dm-k+2*j-2,j,k,k);
    Prod(Dm-2+k%2,j,k,k);
}

/* Restante das eliminacoes simultaneas. */
for (k=Dm;k<=2*Dm-3;k++)
{
    cont = 0;
    for (j=k-Dm+2;j<=(int)ceil((double)k/2)-1;j++)
    {
        Prod(j+cont,j,k,k-Dm+1);
        cont++;
    }
    Prod(Dm-2+k%2,j,k,k-Dm+1);
}

Pivo1();

for (k = Dm; k>1; k--)
    for (linha = 1;linha<=k-1;linha++)
        solucao(k);

for (k = 1; k<=Dm;k++)
    X[k] = A[k] [Dm+1];

impressao();
}

```

O procedimento solução usado neste programa está descrito na seção 3.8.1 com uma leve modificação que é a passagem do parâmetro k que indica a posição da coluna $Dm+1$ que está sendo atualizada. Os primeiros dois argumentos do procedimento *Prod* são os índices i e j , e os outros dois são variáveis de controle para a contagem do número de

operações.

Procedimento Prod

Este procedimento elimina o elemento na posição $(i + 1, j)$ e atualiza todos as posições das linhas i e $i + 1$ afetadas pela rotação de Givens. Primeiramente são calculados p, c e s .

```
void Prod (int i,int j,int k,int h)
{
    int l;
    double p, r, c, s;

    * p=pow(A[i][j]*A[i][j]+A[i+1][j]*A[i+1][j],0.5);
    * c=A[i][j]/p;
    * s=A[i+1][j]/p;

    for (l=1;l<=Dm+1;l++)
    {
        * r = c*A[i][l] + s*A[i+1][l];
        * A[i+1][l] = -s*A[i][l]+ c*A[i+1][l];
        A[i][l] = r;
    }
}
```

Procedimento Pivo1

Este procedimento faz com que os elementos da diagonal principal sejam unitários. Os cálculos afetam somente as posições acima desta diagonal.

```
void Pivo_1()
{
    int i,j;
    double m;

    for (i=1; i<=Dm; i++)
    {
        m = A[i][i];
        A[i][i] = 1;
```

```

    for (j=i+1; j<=Dm+1; j++)
*   A[i][j] = A[i][j]/m;
  }
}

```

4.7 Decomposição de Cholesky

A fatoração QR de uma matriz A pode também ser obtida pela decomposição de Cholesky. Temos, a seguir, uma descrição de como obtermos esta decomposição e é baseada em Datta [16].

Definição 22 Dada uma matriz A positiva, a decomposição de Cholesky de A é o produto $A = LL^t$ onde L é uma matriz triangular inferior.

A decomposição de Cholesky de uma matriz A simétrica positiva de ordem n pode ser obtida pelo algoritmo descrito a seguir.

Sejam A uma matriz simétrica positiva de ordem n , $|A|$ o determinante de A .

Seja $|A_{i,i}|$ o i -ésimo menor principal de A .

Seja

$$P_{k,r} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,k-2} & a_{1,k-1} & a_{1,k} \\ a_{1,2} & a_{2,2} & \cdots & a_{2,k-2} & a_{2,k-1} & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{1,k-1} & a_{2,k-1} & \cdots & \cdots & a_{k-1,k-1} & a_{k-1,k} \\ a_{1,k+r} & a_{2,k+r} & \cdots & \cdots & a_{k-1,k+r} & a_{k,k+r} \end{pmatrix}$$

uma matriz $k \times k$, obtida da matriz $A_{k,k}$ trocando-se a k -ésima linha pelas k primeiras posições da linha $k+r$ da matriz A .

Seja $(A_{k-1,k-1})$ a submatriz obtida de $P_{k,r}$ retirando-se a k -ésima linha e a k -ésima coluna.

Definamos $|P_{1,r}| = a_{1,r+1}$ e $|P_{k,0}| = |A_{k,k}|$ onde $|A_{0,0}| = 1$.

Algoritmo - Decomposição de Cholesky

Entrada: uma matriz A simétrica positiva de ordem n .

Saída: uma matriz triangular inferior L , tal que $A = LL^t$.

1. Para $1 \leq k \leq n$ faça em paralelo

$$\text{Calcular } l_{k,k}^2 = \frac{|A_{k,k}|}{|A_{k-1,k-1}|}$$

2. Para $1 \leq k \leq n-1$ faça em paralelo
 Para $1 \leq i \leq n-k$ faça em paralelo
 Calcular $l_{k+i,k} = \frac{|P_{k,i}|}{|P_{k,0}|} \cdot l_{k,k}$

A seguir demonstraremos que o algoritmo acima realmente fornece a decomposição de Cholesky de A .

Demonstração do algoritmo

Consideremos as equações

$$l_{k,k}^2 = \frac{|A_{k,k}|}{|A_{k-1,k-1}|} \quad (4.1)$$

e

$$l_{k+i,k} = \frac{|P_{k,i}|}{|P_{k,0}|} \cdot l_{k,k} \quad (4.2)$$

- i) Como A é positiva, a decomposição de Cholesky de A será dada por

$$A = LL^t = L_1 D L_1^t \quad (4.3)$$

onde L_1 é uma matriz triangular inferior com a diagonal principal formada de uns e $D = (d_{i,i})$ é uma matriz diagonal positiva, $d_{i,i} = \frac{|A_{i,i}|}{|A_{i-1,i-1}|}$, $i = 1, \dots, n$.

Notemos que $|A| = |D| = |A_{n,n}|$ e como $|A_{i,i}|$, $i = 1, \dots, n$ são todos positivos, $D^{1/2}$ está definida.

Logo, a equação 4.3 pode ser escrita

$$A = L_1 D^{1/2} D^{1/2} L_1^t$$

e obtemos $L = L_1 D^{1/2}$ o que implica

$$l_{k,k}^2 = \frac{|A_{k,k}|}{|A_{k-1,k-1}|}, k = 1, \dots, n$$

onde $|A_{0,0}| = 1$.

- ii) Vamos provar a equação 4.2 usando indução em k .

Usando 4.1, podemos reformular 4.2 como

$$l_{k+i,k} l_{k,k} = \frac{|P_{k,i}|}{|P_{k,0}|} \cdot \frac{|P_{k,0}|}{|P_{k-1,0}|} = \frac{|P_{k,i}|}{|P_{k-1,0}|} \quad (4.4)$$

e vamos provar que ela é verdadeira.

Consideremos $L = (l_{i,j})$ e $L^t = (\bar{l}_{i,j})$, $1 \leq i, j \leq n$.

Sabemos que

$$a_{k,k+i} = \sum_{j=1}^k l_{k,j} \bar{l}_{j,k+i} = \sum_{j=1}^k l_{k+i,j} l_{k,j} = l_{k+i,k} l_{k,k} + \sum_{j=1}^{k-1} l_{k+i,j} l_{k,j}$$

donde

$$l_{k,k} l_{k+i,k} = a_{k,k+i} - \sum_{j=1}^{k-1} l_{k+i,j} l_{k,j}. \quad (4.5)$$

Para $k = 1$,

$$l_{1+i,1} l_{1,1} = a_{1,1+i}$$

De 4.4

$$l_{1+i,1} l_{1,1} = \frac{|P_{1,i}|}{|P_{0,0}|} = a_{1,1+i}$$

pela definição de $|P_{kr}|$. Logo 4.4 vale para $k = 1$.

Vamos supor que a expressão 4.4 é verdadeira para todos os valores de k de 1 a $k-1$.

De 4.1 e 4.5 podemos escrever

$$l_{k,k} l_{k,k} = \frac{|A_{k,k}|}{|A_{k-1,k-1}|} = a_{k,k} - \sum_{j=1}^{k-1} l_{k,j} l_{k,j}$$

$$|A_{k,k}| = |P_{k,0}| = a_{k,k} |P_{k-1,0}| - \sum_{j=1}^{k-1} \frac{|P_{j,k-j}| |P_{j,k-j}|}{|P_{j-1,0}| |P_{j,0}|} |P_{k-1,0}| =$$

$$= a_{k,k} |P_{k-1,0}| + a_{k-1,k} |P_{k-1,1}| + \cdots +$$

$$+ a_{1,k} \left[\frac{|P_{1,k-1}|}{|P_{0,0}| |P_{1,0}|} + c_1^{(2)} \frac{|P_{2,k-2}|}{|P_{1,0}| |P_{2,0}|} + \cdots + c_1^{(k-1)} \frac{|P_{k-1,1}|}{|P_{k-2,0}| |P_{k-1,0}|} \right] |P_{k-1,0}|$$

onde $c_j^{(h)}$ são os cofatores de $A_{h,h}$ removendo-se a h -ésima linha e a j -ésima coluna.

Como $P_{k,i}$ e $P_{k,0}$, pela construção destas matrizes, têm os mesmos cofatores em termos da última linha, temos

$$|P_{k,i}| = a_{k,k+i} |P_{k-1,0}| + a_{k-1,k+i} |P_{k-1,1}| + \cdots +$$

$$+ a_{1,k+i} \left[\frac{|P_{1,k-1}|}{|P_{0,0}| |P_{1,0}|} + c_1^{(2)} \frac{|P_{2,k-2}|}{|P_{1,0}| |P_{2,0}|} + \cdots + c_1^{(k-1)} \frac{|P_{k-1,1}|}{|P_{k-2,0}| |P_{k-1,0}|} \right] |P_{k-1,0}| =$$

$$= a_{k,k+i}|P_{k-1,0}| - \sum_{j=1}^{k-1} \frac{|P_{j,(k+i)-j}||P_{j,k-j}|}{|P_{j-1,0}||P_{j,0}|}|P_{k-1,0}|$$

ou seja,

$$\begin{aligned} \frac{|P_{k,i}|}{|P_{k-1,0}|} &= a_{k,k+i} - \sum_{j=1}^{k-1} \frac{|P_{j,(k+i)-j}||P_{j,k-j}|}{|P_{j-1,0}||P_{j,0}|} = \\ &= a_{k,k+i} - \sum_{j=1}^{k-1} l_{k+i,j}l_{k,j} = \\ &= l_{k+j,k}l_{k,k}. \end{aligned}$$

Assim 4.2 (ou 4.4) está provada para todo valor de k .

□

Os menores principais $|A_{k,k}|, k = 1, \dots, n$ de A podem ser determinados em $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^4)$ processadores usando o método de Csanky [15], que será discutido no próximo capítulo. A divisão e a raiz quadrada do passo 1 levarão mais 2 passos usando n processadores.

No passo 2 a adjunta de $P_{k,i}$ para um i particular pode ser calculado em $\mathcal{O}(\log^2 k)$ passos usando $\mathcal{O}(k^3)$ processadores pelo método de Csanky. Por este mesmo método, para calcular todas as adjuntas de $P_{k,i}$ para i particular com $k = 1, \dots, n$ levamos $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^4)$ processadores. O cálculo dos $|P_{k,i}|, k = 1, \dots, n$ e $i = 1, \dots, n - k$ leva $\mathcal{O}(\log n)$ passos usando $\mathcal{O}(n^4)$ processadores.

No total o algoritmo leva $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^4)$ processadores.

Os cálculos dos determinantes das matrizes $P_{i,j}$ podem ser realizados em $\mathcal{O}(n)$ passos, utilizando $\mathcal{O}(n^2)$ processadores. Esta alternativa melhora a estabilidade do algoritmo, pois, como veremos, o algoritmo de Csanky é muito instável. O procedimento para se obter estes determinantes pode ser visto na seção 4.9.

4.8 Fatoração QR Usando Decomposição de Cholesky

Dada uma matriz A de ordem n . Seja $B = A^t A$. B é uma matriz simétrica positiva de ordem n . Dessa forma, a decomposição de Cholesky da matriz B é dada por $B = LL^t$. Seja $Q = A(L^t)^{-1}$. Q é ortogonal, pois

$$Q^t = (A(L^t)^{-1})^t = ((L^t)^{-1})^t A^t = L^{-1} A^t$$

e

$$\begin{aligned} QQ^t &= A(L^t)^{-1}L^{-1}A^t = A(LL^t)^{-1}A^t = AB^{-1}A^t = A(A^tA)^{-1}A^t = \\ &= AA^{-1}(A^t)^{-1}A^t = I. \end{aligned}$$

A matriz R , triangular superior, é dada por $R = L^t$. Assim, $A = QR$ é a fatoração QR de A .

O algoritmo para obter esta fatoração está descrito a seguir.

Algoritmo QR -Cholesky

Entrada: uma matriz A de ordem n .

Saída: matrizes Q , ortogonal, e $R = (L^t)^{-1}$, triangular superior tais que $A = QR$.

1. Calcular $B = A^tA$ em paralelo
2. Calcular L e L^t em paralelo, usando a decomposição de Cholesky tal que $B = LL^t$
3. Calcular $(L^t)^{-1}$ em paralelo
4. Calcular $Q = A(L^t)^{-1}$ em paralelo

A solução do sistema pode ser dada calculando-se o produto $Q^tb = y$ e resolvendo-se o sistema triangular superior $Rx = y$.

No passo 1 precisamos de $\mathcal{O}(\log n)$ passos e n^3 processadores (seção 1.10). No passo 2, $\mathcal{O}(\log^2 n)$ passos e $\mathcal{O}(n^4)$ processadores. Em 3, $\mathcal{O}(\log^2 n)$ passos e $\mathcal{O}(n^3)$ processadores [5, 28] e no passo 4 $\mathcal{O}(\log n)$ passos com n^3 processadores. O produto $Q^tb = y$ pode ser feito em $\mathcal{O}(\log n)$ passos usando $\mathcal{O}(n^2)$ processadores (seção 1.10). Finalmente o sistema triangular superior $Rx = y$ pode ser solucionado em $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n)$ processadores (seção 2.4.2).

Dessa forma, no total precisaremos de $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^4)$ processadores.

Uma vez que o método de Csanky é altamente instável[23], nos cálculos que envolvam o método de Csanky podemos usar o algoritmo de eliminação de Gauss. Com esta modificação, o algoritmo da fatoração QR levará no total $\mathcal{O}(n)$ passos usando $\mathcal{O}(n^2)$ processadores, uma vez que cada um dos passos acima leva este tempo com o mesmo número de processadores. Isto é feito utilizando uma variação do algoritmo de Gauss-Jordan para a inversão da matriz L^t no passo 3 e uma variação do algoritmo de Gauss para determinar $|P_{i,j}|$ necessário no passo 2.

Para resolver o sistema triangular superior $Rx = y$ levamos $\mathcal{O}(n)$ passos utilizando $\mathcal{O}(n)$ processadores (seção 2.4.1).

Dessa forma, a solução do sistema é obtida em $\mathcal{O}(n)$ passos utilizando $\mathcal{O}(n^2)$ processadores.

Uma vez que $T_1(n) = \mathcal{O}(n^3)$, o ganho é $S_p(n) = \mathcal{O}(n^2)$ e a eficiência $E_p(n) \approx 1$.

4.9 Implementação do Algoritmo QR -Cholesky

A implementação deste algoritmo em linguagem C é dada a seguir. Foram construídos procedimentos e funções dos algoritmos mais específicos. Aqueles trechos que envolviam apenas cálculos com matrizes foram mantidos no procedimento principal.

Principal

```
void main()
{
    int i,j,k;
    double S,Soma;

    /* Entrada de dados. */
    entrada();

    /* Calculo de B. */
    for (i=1;i<=Dm;i++)
        for (j=1;j<=Dm;j++)
        {
            S=0.0;
            for (k=1;k<=Dm;k++)
            *   S = S + A[k][i] * A[k][j];
            B[i][j] = S;
        }

    /* Calculo dos determinantes das matrizes Pij. */
    P[0][0] = 1.0;
    for (k=1;k<=Dm; k++)
        P[1][k-1] = B[1][k];

    for (k=2; k<=Dm; k++)
    {
        LU(k);
    *   P[k][0] = P[k-1][0]*B[k][k];

        for (i=1;i<=Dm;i++)
    *       P[k][i] = P[k-1][0]*B[k+i][k];
    }
}
```

```

/* Determinacao da matriz L-transposta */
for (k=1;k<=Dm;k++)
*   R[k][k]=L[k][k] = pow((P[k][0]/P[k-1][0]),0.5);

    for (k=1;k<=Dm-1;k++)
        for (i=1;i<=Dm-k;i++)
*   R[k][k+i]=L[k][k+i] = P[k][i]/P[k][0]*L[k][k];

    Inv();
    Prod();

/* Calculo de y=Qb*/
for (i=1;i<=Dm;i++)
{
    Soma = 0.0;
    for (k=1;k<=Dm;k++)
*   Soma=Soma+(Q[k][i]*A[k][Dm+1]);
    R[i][Dm+1] = Soma;
}

    trisup();

    for (i=1;i<=Dm;i++)
        X[i] = R[i][Dm+1];

    impressao();
}

```

O cálculo da matriz B foi feita da maneira clássica (sem utilizar o algoritmo *fan-in* associativo), pois queremos preservar o número de processadores em $\mathcal{O}(n^2)$. O cálculo dos determinantes das matrizes P_{ij} foi feito utilizando-se uma variante do algoritmo de Gauss, fazemos uma chamada ao procedimento LU . A partir da determinação destes P_{ij} calculamos a matriz L^t e utilizamos, para isso, a variável L . O produto $y = Qb$ também é determinado da maneira clássica.

Procedimento LU

```

void LU(int c)
{
    int i,j;

```

```

double m;

for (i=c; i<=Dm;i++)
{
    m = B[i][c-1];
    for (j=c;j<=Dm;j++)
        B[i][j] = B[i][j] - m/B[c-1][c-1]*B[c-1][j];
}
}

```

Ao triangularizarmos a matriz B , podemos obter o determinante das submatrizes $B_{kk}, k = 1, \dots, n$ fazendo o produto dos elementos da diagonal principal. Ao final de cada execução deste procedimento, multiplicamos o novo valor de B_{kk} pelo determinante da submatriz anterior.

Dada uma matriz A e seja E a matriz elementar tal que $T = EA$ seja triangular superior. Nesta triangularização mantemos a diagonal principal intacta; com isto, a matriz E é uma matriz triangular inferior com a diagonal formada apenas de elementos unitários, logo $|E| = 1$. Como $|T| = |EA| = |E||A| = 1 \cdot |A|$, mostramos a validade do procedimento.

Procedimento Inv

```

void Inv()
{
    int i,j, row;
    double m;

    for (i=1;i<=Dm; i++)
        LI[i][i] = 1.0;

    for (row=Dm;row>1;row--)
        for (i=row-1; i>=1; i--)
        {
            m = L[i][row];
            for (j=row;j<=Dm;j++)
            {
*          L[i][j] = L[i][j] - m/L[row][row]*L[row][j];
*          LI[i][j] = LI[i][j] - m/L[row][row]*LI[row][j];
            }
        }
}

```

```

for (i=1;i<=Dm;i++)
{
    m = L[i][i];
    for (j=i; j<=Dm;j++)
*   LI[i][j] = LI[i][j]/m;

}
}

```

Este algoritmo inverte a matriz L^t utilizando-se a idéia do algoritmo de Gauss-Jordan. Obtemos uma matriz identidade a partir da matriz L^t com este algoritmo utilizando operações elementares sobre as linhas desta matriz. Estas operações são estendidas a uma matriz identidade. Ao final, obteremos, da modificação desta matriz identidade, a matriz $(L^t)^{-1}$. Outros detalhes do método empregado podem ser obtidos em Caliolli et alli [8] e Hoffman e Kunze [24].

Procedimento Prod

```

void Prod()
{
    int i,j,k;
    double S;

    for (i=1;i<=Dm;i++)
        for (j=1;j<=Dm;j++)
        {
            S=0;
            for (k=1;k<=Dm;k++)
*       S = S + A[i][k] * LI[k][j];
            Q[i][j] = S;
        }
}

```

Este procedimento obtém a matriz $Q = A(L^t)^{-1}$.

Procedimento trisup

```

void trisup()
{

```

```

int i,k,linha;
double m;

for (i=1;i<=Dm;i++)
{
    m = L[i][i];
    R[i][i] = 1.0;
    for (j=i+1;j<=Dm+1;j++)
*   R[i][j] = R[i][j]/m;
}
    for (k=Dm;k>1;k--)
        for (linha=1;linha<=k-1;linha++)
*   R[linha][Dm+1] = R[linha][Dm+1]-R[linha][k]*R[k][Dm+1];
}

```

Para determinarmos a solução do sistema triangular superior $Rx = y$ utilizamos o algoritmo de substituições recorrentes.

4.10 Conclusão

Neste capítulo, apresentamos algoritmos que utilizam a fatoração de uma matriz A na forma de um produto $A = QR$, onde Q é uma matriz ortogonal e R é uma matriz triangular superior. Este tipo de fatoração é uma alternativa à fatoração LU , evitando a pivotação parcial do algoritmo de Gauss.

O algoritmo QR -Givens é um algoritmo simples que utiliza rotações planas para se obter uma matriz triangular superior equivalente à matriz dos coeficientes.

O algoritmo QR -Cholesky determina as matrizes Q e R a partir da decomposição de Cholesky de uma matriz $B = A^t A$. Este algoritmo é mais elaborado que o algoritmo QR -Givens, podendo ter complexidade $\mathcal{O}(\log^2 n)$ utilizando $\mathcal{O}(n^4)$ processadores, que é um resultado teórico muito importante, mas que é numericamente instável por utilizar o método de Csanky, que estudaremos detalhadamente no próximo capítulo. Na implementação deste algoritmo utilizamos uma versão que traz a solução em $\mathcal{O}(n)$ passos, utilizando $\mathcal{O}(n^2)$ processadores, diminuindo esta instabilidade.

Estes algoritmos utilizam raízes quadradas, que são operações complexas. Existem versões para o algoritmo QR -Givens que utilizam rotações livres de raízes, que têm uma manipulação de índices que dificulta a implementação em computadores paralelos.

Capítulo 5

Algoritmo de Csanky

5.1 Introdução

Os algoritmos que vimos até agora envolviam a determinação de matrizes triangulares superiores de forma que o sistema obtido fosse equivalente ao sistema original.

Uma outra forma de se resolver um sistema de equações lineares é através da determinação da inversa da matriz dos coeficientes.

Descreveremos neste capítulo um algoritmo paralelo mais rápido para a determinação da inversa de uma matriz dada. O algoritmo deriva imediatamente do método de Leverrier. Com este algoritmo conseguiu-se resolver um sistema de equações lineares em tempo logarítmico polinomial, incluindo este problema e a inversão de matrizes na classe NC^2 . Algumas referências são encontradas em [5, 15, 23, 28].

5.2 Autovalores e Autovetores

Consideremos T um operador linear sobre um espaço vetorial de dimensão finita (sobre $\mathbb{R}$ ou $\mathbb{C}$).

Definição 23 Um autovalor de T é um escalar λ (em $\mathbb{R}$ ou $\mathbb{C}$) tal que exista um vetor não nulo $u \in V$ tal que $T(u) = \lambda u$. Se λ é um autovalor de T , então todo u em V tal que $T(u) = \lambda u$ é denominado **autovetor** de T associado ao autovalor λ e ao conjunto de todos os autovetores de T associados a um autovalor λ é denominado **autoespaço** associado a λ .

Teorema 3 Seja T um operador linear sobre um espaço V de dimensão n e seja λ um escalar. As seguintes afirmações são equivalentes:

- i) λ é um autovalor de T .
- ii) O operador $(\lambda I - T)$ é singular (não-inversível).
- iii) $\det(\lambda I - T) = 0$.

Dada uma base de V , a cada operador linear podemos associar uma matriz A então $(\lambda I - T)$ é inversível se, e somente se, $(\lambda I - A)$ é inversível. Dessa forma obtemos a definição que se segue.

Definição 24 *Seja A uma matriz de ordem n , um autovalor de A é um escalar λ tal que a matriz $(\lambda I - A)$ seja singular (não inversível).*

O teorema e a definição nos dão uma maneira de obtermos autovalores de uma matriz A . Um escalar λ é um autovalor de A se, e somente se, $\det(\lambda I - A) = 0$. Assim, poderemos obter uma matriz $(xI - A)$ onde seus elementos são polinômios e considerar o polinômio $\phi(x) = \det(xI - A)$. Os autovalores de A são as raízes deste polinômio, que é denominado **polinômio característico** de A .

Podemos escrever este polinômio na forma

$$\phi(\lambda) = (-1)^n \cdot (\lambda^n + p_1 \lambda^{n-1} + p_2 \lambda^{n-2} + \cdots + p_{n-1} \lambda + p_n). \quad (5.1)$$

Temos a seguinte propriedade:

Propriedade 10 *Considerando a equação 5.1, temos:*

$$p_1 = -\sum_{i=1}^n \lambda_i = -\text{tr}(A)$$

$$p_k = (-1)^k \times \text{soma dos menores principais de ordem } k, 2 \leq k \leq n-1$$

$$p_n = (-1)^n \det(A)$$

Estamos interessados em autovalores e polinômios característicos de matrizes. Os resultados a seguir são enunciados para operadores lineares; mas, como a cada matriz está associado um operador linear e vice-versa, enunciaremos em relação a matrizes. A demonstração destes resultados pode ser obtida em Hoffman e Kunze [24].

A propriedade a seguir é bastante simples de ser demonstrada mas bastante importante para a obtenção do algoritmo de Csanky.

Propriedade 11 *Sejam $\lambda_i, i = 1, 2, \dots, n$ os autovalores de uma matriz A e $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_mx^m$ um polinômio. Então $f(\lambda_i), i = 1, 2, \dots, n$ são autovalores da matriz $f(A)$. Em particular, $\lambda_i^k, i = 1, 2, \dots, n$, são autovalores de $A^k, k \geq 1$. [28]*

O teorema de Cayley-Hamilton a seguir é importante na determinação do polinômio minimal, que é útil na diagonalização de operadores. O resultado é simples mas de extrema importância e afirma que dada uma matriz A ela é raiz de seu próprio polinômio característico.

Teorema 4 (Cayley-Hamilton). *Seja A uma matriz de ordem n . Se ϕ é o polinômio característico de A , então $\phi(A) = 0$.*

5.3 Método de Leverrier

Seja A uma matriz $n \times n$ não singular, consideremos o polinômio característico ϕ de A definido por

$$\phi(\lambda) = \det(A - \lambda I) = (-1)^n \cdot \prod_{i=1}^n (\lambda - \lambda_i) \quad (5.2)$$

onde $\lambda_1, \dots, \lambda_n$ são os autovalores da matriz A . Sejam $p_1, \dots, p_n$ os coeficientes do polinômio característico 5.1.

Pelas propriedades 11 e 10, podemos definir, para $1 \leq k \leq n$

$$t_k = \sum_{i=1}^n \lambda_i^k = \text{tr}(A^k). \quad (5.3)$$

A derivada do polinômio característico em 5.1 é dada por

$$\frac{d\phi(\lambda)}{d\lambda} = n\lambda^{n-1} + p_1(n-1)\lambda^{n-2} + \dots + p_{n-1}. \quad (5.4)$$

Por outro lado derivando-se ambos os lados da equação 5.2

$$\frac{d\phi(\lambda)}{d\lambda} = \phi(\lambda) \sum_{i=1}^n (\lambda - \lambda_i)^{-1}. \quad (5.5)$$

Comparando os coeficientes das equações 5.4 e 5.5 e usando a definição de t_k em 5.3, obtemos a seguinte relação de recorrência definindo t_k .

$$t_1 + p_1 = 0$$

e, para $2 \leq k \leq n$,

$$t_k + p_1 t_{k-1} + p_2 t_{k-2} + \dots + p_{k-1} t_1 + k p_k = 0.$$

Esta fórmula é conhecida como *fórmula de Newton*. A partir disto, os coeficientes do polinômio característico podem ser obtidos solucionando-se o sistema triangular inferior

$$Tp = -t$$

onde

$$p = (p_1, p_2, \dots, p_n)^t$$

$$t = (t_1, t_2, \dots, t_n)^t$$

e

$$T = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ t_1 & 2 & 0 & \cdots & 0 & 0 \\ t_2 & t_1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ t_{n-1} & t_{n-2} & t_{n-3} & \cdots & t_1 & n \end{pmatrix}.$$

Multiplicando-se ambos os lados da equação $\phi(A) = 0$ por A^{-1} obtemos

$$A^{-1}A^n + p_1A^{-1}A^{n-1} + p_2A^{-1}A^{n-2} + \cdots + p_{n-1}A^{-1}A + p_nA^{-1} = 0,$$

e daí segue-se que

$$A^{-1} = -\frac{A^{n-1} + p_1A^{n-2} + p_2A^{n-3} + \cdots + p_{n-1}I}{p_n}. \quad (5.6)$$

5.4 Algoritmo de Csanky

O método de Leverrier descrito na seção anterior motiva o algoritmo paralelo de Csanky para inversão de matrizes.

A obtenção das potências da matriz A na equação 5.6 e a obtenção dos traços destas matrizes, necessários à determinação de $p_i, i = 1, \dots, n$, pode ser feita em tempo logarítmico polinomial. Utilizando o algoritmo de Sameh e Brent [38] resolvemos o sistema $Tp = -t$. Dessa forma, obtemos a inversa de A . O algoritmo é dado por:

Algoritmo de Csanky

Entrada: uma matriz A de ordem n .

Saída: a inversa da matriz A .

1. Para $1 \leq j \leq n$ faça em paralelo

Calcular A^j em paralelo

Calcular t_j em paralelo

2. Calcular p em paralelo, resolvendo-se o sistema triangular inferior $Tp = -t$

3. Calcular A^{-1} em paralelo utilizando a fórmula 5.6

No Passo 1 os $A^j, j = 1, \dots, n$, podem ser calculados em paralelo em $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^4)$ processadores (seção 2.5.4). Tendo A^j disponível, t_j pode ser obtido em paralelo em $\mathcal{O}(\log n)$ passos usando $\mathcal{O}(n^2)$ processadores, usando o algoritmo *fan-in* associativo (seção 1.10). Os coeficientes do polinômio característico podem ser obtidos em $\mathcal{O}(\log^2 n)$ passos usando $\mathcal{O}(n^3)$ processadores através do algoritmo descrito em 2.4.2.

No passo 3, $p_j A^{n-j}$ pode ser calculado em 1 passo usando n^2 processadores. Dessa forma, todos os fatores individuais podem ser calculados em 1 passo usando $\mathcal{O}(n^3)$ processadores. O numerador pode ser calculado (usando o algoritmo associativo *fan-in*) em $\mathcal{O}(\log n)$ passos usando $\mathcal{O}(n^3)$ processadores. Finalmente, A^{-1} pode ser obtida com apenas mais uma divisão usando n^2 processadores.

Assim todo o algoritmo leva

$T(n) = \mathcal{O}(\log^2 n)$ passos utilizando $\mathcal{O}(n^4)$ processadores.

Depois de calcularmos A^{-1} , a solução do sistema $Ax = b$ é dado por $x = A^{-1}b$, este produto leva tempo $\mathcal{O}(\log n)$ utilizando $\mathcal{O}(n^2)$ processadores (2.5.2).

Dessa forma, obtemos um algoritmo para a solução de sistemas lineares que leva tempo $\mathcal{O}(\log^2 n)$ envolvendo $\mathcal{O}(n^4)$ processadores.

O ganho deste algoritmo é dado por $S_p(n) = \frac{\mathcal{O}(n^3)}{\mathcal{O}(\log^2 n)}$ e a eficiência é $E_p(n) = \mathcal{O}(\frac{1}{n \log^2 n})$ que tende a zero com o crescimento de n .

Como o cálculo do determinante de A depende de todos os n^2 elementos de A , usando os argumentos padrão do *fan-in* do teorema 1, segue-se que

$$T(n) \geq 2 \log n.$$

Isto é, o tempo de complexidade em paralelo é tal que

$$\log n \leq T(n) \leq \log^2 n.$$

Assim, obtemos uma distância menor em relação à cota inferior para este tipo de problema. Além disso, conseguimos a inclusão dos problemas de inversão de matrizes e solução de sistemas lineares na classe NC^2 .

Este algoritmo é dito ser o mais rápido algoritmo para inversão de matrizes. Apesar disto, na prática, temos problemas quanto ao número elevado de processadores; mesmo que asseguremos este número grande de processadores, este é um algoritmo numericamente instável.

5.5 Implementação

A implementação do algoritmo de Csanky na linguagem C está descrito a seguir.

Principal

```
#define N 80

int Dm, Aux3, Aux;

double A[N+1][N+1][N+1], B[N], p[N+1], T[N][N+1][N+1],
      Taux[N+1][N+1], Inv[N+1][N+1], t[N+1], X[N+1];

void Potencias();
void Tracos();
void resolve_tri_inf();
void Prod(int,int);
void Interno(int);
void Calculo_inversa();
void Vetor_X();
double Soma(double *,int,int,int,int,int);

void main()
{
    Potencias();
    Tracos();
    Resolve_tri_inf();
    Calculo_inversa();
    Vetor_X();

    impressao();
}
```

O procedimento principal simplesmente ativa os procedimentos *Potencias*, *Tracos*, *Resolve_tri_inf*, *Calculo_inversa*, *Vetor_X* e *impressao*.

A variável $A[i][j][k]$ representa uma matriz tridimensional que armazena as potências da matriz A lida. O primeiro índice representa esta potência e os outros dois a posição

do elemento.

Na variável Dm é armazenada a ordem da matriz. A variável $t[\cdot]$ armazena os traços e $T[i][\cdot][\cdot]$ as matrizes $A_i^{(j)}$ utilizadas no algoritmo de Sameh e Brent [38]. O vetor $p[\cdot]$ armazena os coeficientes do polinômio característico de A .

A variável $B[\cdot]$ armazena o vetor b do sistema de equações. A variável $Taux[\cdot][\cdot]$ é uma matriz auxiliar na solução do sistema triangular inferior $Tp = -t$. A variável $Inv[\cdot][\cdot]$ armazena a inversa da matriz A e $X[\cdot]$ armazena a solução do sistema.

Procedimento Potencias

```
void Potencias ()
{
    int i, Aux1, Aux2, lim, g, c, j, h, k, l;
    double S[N];

    Aux = (int)ceil(log(Dm)/log(2.0));

    /* Calcula as potencias de A. */
    for (i=1; i<= Aux; i++)
    {
        Aux1 = lim = (int)pow(2,i);
        if (Dm<lim)
        {
            lim=Dm;
        }
        c = 0;

        for (j=(int)floor((float)Aux1/2)+1; j<=lim; j++)
        {
            c++;

            for (k=1; k<=Dm;k++)
            {
                for (l=1;l<=Dm;l++)
                {
                    for (h=1; h<=floor((float)Dm/2); h++)
                    *
                        S[h] = A[Aux1/2][k][2*h-1]*A[c][2*h-1][1]+
                            A[Aux1/2][k][2*h]*A[c][2*h][1];
                    if (Dm%2)
                    {
```

```

*      S[h] = A[Aux1/2][k][Dm]*A[c][Dm][1];
      if (h%2)
        S[h+1] = 0;
      }
      else
        S[h]=0.0;
      A[j][k][1] = Soma(S,j,k,1,Dm,Aux);
    }
  }
}
}

```

Este procedimento é responsável pelo cálculo das potências da matriz A . Na variável Aux armazenamos o valor de $\lceil \log Dm \rceil$. O contador i é responsável pelo número de iterações que são concorrentes, gerando $Aux1$ que é 2^i . O contador j controla as potências que estão sendo calculadas simultaneamente. Os contadores k e l controlam os índices de linhas e colunas, respectivamente, da potência que está sendo calculada.

No vetor $S[\cdot]$ armazenamos as $\lfloor \frac{Dm}{2} \rfloor$ posições do produto interno da linha k da matriz $A_{\frac{Aux1}{2}}$ pela coluna l da matriz A^c como no algoritmo da seção 2.5.1. O controle dos índices é feito pela variável h . A função da variável c pode ser vista na descrição do algoritmo da seção 2.5.4. Logo após o laço da variável h , testamos a necessidade de mais uma posição (no caso de Dm ímpar), que se necessário é preenchida.

Concluída a inicialização da variável $S[\cdot]$, ativamos a função $Soma$ que finalizará a soma dos elementos de $S[\cdot]$. No retorno desta função teremos o valor da posição (k, l) da matriz A^j .

Procedimento Tracos

```

void Tracos()
{
  int h,j;
  double S[N];

  for (j=1;j<=Dm;j++)
  {
    for (h=1; h<=floor((float)Dm/2); h++)
*      S[h] = A[j][2*h-1][2*h-1] + A[j][2*h][2*h];

    if (Dm%2)

```

```

{
    S[h] = A[j][Dm][Dm];
    if (h%2)
        S[h+1] = 0;
}
else
    S[h]=0.0;
t[j] = Soma(S,1,1,j,Dm,Aux);
}
}

```

Este procedimento calcula os traços das potências da matriz A simultaneamente. O laço *for j* controla as potências da matriz A .

A variável $S[\cdot]$ armazena as somas dos pares de posições da diagonal principal como descrito na seção 1.10, controlado pela variável h . Após este laço, novamente testamos a necessidade de mais uma posição (no caso de Dm ímpar), isto acontecerá todas as vezes que inicializarmos o vetor $S[\cdot]$. Ativamos a função *Soma* e no retorno desta, teremos o traço de A^j .

Procedimento `Resolve_tri_inf`

```

void Resolve_tri_inf()
{
    int i,j,k,g,h,Aux2;

    Aux3 = Aux-1;

    for (i=1;i<=Dm;i++)
*   p[i] = -t[i]/i;

    for (k=1;k<=Dm;k++)
    {
        for (j=1;j<=Dm;j++)
            T[k][j][j]=1.0;
        for(i=k+1;i<=Dm;i++)
*       T[k][i][k] = -t[i-k]/i;
    }

    for(i=1;i<=Dm;i++)
        for (j=1;j<=Dm;j++)

```

```

    Taux[i][j] = T[1][i][j];

for (h=2;h<=floor((float)Dm/2);h++)
    Prod(h-1,1);
Interno(1);

if (Dm%2)
{
    for(i=1;i<=Dm;i++)
        for(j=1;j<=Dm;j++)
            T[h-1][i][j] = T[2*(h-1)][i][j];
    if (h%2)
    {
        for(i=1;i<=Dm;i++)
        {
            for(j=1;j<=Dm;j++)
                T[h][i][j] = 0.0;
            T[h][i][i] = 1.0;
        }
    }
}
Aux2 = 2;
for(h=2;h<=Aux3;h++)
{
    for(i=1;i<=Dm;i++)
        for (j=1;j<=Dm;j++)
            Taux[i][j] = T[1][i][j];

    Aux2 = Aux2*2;
    for (g=2;g<=ceil((float)Dm/Aux2);g++)
        Prod(g-1,Aux2/2);
    Interno(Aux2/2);

    for(i=1;i<=Dm;i++)
    {
        for(j=1;j<=Dm;j++)
            T[g-1][i][j] = 0.0;
        T[g-1][i][i] = 1.0;
    }
}

```

```

for(i=1;i<=Dm;i++)
  for (j=1;j<=Dm;j++)
    Taux[i][j] = T[1][i][j];
Interno(Aux2);
}

```

Este procedimento resolve o sistema triangular inferior $Tp = -t$ utilizando o algoritmo de Sameh e Brent [38] descrito na seção 2.4.2. Ele ativa os procedimentos *Prod* e *Interno*, que serão descritos em seguida.

A solução do sistema será o vetor p armazenado na variável de mesmo nome. Inicialmente, inicializamos a matriz $T[\cdot][\cdot][\cdot]$, tomando o cuidado de dividirmos cada linha de $T[\cdot][\cdot][\cdot]$ por i para que a diagonal principal seja unitária. Essas divisões são estendidas ao vetor de termos independentes (vetor $-t$) e são armazenados no vetor p , $p[i] = -t[i]/i$, $i = 1, \dots, Dm$, inicializando-o. Uma variável bidimensional $Taux[\cdot][\cdot]$ armazenas as matrizes A_1^k do algoritmo.

Utilizamos a idéia do algoritmo *fan-in* associativo para fazer os produtos das matrizes $T[\cdot][\cdot][\cdot]$ aos pares utilizando o procedimento *Prod* e atualizamos o vetor de termos independentes ativando o procedimento *Interno*.

Procedimento Prod

```

void Prod(int k,int iter)
{
  int i,j,h,g,linf1,linf2,tam,Aux;
  double S[N];

  linf1=2*iter*k;
  linf2=linf1+iter;
  tam = iter;

  if (linf2 >= Dm)
  {
    for (i=1;i<=Dm;i++)
      for (j=1;j<=Dm;j++)
        T[k][i][j]=T[2*k][i][j];
  }
  else
  {
    /* Calcula Li+1 Ui */
    for (i=linf2;i<=linf2+tam-1;i++)

```



```

{
    for(j=linf1;j<=linf1+tam-1;j++)
    {
        for (h=1;h<=floor((float)(i-linf2+1)/2);h++)
        *   S[h] = T[2*k+1][i][linf2+2*h-2]*T[2*k][linf2+2*h-2][j]+
            T[2*k+1][i][linf2+2*h-1]*T[2*k][linf2+2*h-1][j];

        if ((i-linf2+1)%2)
        {
            *   S[h]=T[2*k+1][i][i]*T[2*k][i][j];
                if (h%2)
                    S[h+1] = 0.0;
        }
        else
            S[h]=0.0;
        Aux = (int)ceil(log(i-linf2+1)/log(2.0));
        T[k][i][j] = Soma(S,k,i-linf2+1,j-linf1+1,i-linf2+1,Aux);
    }
}

/* Calcula Si+1 Ui */
for (i=linf2+tam;i<=Dm;i++)
{
    for(j=linf1;j<=linf1+tam-1;j++)
    {
        for (h=1;h<=floor((float)tam/2);h++)
        *   S[h] = T[2*k+1][i][linf2+2*h-2]*T[2*k][linf2+2*h-2][j]+
            T[2*k+1][i][linf2+2*h-1]*T[2*k][linf2+2*h-1][j];

        if ((tam)%2)
        {
            S[h]=T[2*k+1][i][linf2+tam-1]*T[2*k][linf2+tam-1][j];
            if (h%2)
                S[h+1] = 0.0;
        }
        else
            S[h]=0.0;
        Aux = (int)ceil(log(tam)/log(2.0));
        T[k][i][j] = Soma(S,k,i-linf2+1,j-linf1+1,tam,Aux);
    }
}

```

```

for(i=linf2+tam;i<=Dm;i++)
{
    for(j=linf1;j<=linf1+tam-1;j++) /* Calcula Si+1 Ui + Vi. */
*   T[k][i][j] = T[k][i][j] + T[2*k][i][j];

    for (j=linf2;j<=linf2+tam-1;j++) /* Copia Si+1. */
        T[k][i][j] = T[2*k+1][i][j];
}
for (i=linf2;i<=linf2+tam-1;i++) /* Copia Li+1. */
    for (j=linf2;j<i;j++)
        T[k][i][j] = T[2*k+1][i][j];

for (i=linf1;i<=linf1+tam-1;i++) /* Copia Li. */
    for (j=linf1;j<i;j++)
        T[k][i][j] = T[2*k][i][j];

for(j=1;j<linf1;j++)
{
    for(i=1;i<=Dm;i++)
        T[k][i][j]=0.0;
    T[k][j][j] = 1.0;
}
}
}

```

O procedimento *Prod* recebe os argumentos k e $inter$, estes argumentos são os índices da matriz $A_k^{(iter)}$. Como estas matrizes $T[\cdot][\cdot][\cdot]$ são matrizes esparsas cujos blocos são bem definidos e alguns são matrizes triangulares inferiores, podemos nos beneficiar disto, economizando operações e processadores.

As variáveis $linf1$ e $linf2$ armazenam as posições a partir de onde as matrizes apresentam os blocos que serão envolvidos nos cálculos, a variável tam armazena o tamanho de tais blocos. A descrição de como são os blocos e o significado das variáveis $linf1$ e $linf2$ podem ser obtidas na seção 2.4.2. No caso de $linf2 \geq Dm$, apenas copiamos a matriz $T[2 * k][\cdot][\cdot]$ em $T[k][\cdot][\cdot]$.

Assim, o programa calcula o produto $L_{i+1}U_i$ e o produto $S_{i+1}U_i$, que são feitos em paralelo, mais uma vez utilizamos o algoritmo da seção 2.5.3.

Após isto, calcula-se $S_{i+1}U_i + V_i$, utilizando $S_{i+1}U_i$ já calculado e fazemos a cópia de S_{i+1} , L_{i+1} e L_i para a nova matriz $A_k^{(iter)}$.

Procedimento Interno

```

void Interno(int iter)
{
    int i,j,h,g,Aux,lsup1,lsup2;
    double d[N+1], S[N];

    lsup1 = 2*iter-1;
    lsup2 = iter;
    if (lsup1 >= Dm)
    {
        lsup1=Dm;
        lsup2=Dm-iter+1;
    }

    /* Calcula L1g2.*/
    for(i=iter;i<=lsup1;i++)
    {
        for(h=1;h<=floor((float)(i-iter+1)/2);h++)
        *   S[h] = Taux[i][iter+2*h-2]*p[iter+2*h-2]+
            Taux[i][iter+2*h-1]*p[iter+2*h-1];

        if ((i-iter+1)%2)
        {
            *   S[h] = Taux[i][i]*p[i];
                if (h%2)
                    S[h+1] = 0.0;
        }
        else
            S[h]=0.0;
        Aux = (int)ceil(log(i-iter+1)/log(2.0));
        d[i] = Soma(S,1,1,lsup1-i+1,i-iter+1,Aux);
    }

    /* Calcula S1g2. */
    for(i=lsup1+1;i<=Dm;i++)
    {
        for(h=1;h<=floor((float)lsup2/2);h++)
        *   S[h] = Taux[i][iter+2*h-2]*p[iter+2*h-2]+
            Taux[i][iter+2*h-1]*p[iter+2*h-1];
    }

```

```

        if (lsup2%2)
        {
*      S[h] = Taux[i][iter]*p[iter];
        if (h%2)
            S[h+1] = 0.0;
        }
        else
            S[h]=0.0;
        Aux = (int)ceil(log(lsup2)/log(2.0));
        d[i] = Soma(S,1,1,Dm-i+lsup1-iter+2,lsup2,Aux);
    }

    for (i=iter;i<=lsup1;i++)
        p[i] = d[i];

    for (i=lsup1+1;i<=Dm;i++) /* Calcula S1g2+g3. */
*   p[i] = d[i]+p[i];
}

```

O procedimento *Interno* se comporta de maneira análoga ao procedimento anterior. Calcula-se L_1g_2 e S_1g_2 em paralelo, utilizando o algoritmo da seção 2.5.1, armazenando os resultados no vetor d . Copiamos o vetor g_1 para o novo vetor e calculamos $S_1g_2 + g_3$ atualizando o subvetor g_3 do vetor p .

Procedimento Calculo_inversa

```

void Calculo_inversa()
{
    int k, row, g, h, i, j, Aux2;
    double S[N];

    p[0] = 1;

    for (i=1;i<=Dm;i++)
        for (j=1;j<=Dm;j++)
        {
            for (h=1; h<= floor((float)Dm/2); h++)
*      S[h] = A[2*h-2][i][j]*p[Dm-2*h+1] +
            A[2*h-1][i][j]*p[Dm-2*h];
            if (Dm%2)

```

```

    {
        S[h] = A[Dm-1][i][j];
        if (h%2)
            S[h+1] = 0;
    }
    else
        S[h]=0.0;
    Inv[i][j] = -Soma(S,1,i,j,Dm,Aux);
}

for (i=1;i<=Dm;i++)
    for (j=1;j<=Dm;j++)
*   Inv[i][j]=Inv[i][j]/p[Dm];
}

```

Este algoritmo calcula a inversa da matriz A através da fórmula 5.6. O produto $A_i p_{Dm-i+1}$ é calculado inicialmente, em seguida calcula-se a soma destes produtos. Com mais uma divisão conclui-se este procedimento com o resultado armazenado na variável $Inv[\cdot][\cdot]$.

Procedimento Vetor_X

```

void Vetor_X()
{
    int i, h, g, Aux2;
    double S[N];

    for (i=1;i<=Dm;i++)
    {
        for (h=1; h<= (int)floor((float)Dm/2); h++)
*   S[h] = Inv[i][2*h-1]*B[2*h-1] +
            Inv[i][2*h]*B[2*h];
        if (Dm%2)
        {
            S[h] = Inv[i][Dm]*B[Dm];
            if (h%2)
                S[h+1] = 0;
        }
        else
            S[h]=0.0;
        X[i] = Soma(S,1,1,i,Dm,Aux);
    }
}

```

```

    }
}

```

Neste procedimento calcula-se o produto da matriz $Inv[\cdot][\cdot]$ pelo vetor $B[\cdot]$. O resultado é armazenado no vetor $X[\cdot]$.

Procedimento Soma

```

double Soma(double *Sm,int a, int b, int c, int d, int Aux)
{
    int g, h, Aux2;
    Aux2 = 2;

    for (h=2; h<=Aux; h++)
    {
        Aux2= Aux2*2;
        for (g=1; g<= ceil((float)d/Aux2); g++)
        *   Sm[g] = Sm[2*g-1]+Sm[2*g];
        Sm[g] = 0;
    }
    return Sm[1];
}

```

Dado um vetor $Sm[\cdot]$ de tamanho d , esta função calcula a soma destas posições utilizando o algoritmo *fan-in*. A variável Aux traz o valor de $\lceil \log d \rceil$. O primeiro valor de h é 2 uma vez que as primeiras $\lceil \frac{d}{2} \rceil$ adições já foram determinadas. O resultado armazenado em $Sm[1]$ é retornado ao procedimento que a ativou. As variáveis a , b e c são utilizadas como índices da matriz de contagem de operações.

5.6 Conclusão

O algoritmo de Csanky apresenta um resultado teórico muito importante, pois inclui os problemas de inversão de matrizes e solução de sistemas lineares na classe NC^2 , diminuindo a diferença entre a complexidade do algoritmo e a cota inferior para estes tipos de problemas.

A implementação deste algoritmo é a mais complexa entre todas as desenvolvidas neste trabalho. Tivemos que tomar cuidado com os índices envolvidos, pois trabalhamos com matrizes cujas ordens não eram potências de dois.

Apesar do resultado teórico obtido com a diminuição do número de operações, o algoritmo é numericamente instável e o número de processadores é elevado para uma utilização prática.

Capítulo 6

Resultados

6.1 Introdução

Nos três capítulos anteriores, apresentamos alguns algoritmos paralelos para a solução de sistemas lineares e suas respectivas implementações. Nas implementações utilizamos matrizes de contadores para armazenar o número de operações por processador para cada algoritmo, utilizando, preferencialmente, as posições de menores índices. Um exemplo da utilização destas matrizes pode ser vista na seção 3.8. Através destas matrizes, obtivemos alguns resultados do comportamento destes algoritmos, que estão apresentados neste capítulo. Estes resultados são utilizados para comparar o desempenho dos algoritmos estudados. Os dados não-inteiros foram arredondados na terceira casa decimal.

Teoricamente, com exceção dos algoritmos de Gauss com pivotação e de Csanky, todos os algoritmos apresentam uma complexidade de ordem linear, como pode ser visto na tabela 6.1.

Algoritmo	Complexidade	Processadores
Gauss	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Gauss com pivotação	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Gauss-Jordan	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
QR -Givens	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
QR -Cholesky	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Csanky	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n^4)$

Tabela 6.1: Tabela de complexidades e número de processadores de cada um dos algoritmos.

As implementações foram realizadas em linguagem C nas estações SUN. Nos preocupa-

mos em contar operações básicas (adição, subtração, multiplicação, divisão e comparação); a operação raiz quadrada, presente em alguns algoritmos, foi contada à parte, não sendo incluída na totalização do número de operações. Nos algoritmos de Gauss, Gauss com pivotação, Gauss-Jordan e QR -Cholesky utilizamos matrizes de contadores de tamanho n^2 ; no algoritmo QR -Givens $4(n^2 + n)$; e $n^4/2$ no algoritmo de Csanky.

Como nosso principal objetivo é a comparação destes algoritmos, executamos suas implementações para matrizes de ordem máxima 100, sendo possível executar estes algoritmos para matrizes maiores. O único inconveniente seria na visualização dos valores, pois alguns deles são muito altos em relação aos demais, prejudicando esta visualização. Só foi possível executar o algoritmo de Csanky com matrizes de ordem até 70, por causa da insuficiência de memória das máquinas pela utilização da matriz de contagem de operações de dimensão $n^4/2$.

6.2 Número de Operações

Em cada um dos algoritmos, contamos as operações efetuadas pelos processadores para cada ordem da matriz. Obtivemos o número máximo de operações, que corresponde à complexidade do algoritmo, e o número médio de operações por processador. Consideramos, para cada uma das medidas, o total de operações e cada operação separadamente. A seguir apresentamos os resultados destas medições.

6.2.1 Total de Operações - Máximo

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	46	71	30	44	164	45
20	96	165	60	89	334	63
30	146	265	90	134	504	68
40	196	373	120	179	674	84
50	246	483	150	224	844	90
60	296	593	180	269	1.014	90
70	346	709	210	314	1.184	106
80	396	829	240	359	1.354	
90	446	949	270	404	1.524	
100	496	1.069	300	449	1.694	

Tabela 6.2: Tabela do número máximo de operações.

Na tabela 6.2 apresentamos o número máximo de operações para cada valor de n . Estes dados podem ser visualizados no gráfico da figura 6.1.

Figura 6.1: Gráfico do número total máximo de operações em função da ordem das matrizes

Observemos que o melhor algoritmo é o de Csanky. Este algoritmo teoricamente é de ordem logarítmica quadrada. A constante multiplicativa acaba fazendo com que este algoritmo tenha um desempenho inferior ao algoritmo de Gauss-Jordan para matrizes de ordem 10 e 20. Para matrizes de ordem 10, o número total de operações do algoritmo de Csanky e dos algoritmos de Gauss e QR -Givens é muito próximo; o algoritmo de Gauss-Jordan apresenta o menor valor. Os algoritmos de Gauss e QR -Givens apresentam pouca diferença no número de operações, no gráfico da figura 6.1 podemos observar um crescimento muito próximo destes dois algoritmos. O algoritmo de Gauss com pivotação apresenta-se como o segundo pior por causa da determinação do pivô. O algoritmo QR -Cholesky é o que apresenta maior número de operações.

O algoritmo de Csanky apresenta intervalos de valores de n onde o número de operações é constante. Como consideramos o teto do logaritmo de n como sendo o número de operações dos laços e nos preocupamos em utilizar os processadores com menores índices, os valores entre duas potências de dois consecutivas deveriam ser os mesmos. Porém, isto só acontece em alguns subintervalos, por causa do algoritmo de Sameh e Brent que utilizamos para resolver o sistema triangular inferior $Tp = -t$.

6.2.2 Total de Operações - Média

No caso da média o algoritmo QR -Cholesky é o de maior número médio de operações. O crescimento é muito parecido com o gráfico do número máximo de operações.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	8,050	8,600	10,450	7,591	64,440	4,329
20	14,750	15,323	20,475	15,167	124,473	4,136
30	21,461	21,998	30,483	22,694	184,482	4,018
40	28,138	28,673	40,488	30,207	244,487	4,061
50	34,810	35,342	50,490	37,716	304,489	4,089
60	41,481	42,007	60,492	45,221	364,491	4,006
70	48,150	48,675	70,493	52,725	424,493	4,091
80	54,819	55,353	80,494	60,228	484,494	
90	61,487	62,010	90,494	67,730	544,494	
100	68,155	68,676	100,495	75,233	604,495	

Tabela 6.3: Tabela do número médio de operações por processador em cada algoritmo.

O número médio de operações do algoritmo de Csanky mantém-se em cerca de 4 operações por processador. Os algoritmos de Gauss e Gauss com pivotação apresentam uma diferença de meia unidade nos seus valores. O algoritmo QR -Givens tem um número médio de operações maior que a dos dois anteriores. O algoritmo de Gauss-Jordan apresenta número médio de operações pouco acima de um terço dos seus valores máximos. O algoritmo QR -Cholesky é o que apresenta maior número de operações.

6.2.3 Adições - Máximo

O número de adições máximo efetuado por cada algoritmo pode ser visto no gráfico da figura 6.3. Os algoritmos de Gauss, Gauss com pivotação e Gauss-Jordan não efetuam adições.

Entre os três algoritmos o que apresenta o maior número de adições é o algoritmo QR -Cholesky para $n \geq 30$. O algoritmo de Csanky apresenta um número maior de adições para matrizes de ordem 10 e 20. Apesar de não conseguirmos observar matrizes de ordem maior que 70, pelo gráfico pudemos ver que o algoritmo de Csanky tende a apresentar o menor número máximo desta operação com o crescimento da ordem das matrizes.

Figura 6.2: Gráfico do número médio de operações em função da ordem das matrizes.

Figura 6.3: Gráfico do número máximo de adições em função da ordem das matrizes.

6.2.4 Adições - Média

Novamente temos um crescimento mais rápido do algoritmo QR -Cholesky. Os outros dois algoritmos apresentam um valor muito próximo para matrizes de ordem 10. Este valor se mantém em cerca de 2 adições por processador no algoritmo de Csanky e apresenta um crescimento pouco acentuado no algoritmo QR -Givens.

Figura 6.4: Gráfico do número médio de adições por processador em função da ordem das matrizes.

6.2.5 Subtrações - Máximo

Nas subtrações o comportamento dos algoritmos Gauss-Jordan e QR -Givens é quase o mesmo, com uma diferença de 1 unidade, e estes são os que apresentam o menor número máximo de subtrações. Os algoritmos de Gauss e Gauss com pivotação têm o mesmo número de subtrações, o que era de se esperar, pois a única diferença entre estes dois algoritmos está na determinação do pivô que envolve apenas comparações, trazendo diferenças apenas no número total de operações.

O algoritmo de Csanky não efetua esta operação e mais uma vez o algoritmo QR -Cholesky é o de maior quantidade de operações, como podemos observar na figura 6.5.

Figura 6.5: Gráfico do número máximo de subtrações em função da ordem das matrizes.

6.2.6 Subtrações - Média

O algoritmo QR -Givens apresenta um número médio de subtrações muito próximo de zero, pois ele só efetua subtrações na determinação da solução do sistema triangular superior equivalente ao sistema original. Os algoritmos Gauss e Gauss com pivotação possuem o mesmo crescimento. Gauss-Jordan apresenta um número médio de subtrações superior aos demais sendo ultrapassado somente pelo algoritmo QR -Cholesky.

6.2.7 Multiplicações - Máximo

Todos os algoritmos, exceto QR -Cholesky apresentam um número máximo de multiplicações muito próximo para matrizes de ordem 10 como pode ser acompanhado no gráfico da figura 6.7. Os algoritmos de Gauss, Gauss com pivotação e QR -Givens apresentam valores muito próximos para o número máximo de multiplicações, sendo o mesmo para os dois primeiros e uma unidade superior para o terceiro. O número máximo de multiplicações do algoritmo de Gauss-Jordan é pouco mais que metade do valor desta medida para os três algoritmos anteriores. O algoritmo de Csanky apresenta valores entre 9 e 15 para esta operação. O algoritmo QR -Cholesky tem os maiores valores, que estão acima de 70 multiplicações.

Figura 6.6: Gráfico do número médio de subtrações por processador em função da ordem das matrizes.

Figura 6.7: Gráfico do número máximo de multiplicações em função da ordem das matrizes.

6.2.8 Multiplicações - Média

O número médio de multiplicações para matrizes de ordem 10 está entre 2 e 4 multiplicações por processador para todos os algoritmos, com a exceção do algoritmo *QR-Cholesky*. Os algoritmos de Gauss e Gauss com pivotação apresentam o mesmo crescimento. Nos algoritmos de Gauss-Jordan e *QR-Givens* os valores são praticamente os mesmos, com uma diferença de décimos, mas têm um crescimento superior ao dos dois anteriores.

Figura 6.8: Gráfico do número médio de multiplicações por processador em função da ordem das matrizes.

6.2.9 Divisão - Máximo

O algoritmo de Csanky apresenta um número de divisões constante e igual a 3 pois em apenas três pontos do algoritmo são efetuadas divisões. Logo a seguir está o algoritmo *QR-Givens*. Os algoritmos de Gauss, Gauss com pivotação e Gauss-Jordan apresentam o mesmo número máximo de divisões. Os maiores valores são obtidos pelo algoritmo *QR-Cholesky* por causa da determinação da decomposição de Cholesky.

Figura 6.9: Gráfico do número máximo de divisões em função da ordem das matrizes

6.2.10 Divisão - Média

Em todos os algoritmos, exceto *QR-Cholesky*, o número total de divisões efetuadas por todos os algoritmos é muito baixa, com isto a média de divisões por processador é muito próxima de zero como podemos observar no gráfico da figura 6.10. Nos algoritmos de Gauss, Gauss com pivotação e Gauss-Jordan este número é o mesmo e próximo a 0,5 e decresce com o aumento de n . *QR-Givens* apresenta valores crescentes e pouco acima de 3 décimos. Os valores para o algoritmo de Csanky são os mais baixos e atingem 1 milésimo para $n = 70$ e seus valores continuam decrescendo.

6.2.11 Raízes Quadradas - Máximo

Os únicos algoritmos que efetuam raízes quadradas são os da fatoração *QR*. No gráfico da figura 6.11 podemos acompanhar o número máximo desta operação. Neste caso o algoritmo *QR-Cholesky* apresenta um número de operações constante e igual a 1. Neste algoritmo a raiz quadrada é aplicada apenas na determinação dos elementos da diagonal principal da matriz L , que é feita simultaneamente. Já o algoritmo *QR-Givens* efetua uma raiz quadrada a cada rotação aplicada à matriz em cada iteração.

Figura 6.10: Gráfico do número médio de divisões por processador em função da ordem das matrizes.

Figura 6.11: Gráfico do número máximo de raízes quadradas em função da ordem das matrizes.

6.2.12 Raízes Quadradas - Média

O número médio de raízes quadradas por processador decai com o crescimento da ordem das matrizes no algoritmo *QR*-Cholesky e se mantém praticamente constante no algoritmo *QR*-Givens.

Figura 6.12: Gráfico do número médio de raízes quadradas por processador em função da ordem das matrizes.

6.2.13 Comparações - Máximo

O número máximo de comparações obtidas no algoritmo de Gauss com pivotação está mostrado no gráfico da figura 6.13. Este é o único algoritmo que realiza esta operação.

6.2.14 Comparações - Média

No gráfico da figura 6.14 apresentamos o número de comparações média por processador para esta operação no algoritmos de Gauss com pivotação. Esta operação é realizada na determinação do pivô. Vemos que o número médio destas operações está em cerca de meia unidade, com seus valores decrescendo à medida que n cresce.

Figura 6.13: Gráfico do número máximo de comparações em função da ordem das matrizes

Figura 6.14: Gráfico do número médio de comparações por processador em função da ordem das matrizes.

6.3 Utilização dos Processadores

Para se ter uma noção empírica da utilização dos processadores obtivemos a porcentagem de processadores totalmente ociosos e a porcentagem de processadores que executaram menos operações que a média de operações por processador. Estas medidas não são significativas se tomadas isoladamente, mas podemos ter uma noção de utilização dos processadores comparando os valores em cada algoritmo.

6.3.1 Porcentagem de Processadores com Carga Nula

No total de operações o algoritmo de Csanky é o que apresenta maior ociosidade. Apesar de, teoricamente, o número de processadores ser $\mathcal{O}(n^4)$, utilizamos, na implementação deste algoritmo, $n^4/2$ processadores, que são suficientes. Os demais algoritmos apresentam uma ociosidade próxima a 10% para matrizes de ordem 10 e decai até pouco menos de 1% para matrizes de ordem 100. O algoritmo *QR*-Cholesky não apresenta processadores ociosos. A oscilação destes valores para o algoritmo de Csanky é causada pela resolução do sistema triangular inferior $Tp = -t$ como já observamos na seção 6.2.1. Um gráfico de barras destes dados pode ser visto no apêndice C.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	<i>QR</i> -Givens	<i>QR</i> -Cholesky	Csanky
10	9,000	9,000	9,000	7,727	0,000	19,800
20	4,750	4,750	4,750	3,810	0,000	19,973
30	3,222	3,222	3,222	2,527	0,000	6,659
40	2,438	2,438	2,438	1,890	0,000	19,996
50	1,960	1,960	1,960	1,509	0,000	27,997
60	1,639	1,639	1,639	1,257	0,000	6,666
70	1,408	1,408	1,408	1,076	0,000	8,571
80	1,234	1,234	1,234	0,941	0,000	
90	1,099	1,099	1,099	0,836	0,000	
100	0,990	0,990	0,990	0,752	0,000	

Tabela 6.4: Tabela das porcentagens de processadores totalmente ociosos considerando o número total de operações realizadas por cada algoritmo para cada valor de n

Se observarmos a porcentagem de zeros considerando cada operação separadamente, vemos que na adição o algoritmo *QR*-Cholesky não apresenta processadores ociosos, o algoritmo *QR*-Givens apresenta uma porcentagem de 55% de processadores ociosos para $n = 10$, decrescendo até cerca de 50% para $n = 100$. O algoritmo de Csanky apresenta porcentagens oscilando entre 53% e 64% de processadores ociosos.

n	Algoritmo		
	QR -Givens	QR -Cholesky	Csanky
10	55,000	0,000	59,820
20	52,500	0,000	59,975
30	51,666	0,000	53,326
40	51,250	0,000	59,996
50	51,000	0,000	63,997
60	50,833	0,000	53,332
70	50,714	0,000	54,285
80	50,625	0,000	
90	50,556	0,000	
100	50,500	0,000	

Tabela 6.5: Tabela das porcentagens de processadores totalmente ociosos considerando o número de adições executadas por cada algoritmo em relação às ordens das matrizes.

n	Algoritmo				
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky
10	10,000	10,000	10,000	97,955	19,000
20	5,000	5,000	5,000	98,869	9,750
30	3,333	3,333	3,333	99,220	6,556
40	2,500	2,500	2,500	99,405	4,938
50	2,000	2,000	2,000	99,520	3,960
60	1,667	1,667	1,667	99,597	3,306
70	1,429	1,429	1,429	99,653	2,837
80	1,250	1,250	1,250	99,695	2,484
90	1,111	1,111	1,111	99,728	2,210
100	1,000	1,000	1,000	99,755	1,990

Tabela 6.6: Tabela das porcentagens de processadores totalmente ociosos considerando o número de subtrações realizadas por cada algoritmo para cada valor de n

A subtração apresenta cerca de 10% de processadores ociosos para matrizes de ordem 10 chegando a cerca de 1% de ociosidade para matrizes de ordem 100 nos algoritmos de Gauss, Gauss com pivotação e Gauss-Jordan, caindo pela metade cada vez que n dobra de valor. O algoritmo QR -Cholesky apresenta 19% de ociosidade para matrizes de ordem 10, atingindo uma porcentagem de cerca de 2% para $n = 100$. O algoritmo QR -Givens apresenta uma ociosidade próxima a 100% para esta operação, isto ocorre porque esta operação é executada somente na solução do sistema triangular superior, utilizando substituições recorrentes.

Na multiplicação o algoritmo QR -Cholesky não apresenta processadores ociosos. Os algoritmos de Gauss, Gauss com pivotação e Gauss-Jordan comportam-se como na subtração, no algoritmo QR -Givens obtemos os mesmos valores do algoritmo de Gauss. O algoritmo com maior ociosidade é o algoritmo de Csanky com valores entre 6% e 28%.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	10,000	10,000	10,000	10,000	0,000	19,800
20	5,000	5,000	5,000	5,000	0,000	19,973
30	3,333	3,333	3,333	3,333	0,000	6,659
40	2,500	2,500	2,500	2,500	0,000	19,996
50	2,000	2,000	2,000	2,000	0,000	27,997
60	1,667	1,667	1,667	1,667	0,000	6,666
70	1,429	1,429	1,429	1,429	0,000	8,571
80	1,250	1,250	1,250	1,250	0,000	
90	1,111	1,111	1,111	1,111	0,000	
100	1,000	1,000	1,000	1,000	0,000	

Tabela 6.7: Tabela das porcentagens de processadores totalmente ociosos considerando o número de multiplicações realizadas por cada algoritmo para cada valor de n

A divisão é a operação menos utilizada. O algoritmo QR -Cholesky tem uma porcentagem de ociosidade abaixo de 20% atingindo cerca de 2% para $n = 100$. O algoritmo QR -Givens apresenta ociosidade abaixo de 90%, todos os demais apresentam ociosidade acima de 90%, chegando a quase 100% para $n = 100$.

A ociosidade dos processadores na operação raiz quadrada é alta, acima de 90% para os dois algoritmos que a utilizam atingindo ociosidade próxima a 100% para matrizes de ordem 100. Isto é causado pela baixa utilização desta operação.

Na operação comparação, a porcentagem de processadores com carga nula está acima de 95%, atingindo mais de 99% para $n = 100$.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	90,000	90,000	90,000	85,455	17,000	98,000
20	95,000	95,000	95,000	86,369	9,250	99,500
30	96,666	96,666	96,666	86,720	6,333	99,778
40	97,500	97,500	97,500	86,905	4,813	99,875
50	98,000	98,000	98,000	87,020	3,880	99,920
60	98,333	98,333	98,333	87,097	3,250	99,944
70	98,571	98,571	98,571	87,153	2,796	99,959
80	98,750	98,750	98,750	87,195	2,453	
90	98,889	98,889	98,889	87,228	2,185	
100	99,000	99,000	99,000	87,255	1,970	

Tabela 6.8: Tabela das porcentagens de processadores totalmente ociosos considerando o número de divisões realizadas por cada algoritmo para cada valor de n .

n	Algoritmo	
	QR -Givens	QR -Cholesky
10	97,955	90,000
20	98,869	95,000
30	99,220	96,667
40	99,405	97,500
50	99,520	98,000
60	99,597	98,333
70	99,653	98,571
80	99,695	98,750
90	99,728	98,889
100	99,755	99,000

Tabela 6.9: Tabela das porcentagens de processadores totalmente ociosos considerando o número de raízes quadradas realizadas por cada algoritmo para cada valor de n .

n	Algoritmo
	Gauss(P)
10	95,000
20	97,500
30	98,333
40	98,750
50	99,000
60	99,167
70	99,286
80	99,375
90	99,444
100	99,500

Tabela 6.10: Tabela das porcentagens de processadores totalmente ociosos considerando o número de comparações realizadas para cada valor de n .

6.3.2 Porcentagem de Processadores abaixo da Média

Os algoritmos de Gauss e Gauss com pivotação apresentam entre 55% e 67% de processadores com carga abaixo da média considerando o número total de operações. Nos algoritmos Gauss-Jordan e QR -Givens entre 50% e 55% trabalham abaixo da carga média, com os valores decaindo à medida que n cresce, e a maior taxa é do algoritmo de Csanky com valores entre 66% e 74% de processadores trabalhando abaixo da média. O algoritmo QR -Cholesky mantém seus valores em torno de 63% de processadores com carga abaixo da média. Estes valores podem ser vistos em um gráfico de barras no apêndice C.

Ao observarmos cada operação separadamente, o algoritmo QR -Cholesky não apresenta processadores trabalhando abaixo da média para a adição. QR -Givens apresenta uma porcentagem destes processadores entre 50% e 55% e Csanky entre 72% e 77%.

No caso da subtração os algoritmos de Gauss e Gauss com pivotação apresentam um pico de 60% de processadores trabalhando abaixo da média para matrizes de ordem 20 mas mantém-se em cerca de 55% para os demais valores de n . O algoritmo de Gauss-Jordan tem um valor inicial próximo de 46% chegando a pouco menos de 50% de processadores com carga abaixo da média para matrizes de ordem 100. QR -Cholesky apresenta esta porcentagem entre 63% e 64,5% e QR -Givens atinge quase 100% de processadores subutilizados para esta operação, isto causado pela baixa quantidade de subtrações deste algoritmo.

Nas multiplicações, QR -Givens tem 10% dos processadores trabalhando abaixo da média inicialmente, esta porcentagem chega a 1% com o crescimento de n . Gauss e Gauss

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	67,000	67,000	55,000	55,000	63,000	66,000
20	59,750	59,750	52,500	52,500	63,250	71,000
30	57,111	57,111	51,666	51,666	63,333	73,555
40	58,813	58,813	51,250	51,250	63,625	72,000
50	57,320	57,320	51,000	51,000	63,520	71,040
60	56,333	56,333	50,833	50,833	63,500	74,000
70	57,449	57,449	50,714	50,714	63,673	73,714
80	56,672	56,672	50,625	50,625	63,609	
90	47,362	47,362	50,556	50,556	63,568	
100	52,109	52,109	50,500	50,500	63,680	

Tabela 6.11: Tabela das porcentagens de processadores com carga abaixo da média do total de operações por processador.

com pivotação apresentam uma porcentagem de subutilização de 56% aproximadamente, este valor chega a 60% somente para matrizes de ordem 20. QR -Cholesky tem uma porcentagem entre 63% e 65% dos processadores com carga abaixo da média para esta operação. O algoritmo de Gauss-Jordan tem inicialmente uma porcentagem de 46%, esta porcentagem chega a pouco menos de 50% para $n = 100$. O algoritmo de Csanky tem entre 59% e pouco mais de 77% dos processadores com número de multiplicações abaixo da média.

A divisão é uma operação pouco utilizada pelos algoritmos. Os algoritmos de Gauss, Gauss com pivotação e Gauss-Jordan apresentam valores, inicialmente, próximos a 90% chegando a quase 99% com o crescimento de n . Csanky mantém seus valores acima de 98%. QR -Givens tem uma porcentagem de divisões abaixo da média entre 85% e 88%. O algoritmo QR -Cholesky apresenta valores entre 59% e 64% de processadores com carga abaixo da média de divisões.

A raiz quadrada apresenta porcentagens de subutilização acima de 90% para ambos algoritmos uma vez que é uma operação de pouca utilização.

A porcentagem de processadores com carga abaixo da média de comparações por processador é a mesma da porcentagem de processadores com carga nula. Esta operação é pouco utilizada e necessária somente na determinação do pivô.

n	Algoritmo		
	QR -Givens	QR -Cholesky	Csanky
10	55,000	0,000	72,000
20	52,500	0,000	74,000
30	51,666	0,000	74,222
40	51,250	0,000	75,000
50	51,000	0,000	74,560
60	50,833	0,000	74,667
70	50,714	0,000	76,816
80	50,625	0,000	
90	50,556	0,000	
100	50,500	0,000	

Tabela 6.12: Tabela das porcentagens de processadores com carga abaixo da média do número de adições por processador.

n	Algoritmo				
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky
10	56,000	56,000	46,000	97,955	64,000
20	60,000	60,000	47,750	98,869	64,500
30	57,222	57,222	48,444	99,220	63,556
40	55,688	55,688	48,813	99,405	63,813
50	57,400	57,400	49,040	99,520	64,000
60	56,389	56,389	49,194	99,597	63,611
70	55,633	55,633	49,306	99,653	63,776
80	56,719	56,719	49,391	99,695	63,891
90	56,111	56,111	49,457	99,728	63,641
100	55,610	55,610	49,510	99,755	63,750

Tabela 6.13: Tabela das porcentagens de processadores com carga abaixo da média do número de subtrações por processador.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	56,000	56,000	46,000	10,000	65,000	59,920
20	60,000	60,000	47,750	5,000	63,500	59,988
30	57,222	57,222	48,444	3,333	63,889	73,296
40	55,688	55,688	48,813	2,500	64,125	59,998
50	57,400	57,400	49,040	2,000	63,640	67,989
60	56,389	56,389	49,194	1,667	63,833	73,324
70	55,633	55,633	49,306	1,429	63,959	77,134
80	56,719	56,719	49,391	1,250	63,672	
90	56,111	56,111	49,457	1,111	63,790	
100	55,610	55,610	49,510	1,000	63,890	

Tabela 6.14: Tabela das porcentagens de processadores com carga abaixo da média do número de multiplicações por processador.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	90,000	90,000	90,000	85,455	59,000	98,000
20	95,000	95,000	95,000	86,369	61,500	99,500
30	96,666	96,666	96,666	86,720	62,556	99,778
40	97,500	97,500	97,500	86,905	62,375	99,875
50	98,000	98,000	98,000	87,020	62,880	99,920
60	98,333	98,333	98,333	87,097	63,167	99,944
70	98,571	98,571	98,571	87,153	62,959	99,959
80	98,750	98,750	98,750	87,195	63,172	
90	98,889	98,889	98,889	87,228	63,358	
100	99,000	99,000	99,000	87,255	63,190	

Tabela 6.15: Tabela das porcentagens de processadores com carga abaixo da média do número de divisões por processador.

n	Algoritmo	
	QR -Givens	QR -Cholesky
10	97,955	90,000
20	98,869	95,000
30	99,220	96,667
40	99,405	97,500
50	99,520	98,000
60	99,597	98,333
70	99,653	98,571
80	99,695	98,750
90	99,728	98,889
100	99,755	99,000

Tabela 6.16: Tabela das porcentagens de processadores com carga abaixo da média do número de raízes quadradas por processador.

n	Algoritmo
	Gauss(P)
10	95,000
20	97,500
30	98,333
40	98,750
50	99,000
60	99,167
70	99,286
80	99,375
90	99,444
100	99,500

Tabela 6.17: Tabela das porcentagens de processadores com carga abaixo da média do número de comparações por processador.

6.4 Erros

Ao executarmos as implementações dos algoritmos nos preocupamos em obter também os erros na determinação das soluções. Foram geradas matrizes de coeficientes que estão descritas no apêndice B. Os três tipos de sistemas foram: Binomial, de Hilbert e Tridiagonal, cujas soluções são vetores formados de uns. A análise teórica dos erros destes algoritmos pode ser obtida em [6, 17, 22, 23, 38, 48, 49].

O algoritmo *QR*-Givens não pôde ser utilizado para matrizes de coeficientes esparsas uma vez que são realizadas divisões cujos quocientes são obtidos pela adição de dois elementos da matriz. Por este motivo fizemos uma pequena adaptação nos sistemas de Hilbert para que a matriz gerada fosse triangular inferior e não triangular superior. Este algoritmo não foi executado para matrizes tridiagonais pelo motivo que acabamos de expor. Uma variação deste algoritmo para este tipo de matrizes pode ser obtido em Sameh e Kuck [40].

Na tabela 6.18 apresentamos os erros utilizando sistemas Binomiais. Estes testes acabaram sendo muito restritos uma vez que o vetor de termos independentes apresenta valores muito elevados com o crescimento da ordem da matriz.

Algoritmo	Ordem					
	10	20	30	40	50	60
Gauss	–	–	–	–	–	*
Gauss com pivotação	–	10^{-8}	*			
Gauss-Jordan	–	–	–	–	–	*
<i>QR</i> -Givens	–	10^{-8}	*			
<i>QR</i> -Cholesky	–	*				
Csanky	10^{-7}	*				

Tabela 6.18: Tabela dos erros na solução dos sistemas Binomiais para cada um dos algoritmos. As posições marcadas com – representam algoritmos que não apresentaram erro para aquela ordem de matriz. As posições marcadas com * representam erros muito grandes (na casa das unidades).

Na execução utilizando sistemas de Hilbert obtivemos os erros mostrados na tabela 6.19.

Os sistemas tridiagonais não apresentaram erros nos algoritmos executados, à exceção do algoritmo de Csanky cujos erros estão descritos na tabela 6.20.

Os algoritmos de Gauss, Gauss com pivotação, Gauss-Jordan e *QR*-Givens têm o mesmo comportamento numérico das suas correspondentes versões sequenciais, pois nas

Algoritmo	Ordem									
	10	20	30	40	50	60	70	80	90	100
Gauss	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
Gauss (pivot.)	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
Gauss-Jordan	10^{-5}	10^{-5}	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-3}	10^{-4}	10^{-4}	10^{-4}
QR-Givens	10^{-5}	10^{-5}	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-3}	10^{-4}	10^{-4}	10^{-4}
QR-Cholesky	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-4}	10^{-4}	10^{-3}	10^{-4}	*	
Csanky	10^{-5}	10^{-2}	*							

Tabela 6.19: Tabela dos erros na solução dos sistemas de Hilbert para cada um dos algoritmos. As posições marcadas com * representam erros muito grandes (na casa das unidades).

Algoritmo	Ordem		
	10	20	30
Csanky	10^{-10}	10^{-4}	*

Tabela 6.20: Tabela dos erros na solução dos sistemas tridiagonais para o algoritmo de Csanky. As posições marcadas com * representam erros muito grandes (na casa das unidades).

implementações paralelas a quantidade total de operações e o tipo de operações sobre os elementos são os mesmos dos algoritmos sequenciais, a diferença está na execução simultânea de algumas tarefas.

6.5 Ganho

Tendo o número máximo de operações dos algoritmos, podemos obter o ganho. Como vimos, o ganho ($S_p(n)$), teoricamente, é dado pela razão entre a complexidade do melhor algoritmo sequencial ($T(n)$) e a complexidade do algoritmo paralelo. Ao trabalharmos com os dados numéricos, optamos por utilizar $T(n) = n^3$. Os resultados estão disponíveis no gráfico da figura 6.15. Este gráfico nos dá uma idéia melhor do ganho destes algoritmos, pois comparamos o número de operações máximo de cada um deles com um mesmo valor para o caso sequencial.

Observamos que o melhor ganho é do algoritmo de Csanky, pois ele apresenta o menor número máximo de operações. Por outro lado, o algoritmo QR-Cholesky é o que apresenta menor ganho, obviamente por causa do seu número máximo de operações maior.

Figura 6.15: Gráfico do ganho dos algoritmos em função da ordem das matrizes ($T(n) = n^3$).

Obtivemos, também, o ganho ($S_p^*(n)$) através da razão entre o número de operações total do algoritmo utilizando um único processador e o número máximo de operações. Estes dados estão apresentados no gráfico da figura 6.16.

Novamente, o melhor ganho é do algoritmo de Csanky, pois o número de operações deste algoritmo utilizando um único processador é grande, e o número máximo de operações é o menor dentre os algoritmos estudados. Os menores ganhos são dos algoritmos de Gauss e Gauss com pivotação, pois o número de operações utilizando um único processador é pequeno (ver tabela C.13), mas o número máximo de operações é o segundo maior.

6.6 Conclusão

Através dos resultados das implementações mostrados neste capítulo, pudemos avaliar o comportamento dos algoritmos.

Considerando o número total de operações, o algoritmo de Csanky é o melhor dos algoritmos estudados, comprovando o que já havia sido visto teoricamente. O algoritmo de Gauss-Jordan apresenta um desempenho melhor que o algoritmo de Gauss no caso paralelo, invertendo a situação que existia no caso seqüencial. Os algoritmos de Gauss e QR -Givens tem um número de operações muito próximo e o algoritmo de Gauss com pivotação apresenta um número de operações um pouco maior que estes dois, por causa da determinação do pivô. O algoritmo QR -Cholesky é o que apresenta maior número de operações, por causa da versão do algoritmo que utilizamos.

O número médio de operações por processador mostra a utilização dos processadores. Neste caso acontece uma inversão nos gráficos dos algoritmos Gauss com pivotação e Gauss-Jordan, este passa a ser o segundo pior algoritmo e o algoritmo de Gauss com pivotação acompanha o crescimento dos algoritmos de Gauss e QR -Givens.

As duas medições acima foram feitas para cada operação separadamente para mostrar a utilização dos processadores por cada uma delas.

Ao mostrarmos a utilização dos processadores, determinando as porcentagens de processadores que não efetuaram operações e as porcentagens de processadores com carga abaixo da média de operações por processador, chegamos a valores abaixo de 10% para os algoritmos de Gauss, Gauss com pivotação, Gauss-Jordan e QR -Givens. QR -Cholesky não apresentou processadores ociosos por causa da utilização total dos processadores. Os maiores valores são obtidos no algoritmo de Csanky. Na porcentagem de processadores com carga abaixo da média os valores aumentam. Os melhores algoritmos são os algoritmos QR -Givens e Gauss-Jordan. Csanky é o que apresenta os maiores valores, e os outros três apresentam valores intermediários entre os dois primeiros e o algoritmo de Csanky. Estas medições também foram feitas para as operações separadamente, para termos uma noção da utilização dos processadores por cada uma delas.

Figura 6.16: Gráfico do ganho dos algoritmos em função da ordem das matrizes ($T(n)$ é o número de operações do algoritmo, utilizando um único processador).

Na determinação dos erros, os mais significativos foram obtidos pelo algoritmo de Csanky. Dos sistemas padrão utilizados, os sistemas de Hilbert foram os que possibilitaram uma visualização melhor dos erros cometidos. As menores diferenças ocorrem nos algoritmos de Gauss e Gauss com pivotação, com uma leve melhora no segundo. Os outros algoritmos apresentaram erros entre 10^{-3} e 10^{-5} . Os algoritmos QR -Cholesky e Csanky apresentaram erros muito grandes (na casa das unidades).

O melhor ganho, nas duas formas mostradas ($S_p(n)$ e $S_p^*(n)$), foi do algoritmo de Csanky, por causa do número máximo de operações pequeno; no caso de $S_p^*(n)$, o número de operações do algoritmo utilizando um processador é grande.

Com estes resultados mostramos que realmente o melhor algoritmo é o de Csanky considerando o número de operações máximo e médio por processador, mas apresenta valores altos nas porcentagens de utilização dos processadores causados pelo número elevado de processadores. A instabilidade numérica deste algoritmo é muito grande. Dessa forma, a utilização prática deste algoritmo é muito restrita e pouco compensadora, servindo mais como resultado teórico do que uma alternativa para a solução de sistemas lineares.

Na prática, o algoritmo de Gauss com pivotação é amplamente utilizado. Nos resultados pudemos observar que os erros cometidos por este algoritmos são os menores. Na utilização dos processadores observamos que este algoritmo mantém uma boa quantidade de processadores trabalhando.

Os outros algoritmos apresentam algumas particularidades, sendo que sua utilização é justificada quando se deseja explorar as características inerentes a cada um deles.

Capítulo 7

Conclusões

Neste trabalho, analisamos algoritmos paralelos diretos para a solução de sistemas lineares densos de ordem n . Nas fatorações LU e QR , determinamos sistemas triangulares equivalentes ao sistema original, que são facilmente resolvidos, utilizando substituições recorrentes. O algoritmo de Csanky obtém a inversa de uma matriz dada. A solução de um sistema $Ax = b$, utilizando a inversa da matriz A dos coeficientes é dada por $x = A^{-1}b$. A descrição do algoritmo de Sameh e Brent [38], no livro de Lakshmivarahan e Dhall [28] apresentou alguns erros de sinais nas definições das matrizes utilizadas pelo algoritmo, que foram corrigidos.

Utilizamos a forma algebricamente menos formal do algoritmo da fatoração LU , o algoritmo de Gauss. Este algoritmo, utilizando pivotação parcial, é amplamente utilizado. Apresentamos o algoritmo de Gauss-Jordan, que é uma variante do algoritmo de Gauss e que, na versão seqüencial, executa mais operações que o algoritmo de Gauss, mas em paralelo é mais rápido.

Os algoritmos da fatoração QR são uma opção à pivotação do algoritmo de Gauss. QR -Givens tem uma descrição bastante simples e utiliza rotações planas para fatorar a matriz dos coeficientes no produto QR . Já o algoritmo QR -Cholesky oferece um resultado teórico importante utilizando o método de Csanky para algumas operações, porém preferimos utilizar, na implementação, uma versão numericamente mais estável e que envolve menos processadores. A demonstração do algoritmo da decomposição de Cholesky apresentava alguns índices errados no texto original de Datta [16], que estão corrigidas no presente texto.

O algoritmo de Csanky é o mais rápido algoritmo para a inversão de matrizes conhecido. Com o surgimento deste algoritmo obtivemos a inclusão deste problema e do problema da solução de sistemas lineares na classe de complexidade paralela NC^2 . O algoritmo de Csanky apresenta apenas um importante resultado teórico.

Com a execução das implementações dos algoritmos observamos que o algoritmo de

Csanky é o algoritmo que apresenta menor número máximo de operações. O algoritmo de Gauss é pior que o algoritmo de Gauss-Jordan, o inverso do que ocorre no caso sequencial. O algoritmo QR -Givens apresenta valores muito próximos aos obtidos com o algoritmo de Gauss e o algoritmo de Gauss com pivotação apresenta valores um pouco maiores. Os valores mais altos são obtidos no algoritmo QR -Cholesky.

No caso do número médio de operações por processador, os algoritmos de Gauss, Gauss com pivotação e QR -Givens apresentam valores bastante próximos. O algoritmo de Csanky apresenta os menores valores. O algoritmo Gauss-Jordan é o segundo pior e QR -Cholesky é o que apresenta os maiores valores.

A porcentagem de processadores com carga nula esteve abaixo de 10 pontos, chegando a 1%, nos algoritmos de Gauss, Gauss com pivotação, Gauss-Jordan e QR -Givens. O algoritmo QR -Cholesky não apresentou processadores ociosos e o algoritmo de Csanky apresentou os maiores valores.

Os algoritmos QR -Givens e Gauss-Jordan tiveram os menores valores para a porcentagem de processadores abaixo da média. Os maiores valores foram obtidos com o algoritmo de Csanky.

Utilizamos os sistemas Binomial, de Hilbert e Tridiagonal, descritos no apêndice B, na obtenção de erros numéricos na solução. Os erros mais significativos foram obtidos no algoritmo de Csanky. Os sistemas de Hilbert ofereceram uma melhor visualização dos erros. Os menores erros para estes sistemas foram obtidos nos algoritmos de Gauss e Gauss com pivotação.

O melhor ganho, apesar de não ser ótimo, é obtido no algoritmo de Csanky.

Com os resultados gerados pelas implementações confirmamos o número de operações mais baixo no algoritmo de Csanky. Apesar disto, os erros numéricos obtidos foram grandes confirmando a instabilidade deste algoritmo.

Os experimentos indicam que o algoritmo de Gauss com pivotação é um bom algoritmo em aplicações. O algoritmo QR -Givens poderia ser uma opção. Entretanto, ao aplicarmos uma rotação em uma matriz, alteramos a estrutura de duas linhas desta matriz, enquanto na eliminação gaussiana, apenas uma linha é alterada. O algoritmo de Gauss-Jordan, apesar de apresentar menos operações, não apresenta opções para o controle de multiplicadores muito grandes.

A compilação das implementações dos algoritmos, utilizando o compilador GNU CC, geraram programas executáveis cujos tamanhos, em bytes, estão mostrados na tabela 7.1.

A solução de sistemas lineares utilizando algoritmos paralelos é um problema que pode ser amplamente estudado. A descrição e o comportamento dos algoritmos muda conforme o modelo que está sendo utilizado. Em um modelo de memória distribuída a forma de armazenamento dos dados e a topologia escolhida influem no desenvolvimento do algo-

Algoritmo	Tamanho
Gauss	36.453
Gauss com pivotação	44.776
Gauss-Jordan	36.432
QR -Givens	44.950
QR -Cholesky	44.986
Csanky	69.838

Tabela 7.1: Tabela dos tamanhos (em bytes) dos programas executáveis dos algoritmos

ritmo e no seu desempenho. Pode-se resolver este problema iterativamente ou alterar-se o número de processadores, analisando o desempenho dos mesmos e a complexidade do algoritmo. Os algoritmos da fatoração QR também são utilizados para resolver problemas de autovalores. Os algoritmos aqui apresentados foram estudados para matrizes densas, variantes destes algoritmos podem ser estudados para matrizes esparsas, por exemplo, blocadas ou tridiagonais. As idéias e algoritmos contidos neste trabalho podem ser aproveitados em outros problemas de álgebra linear, em outros modelos e utilizando outras quantidades de processadores.

Apêndice A

PVM

Neste apêndice será mostrada uma visão geral do funcionamento da PVM e os principais comandos para a linguagem C. Maiores detalhes do funcionamento da PVM podem ser obtidos em [3].

A.1 Conceitos Básicos

A PVM (*Parallel Virtual Machine*) é um pacote que permite a computação concorrente em uma rede heterogênea de computadores paralelos ou seqüenciais, chamados *elementos de processamento*. Estes elementos de processamento são acessados por aplicações via uma biblioteca de rotinas de *interface* padrão. Estas rotinas permitem a iniciação e finalização de processos através da rede, bem como a comunicação e a sincronização entre processos.

Os programas de aplicação são formados de *componentes* que são subtarefas em um nível alto de granularidade¹. Durante a execução, *instâncias* múltiplas de cada componente podem ser iniciadas. A PVM é vista pelos programas de aplicação como um recurso geral e flexível que suporta um modelo de computação de troca de mensagens.

A PVM consiste de duas partes: um processo *daemon* que qualquer usuário pode instalar em uma máquina, e uma biblioteca de usuário que contém rotinas para iniciar processos em outras máquinas, para comunicação entre processos, e para sincronização de processos.

O *daemon*, chamado *pvm*, que reside em todos os computadores da rede é projetado de tal forma que qualquer usuário da rede pode instalá-lo em uma máquina. Quando um usuário executa uma aplicação PVM, ele executa um dos *pvm*s instalados o qual inicia um subconjunto de *pvm*s instalados. Esta coleção de *pvm*s em execução definem uma

¹A granularidade mede a relação entre a quantidade de dados e o número de processadores [31].

configuração PVM para aquele usuário.

Para utilizarmos a PVM devemos configurar um conjunto de máquinas. Inicialmente deve-se criar um arquivo contendo o nome dos mestres, um por linha. O mestre onde será executado o *pvm* deve ser o primeiro. A PVM é ativada com o comando *pvm nome-arquivo*. Isto fará com que um *pvm* se inicie em cada uma das outras máquinas.

A partir disto, pode-se executar os programas PVM. Para interromper-se a PVM deve-se matar o processo, que interromperá todos os outros *pvm*s e todos os processos nesta PVM.

A.2 Biblioteca de Comandos

A segunda parte da PVM é formada pela biblioteca de rotinas. Na tabela A.1 descrevemos as rotinas de comunicação e de *interface* PVM para a linguagem C.

Em um modelo mestre/escravo o programa mestre é iniciado e aciona os programas nos escravos, que são executados. A PVM não é restrita a este modelo, sendo possível iniciar-se um processo em uma máquina a partir de um processo PVM. No capítulo 3 utilizamos o modelo mestre/escravo. O mestre ativa *enroll()* para permitir o uso da PVM e a comunicação entre os processadores. Utiliza-se *snd()* e *rcv()* para passar mensagens entre os processadores.

Ao fim da execução dos programas PVM, deve-se ativar *leave()*, que permite à PVM desconectar-se dos processos e acompanhar quais processos ainda estão sendo executados.

Na PVM, o *buffer* mantém-se válido até a próxima chamada a *initsend()*. Assim, a mesma mensagem pode ser enviada várias vezes. Ao final do recebimento, o processador está ocioso (ou bloqueado) até que chegue uma mensagem satisfazendo os requisitos de chegada e esta é copiada para um *buffer*. O programa não terminará se um comando *rcv()* nunca for satisfeito com uma mensagem de chegada do tipo correto. Mais ainda, somente a área do tipo distingue diferentes mensagens na PVM. Um erro comum é a não utilização de tipos distintos suficientes nos comandos *snd()* e *rcv()* para diferenciar mensagens diferentes nos programas. Este tipo de erro leva a um comportamento imprevisível do algoritmo.

int enroll(char *component_name) - registra o processo na PVM e retorna o número da instância (≥ 0) em caso de sucesso ou < 0 caso contrário.
int get[tipo]([tipo] *x, int num) - obtém <i>num</i> valores do tipo [tipo] das mensagens recebidas e os associa a <i>x</i> . Retorna -1 se o <i>buffer</i> está vazio. [tipo] deve ser <i>nint</i> , <i>nfloat</i> , <i>ndfloat</i> , <i>ncplx</i> , <i>ndcplx</i> , <i>string</i> , ou <i>bytes</i> .
int initiate(char *arquivo_objeto, char *arq) - inicia um novo processo e retorna o número da instância (≥ 0) em caso de sucesso ou < 0 caso contrário. Se a arquitetura (<i>arq</i>) é NULL, então a PVM escolhe uma arquitetura.
void initsend() - inicializa o <i>buffer</i> de envio.
void leave() - saída dos processos da PVM.
int put[tipo]([tipo] *ptr, int num) - insere <i>num</i> valores começando em <i>ptr</i> no <i>buffer</i> de envio. Retorna -1 se houver falha. [tipo] deve ser <i>nint</i> , <i>nfloat</i> , <i>ndfloat</i> , <i>ncplx</i> , <i>ndcplx</i> , <i>string</i> , ou <i>bytes</i> .
int rcv(int msgtipo) - recebe uma mensagem do tipo especificado ou qualquer uma se <i>msgtipo</i> =-1. Retorna o tipo de mensagem corrente.
int rcvinfo(int *bytes, int *msgtipo, char *componente, int *instância) - retorna o comprimento, tipo e remetente do último <i>rcv</i> . Retorna -1 se <i>rcv</i> não foi chamado.
int snd(char *componente, int instância, int msgtipo) - envia mensagem do <i>buffer</i> de envio para a <i>instância</i> de <i>componente</i> especificada. Se <i>instância</i> =-1, então é distribuído para todas as instâncias. Retorna < 0 em caso de erro.

Tabela A.1: Rotinas em C da PVM.

Apêndice B

Sistemas Lineares para Verificação dos Erros

Neste apêndice apresentaremos a descrição dos sistemas lineares de ordem n utilizados nos testes para a determinação dos erros. Estes sistemas tem como solução um vetor cujos elementos são unitários. Cada sistema está descrito pela sua matriz dos coeficientes A e pelo vetor de termos independentes b . Estes sistemas foram obtidos em [42].

- Binomial

$$A = (a_{ij}), \quad a_{ij} = \begin{cases} \binom{i}{j-1} & \Leftrightarrow j \leq i \\ 0 & \Leftrightarrow j > i \end{cases}, \quad b = \begin{pmatrix} 2^1 - 1 \\ 2^2 - 1 \\ \vdots \\ 2^n - 1 \end{pmatrix}$$

- Hilbert

$$A = (a_{ij}), \quad a_{ij} = 1/(i+j-1), \quad b_i = \sum_{j=1}^n 1/(i+j-1), \quad 1 \leq i \leq n, \quad 1 \leq j \leq n$$

- Tridiagonal

$$A = (a_{ij}), \quad a_{ij} = \begin{cases} 2 & \Leftrightarrow i = j \\ -1 & \Leftrightarrow |i-j| = 1 \\ 0 & \text{c/c} \end{cases}, \quad b_i = \begin{cases} 1 & \text{para } i \in \{1, n\} \\ 0 & \text{c/c} \end{cases}$$

Apêndice C

Tabelas e Gráficos

Neste apêndice apresentaremos as tabelas dos números de operações máximo e médio por processador de cada uma das operações elementares e da raiz quadrada, e a tabela do número de operações total dos algoritmos utilizando um único processador.

n	Algoritmo		
	QR -Givens	QR -Cholesky	Csanky
10	10	21	33
20	20	41	49
30	30	61	53
40	40	81	68
50	50	101	73
60	60	121	73
70	70	141	88
80	80	161	
90	90	181	
100	100	201	

Tabela C.1: Tabela do número máximo de adições.

n	Algoritmo				
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky
10	18	18	10	9	36
20	38	38	20	19	76
30	58	58	30	29	116
40	78	78	40	39	156
50	98	98	50	49	196
60	118	118	60	59	236
70	138	138	70	69	276
80	158	158	80	79	316
90	178	178	90	89	356
100	198	198	100	99	396

Tabela C.2: Tabela do número máximo de subtrações.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	18	18	10	19	76	9
20	38	38	20	39	156	11
30	58	58	30	59	236	12
40	78	78	40	79	316	13
50	98	98	50	99	396	14
60	118	118	60	119	476	14
70	138	138	70	139	556	15
80	158	158	80	159	636	
90	178	178	90	179	716	
100	198	198	100	199	796	

Tabela C.3: Tabela do número máximo de multiplicações.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	10	10	10	6	31	3
20	20	20	20	11	61	3
30	30	30	30	16	91	3
40	40	40	40	21	121	3
50	50	50	50	26	151	3
60	60	60	60	31	181	3
70	70	70	70	36	211	3
80	80	80	80	41	241	
90	90	90	90	46	271	
100	100	100	51	199	301	

Tabela C.4: Tabela do número máximo de divisões.

n	Algoritmo	
	QR -Givens	QR -Cholesky
10	5	1
20	10	1
30	15	1
40	20	1
50	25	1
60	30	1
70	35	1
80	40	1
90	45	1
100	50	1

Tabela C.5: Tabela do número máximo de raízes quadradas.

n	Algoritmo
	Gauss(P)
10	25
20	69
30	119
40	177
50	237
60	297
70	363
80	433
90	503
100	573

Tabela C.6: Tabela do número máximo de comparações.

n	Algoritmo		
	QR -Givens	QR -Cholesky	Csanky
10	2,352	21,000	2,259
20	4,863	41,000	2,117
30	7,367	61,000	2,008
40	9,869	81,000	2,055
50	12,371	101,000	2,084
60	14,871	121,000	2,003
70	17,371	141,000	2,088
80	19,872	161,000	
90	22,372	181,000	
100	24,873	201,000	

Tabela C.7: Tabela do número médio de adições por processador em cada algoritmo.

n	Algoritmo				
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky
10	3,750	3,750	4,950	0,102	6,600
20	7,125	7,125	9,975	0,113	13,300
30	10,472	10,472	14,983	0,117	19,978
40	13,813	13,813	19,988	0,119	26,650
50	17,150	17,150	24,990	0,120	33,320
60	20,486	20,486	29,992	0,121	39,989
70	23,821	23,821	34,993	0,121	46,657
80	27,156	27,156	39,994	0,122	53,325
90	30,491	30,491	44,994	0,122	59,993
100	33,825	33,825	49,995	0,123	66,660

Tabela C.8: Tabela do número médio de subtrações por processador em cada algoritmo.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	3,750	3,750	4,950	4,807	29,040	2,039
20	7,125	7,125	9,975	9,839	55,773	2,012
30	10,472	10,472	14,983	14,851	82,460	2,007
40	13,813	13,813	19,988	19,857	109,137	2,004
50	17,150	17,150	24,990	24,860	135,810	2,003
60	20,486	20,486	29,992	29,863	162,480	2,002
70	23,821	23,821	34,993	34,864	189,150	2,002
80	27,156	27,156	39,994	39,866	215,819	
90	30,491	30,491	44,994	44,867	242,487	
100	33,825	33,825	49,995	49,868	269,155	

Tabela C.9: Tabela do número médio de multiplicações por processador em cada algoritmo.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	0,550	0,550	0,550	0,330	7,800	0,031
20	0,525	0,525	0,525	0,351	14,400	0,008
30	0,517	0,517	0,517	0,359	21,044	0,003
40	0,513	0,513	0,513	0,363	27,700	0,002
50	0,510	0,510	0,510	0,365	34,360	0,001
60	0,508	0,508	0,508	0,367	41,022	0,001
70	0,507	0,507	0,507	0,368	47,690	0,001
80	0,506	0,506	0,506	0,369	54,350	
90	0,506	0,506	0,506	0,370	61,015	
100	0,505	0,505	0,505	0,370	67,680	

Tabela C.10: Tabela do número médio de divisões por processador em cada algoritmo.

n	Algoritmo	
	QR -Givens	QR -Cholesky
10	0,102	0,100
20	0,113	0,050
30	0,117	0,033
40	0,119	0,025
50	0,120	0,020
60	0,121	0,017
70	0,121	0,014
80	0,122	0,013
90	0,122	0,011
100	0,123	0,010

Tabela C.11: Tabela do número médio de raízes quadradas por processador em cada algoritmo.

Carga nula em % - Total

Figura C.1: Gráfico das porcentagens de processadores totalmente ociosos considerando o número total de operações realizadas por cada algoritmo para cada valor de n .

Carga abaixo da média em % - Total

Figura C.2: Gráfico das porcentagens de processadores com carga abaixo da média do total de operações por processador.

n	Algoritmo
	Gauss(P)
10	0,550
20	0,548
30	0,537
40	0,536
50	0,532
60	0,526
70	0,525
80	0,524
90	0,523
100	0,521

Tabela C.12: Tabela do número médio de comparações por processador em cada algoritmo.

n	Algoritmo					
	Gauss	Gauss(P)	Gauss-Jordan	QR -Givens	QR -Cholesky	Csanky
10	805	841	1.045	3.340	6.444	19.650
20	5.910	6.081	8.190	25.480	49.789	318.700
30	19.315	19.721	27.435	84.420	166.034	1.617.150
40	45.020	45.761	64.780	198.160	391.179	5.115.000
50	87.025	88.201	126.225	384.700	761.224	12.492.250
60	149.330	151.041	217.770	662.040	1.312.169	25.908.900
70	235.935	238.281	345.415	1.048.180	2.080.014	48.004.950
80	350.840	353.921	515.160	1.561.120	3.100.759	81.900.400
90	498.045	501.961	733.005	2.218.860	4.410.404	131.195.250
100	681.550	686.401	1.004.950	3.039.400	6.044.949	199.969.500

Tabela C.13: Tabela do número de operações total dos algoritmos utilizando um único processador.

Bibliografia

- [1] A. V. Aho, J. E. Hopcroft, e J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Massachusetts, 1975.
- [2] S. G. Akl. *The design and Analysis of Parallel Algorithms*. Prentice-Hall International, New Jersey, 1989.
- [3] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, e V. Sunderan. *A user's guide to PVM - Parallel Virtual Machine*. Oak Ridge National Laboratory.
- [4] S. J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18:147–150, 1984.
- [5] D. P. Bertsekas e J. N. Tsitsiklis. *Parallel and Distributed Computation: numerical methods*. Prentice-Hall International, New Jersey, 1989.
- [6] A. Bojańczyk. Complexity of solving linear systems in different models of computation. *SIAM J. Numerical Analysis*, 21:591–603, 1984.
- [7] A. Borodin. On relating time and space to size and depth. *SIAM J. Computing*, 6:733–744, 1977.
- [8] C. A. Caliulli, H. H. Domingues, e R. C. F. Costa. *Introdução à Álgebra Linear e Aplicações*. Atual Editora, São Paulo, 1990.
- [9] E. Chu e A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.
- [10] S. A. Cook. An observation on time-storage trade off. *J. Computer System Science*, 9:308–316, 1974.
- [11] S. A. Cook. Towards a complexity theory of synchronous parallel computations. *L'Enseignement Mathématique*, 27:99–124, 1981.
- [12] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

- [13] M. Cosnard e M. Daoudi. Optimal algorithms for parallel Givens factorization on a coarse-grained PRAM. *J. ACM*, 41:399–421, 1994.
- [14] M. Cosnard e Y. Robert. Complexity of parallel QR factorization. *J. ACM*, 33:712–723, 1986.
- [15] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Computing*, 5:618–623, 1976.
- [16] K. Datta. Parallel complexities and computations of Cholesky's decomposition and QR factorization. *Inter. J. Computer Math.*, 18:67–82, 1985.
- [17] D. K. Faddeev e V. N. Faddeeva. *Computational Methods of Linear Algebra*. Freeman, San Francisco, 1963.
- [18] M. J. Flynn. Some computer organization and their effectiveness. *IEEE Trans. Computers*, 21:948–960, 1972.
- [19] G. A. Geist, M. T. Heath, e E. Ng. Parallel algorithms for matrix computations. Em L. H. Jamieson, D. B. Gannon, e R. J. Douglas, editores, *The Characteristics of Parallel Algorithms*, pages 233–251. MIT Press, 1987.
- [20] G. A. Geist e C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM J. Sci. Stat. Comput.*, 9:639–649, 1988.
- [21] W. M. Gentleman. Some complexity results for matrix computations on parallel processors. *J. Of ACM*, 25:112–115, 1978.
- [22] G. H. Golub e C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, 1991.
- [23] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20:740–777, 1978.
- [24] K. Hoffman e R. Kunze. *Álgebra Linear*. LTC Editora, São Paulo, 1979.
- [25] O. H. Ibarra e M. H. Kim. Fast parallel algorithms for solving triangular systems of linear equations on the hypercube. *J. Parallel and Distributed Computing*, 20:303–316, 1994.
- [26] J. Jája. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Massachusetts, 1992.
- [27] L. H. Jamieson. Characterizing parallel computations. Em L. H. Jamieson, D. B. Gannon, e R. J. Douglas, editores, *The Characteristics of Parallel Algorithms*, pages 65–100. MIT Press, 1987.

- [28] S. Lakshmivarahan e S. K. Dhall. *Analysis and Design of Parallel Algorithms. Arithmetic and Matrix Problems*. McGraw-Hill Publishing Company, New York, 1990.
- [29] R. E. Lord, J. S. Kowalik, e S.P. Kumar. Solving linear algebraic equations on an MIMD computer. *J. ACM*, 30:103–117, 1983.
- [30] R. Melhem. Parallel Gauss-Jordan elimination for the solution of dense linear systems. *Parallel Computing*, 4:339–343, 1987.
- [31] J. J. Modi. *Parallel Algorithms and Matrix Computations*. Claredon Press, Oxford, 1988.
- [32] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7:101–104, 1987.
- [33] I. Munro e M. Patterson. Optimal algorithms for parallel polynomial evaluation. *J. Computers and System Sciences*, 7:189–198, 1973.
- [34] P. O. A. Navaux. *Processadores Pipeline e Processamento Vetorial*. IME/USP, São Paulo, 1990.
- [35] M. C. Pease. Matrix inversion using parallel processing. *J. ACM*, 14:757–764, 1967.
- [36] M. C. Pease. Inversion of matrices by partitioning. *J. ACM*, 16:302–314, 1969.
- [37] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Manchester University Press, Manchester, 1990.
- [38] A. H. Sameh e R. P. Brent. Solving triangular systems on a parallel computer. *SIAM J. Numerical Analysis*, 14:1101–1113, 1977.
- [39] A. H. Sameh e D. J. Kuck. A parallel QR algorithm for symmetric tridiagonal matrices. *IEEE Trans. Computers*, 26:147–153, 1977.
- [40] A. H. Sameh e J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25:81–91, 1978.
- [41] M. Sloman e J. Kramer. *Distributed Systems and Computer Networks*. Prentice-Hall International, New Jersey, 1987.
- [42] J. M. Stern. *Esparsidade, Estrutura, Estabilidade e Escalonamento em Álgebra Linear Computacional*. UFPE/DI, Recife, 1994.
- [43] G. W. Stewart. A parallel implementation of the QR algorithm. *Parallel Computing*, 5:187–196, 1987.

- [44] R. Terada. *Introdução à Complexidade de Algoritmos Paralelos*. IME/USP, São Paulo, 1990.
- [45] R. Terada. *Desenvolvimento de Algoritmo e Estrutura de Dados*. Markron Books do Brasil, São Paulo, 1991.
- [46] L. G. Valliant. Parallelism in comparison problems. *SIAM J. Computing*, 4:348–355, 1975.
- [47] G. Villard. Exact parallel solution of linear systems. Em J. D. Dora e J. Fitch, editores, *Computer Algebra and Parallelism*, pages 197–205. Academic Press, 1989.
- [48] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Englewood Cliffs, New Jersey, 1963.
- [49] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1972.
- [50] O. Wing e J. W. Huang. A computation model of parallel solution of linear equations. *IEEE Transaction on Computers*, 29:632–638, 1980.