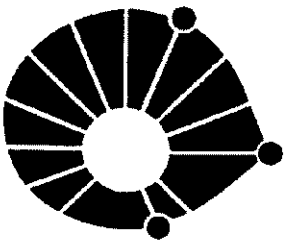


Universidade Estadual de Campinas

LCSI

Faculdade de Engenharia Elétrica e de Computação (FEEC)
Departamento de Máquinas, Componentes e Sistemas Inteligentes
Laboratório de Controle e Sistemas Inteligentes

Uma nova proposta para Solução Computacional da Equação Algébrica de Riccati em formas Sequencial e Paralela

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica

por

Annabell Del Real Tamariz
Licenciada em Informática - UH
Prof. Dr. Celso Pascoli Bottura
Orientador - FEEC/UNICAMP/SP

Aprovada em 15 de março de 1999 pela banca examinadora:

Prof. Dr. Celso Pascoli Bottura (Presidente)
Prof. Dr. Eugenius Kaszkurewicz - UFRJ
Prof. Dr. André L. Morelato França - UNICAMP
Marconi Kolm Madrid - UNICAMP (Suplente)

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

D387n

Del Real Tamariz, Annabell
Uma nova proposta para solução computacional da equação algébrica de Riccati em formas sequencial e paralela. / Annabell Del Real Tamariz.--Campinas, SP: [s.n.], 1999.

Orientador: Celso Pascoli Bottura
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Riccati, Equação de. 2. Programação paralela (Computação). 3. Algebra linear. 4. Transformações (Matemática). I. Bottura, Celso Pascoli. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Uma proposta de metodologia para a solução da *Equação Algébrica de Riccati (EAR)* em formas Sequencial e Paralela e Distribuída é apresentada.

O método modifica e propõe uma paralelização do *Método de Schur* [3]. *Transformações de Similaridade Elementares Estabilizadas(TSEE)* são utilizadas para transformar a matriz simplética/Hamiltoniana, em uma forma simples. Neste trabalho fazemos uma implementação sequencial do algoritmo proposto para matrizes densas, bem condicionadas e propomos uma implementação paralela do algoritmo num sistema com memória distribuída e estrategia de paralelização síncrona numa rede de estações de trabalho.

Abstract

A proposal of methodology for solving the *Algebraic Riccati Equation* in Sequential and Parallel and Distributed forms is presented.

The method modifies and proposes a parallelization for the *Schur Method* [3]. To transform the simpletyc/Hamiltonian matrix in a simple form, *Elementary Stabilized Similarity Transformations* are utilized. In this work a sequential implementation of the proposed algorithm for dense, well conditioned matrices is made and a parallel implementation on a distributed memory system with an synchronous parallelization strategy over a workstation network is proposed.

Agradecimentos

À minha pequena princesa, Natalie, que é a verdadeira autora deste trabalho.
Ao meu esposo Raúl, por alentar-me e sempre estar ao meu lado.
Aos meus pais, que sempre me deram força e coragem para chegar até aqui.
Ao meu orientador, Celso Pascoli Bottura, por ter conduzido tão bem este trabalho, sempre ter confiado em mim e pelos ensinamentos pessoais sobre a vida.
À minha familia em Cuba por apoiar-me em todo momento.
À minha mãe-amiga, Renata, por fazer da sua casa minha casa.
Aos meus amigos-colegas do LCSI: João, Paulo, Gilmar, Maurício, Manoel; que sempre me deram sua ajuda e sofreram comigo todo este tempo.
Aos meus amigos cubanos: Eduardo, Sahudy, Luisma, Marta, Ania e Juan Carlos que, de uma forma ou de outra, me ajudaram para que este trabalho chegasse ao final.
Aos meus amigos brasileiros por fazer do Brasil a minha segunda pátria.
À CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Uma das equações matriciais não lineares estudada mais profundamente e utilizada por engenheiros, economistas, matemáticos e estatisticos é a Equação de Riccati. O termo genérico **"Equação de Riccati"** pode significar qualquer classe de matrizes: quadrática, algébrica ou diferencial (equação diferencial de tipo simétrico ou assimétrico) surgida no estudo de sistemas dinâmicos contínuos ou discretos no tempo [5].

As Equações de Riccati surgem naturalmente numa ampla variedade de situações e desempenham papel muito importante em teoria de sistemas, em especial na teoria de controle desenvolvida nos últimos 40 anos.

Neste trabalho uma proposta de metodologia de solução numérica para a *Equação Algébrica de Riccati* é apresentada. Apesar de aplicável tanto para a Equação de Riccati contínua como para a discreta, nos concentraremos no caso discreto. Estas equações desempenham um papel fundamental na análise, síntese e projeto de sistemas de controle, estimação de estado e modelagem de sistemas dinâmicos lineares, processamento de sinais e fatoração espectral. Nosso objetivo com este trabalho é obter uma única solução simétrica e definida não negativa da equação algébrica e comparar os resultados com os de métodos existentes.

O problema computacional principal surgido na otimização quadrática de sistemas de controle linear é a solução da correspondente equação matricial de

Riccati. Dado que a solução analítica para esta equação só é possível em casos restritos, é necessário em geral implementar algum método numericamente estável.

Do ponto de vista numérico a utilização de autovetores é altamente não satisfatória e o **Método de Schur** é numericamente mais atrativo para obter uma base para certos subespaços de interesse. Porém, neste trabalho implementamos uma variante do método de Schur baseada sempre na idéia seguida por Laub em [3].

1.1 Trabalhos Afins

Um aspecto importante da análise de qualquer problema numérico é o do condicionamento: se os dados do problema sofrem pequenas perturbações, na solução vamos ter mudança muito grande: se o sistema é mal condicionado, ou muito pequena: se o sistema é bem condicionado. Nesta tese nos restringimos às matrizes muito bem condicionadas; com tal limitação em nosso algoritmo, asseguramos a obtenção de uma solução numérica confiável, essencial para o alcançe dos nossos objetivos principais.

Certas equações matriciais surgem naturalmente em Controle Linear e Teoria de Sistemas. Frequentemente encontramos na análise e projeto de sistemas discretos e contínuos no tempo as equações de **Lyapunov**:

$$AXA^T - X + Q = 0 : \textit{Stein}$$

$$AX + XA^T + Q = 0 : \textit{contínua}$$

e as equações de **Sylvester**:

$$AXF - X + Q = 0 : \textit{discreta}$$

$$AX + XF + Q = 0 : \textit{contínua}$$

Existe uma quantidade de abordagens apreciável em controle e teoria de sistemas lineares para solucionar as mesmas. Partindo da análise profunda

destas equações, podemos compreender muito melhor o condicionamento da equação de **Riccati**, para a qual existe uma literatura considerável no aspecto teórico, porém muito limitada do ponto de vista numérico. Contudo não enfatizaremos no aspecto do condicionamento nesta tese.

As **Equações Algébricas Simétricas de Riccati** $(n \times n)$ têm as seguintes formas para os casos de sistemas contínuos e discretos no tempo, respectivamente:

$$AX + XA^T - XGX + Q = 0 \tag{1.1}$$

$$AXA^T - X - A^T XG_1(G_2 + G_1^T XG_1)^{-1}G_1^T XA + Q = 0 \tag{1.2}$$

onde $A, Q, X \in R^{n\times n}$; $G_1 \in R^{n\times m}$; $G_2 \in R^m$

Sempre são adotadas restrições sobre os coeficientes das matrizes para garantir a existência e/ou unicidade de certos tipos de solução X.

Vários algoritmos foram propostos para resolver as equações (1.1 e 1.2), dentre os quais podemos citar [3, 2, 19, 1]. Estes algoritmos podem ser classificados em duas categorias [30] :

Métodos iterativos: Produzem uma sequência de matrizes similares que convergem à solução; em geral são numericamente precisos mas sua aplicação pratica é bastante limitada.

Métodos de subespaço invariante: Têm um grande alcançe de aplicabilidade e são métodos numericamente estáveis.

Dos vários algoritmos que existem para solucionar a referida equação, um dos métodos de propósito geral mais confiáveis é o **Método de Schur** [3], definido como um método de subespaço invariante. Por exemplo, para o caso da equação (1.2), o método baseia-se na redução à **Forma Real de Schur (FRS)** da matriz simplética H associada:

$$H = \begin{bmatrix} A + GA^{-T}Q & -GA^{-T} \\ -A^{-T}Q & A^{-T} \end{bmatrix} \tag{1.3}$$

onde $G = G_1 G_2^{-1} G_1^T$. H está caracterizada por $J^{-1} H^T J = H^{-1}$, onde J é uma matriz $(2n \times 2n)$ definida como:

$$J = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix}$$

e I é uma matriz identidade $(n \times n)$.

Como é exposto no conhecido Teorema 1,[3], os autovalores de uma matriz Simplética **H** ocorrem em pares $(\lambda,\ 1/\lambda)$; baseado nesta informação e com ajuda de outros procedimentos, realizamos o processo de obtenção da solução simétrica, definida não negativa da equação algébrica de Riccati discreta.

1.1.1 Características da solução numérica de Laub

O método de resolução da equação algébrica de Riccati proposto por Laub, tem dois passos fundamentais; no primeiro reduz uma matriz $H \in R^{2n \times 2n}$ à forma real de Schur ordenada, e no segundo calcula a solução de uma equação matricial linear de ordem n.

Para resolver o seu algoritmo, Laub realiza o procedimento a seguir:

1- Faz um balanceamento da matriz geral real (Precondições).

2- Reduz a matriz a forma superior de Hessenberg utilizando transformações ortogonais.

3- Acumula as transformações da redução de Hessenberg.

4- Determina a forma real de Schur ordenada da matriz de Hessenberg obtida anteriormente.

5- Faz uma transformação para trás da matriz ortogonal a uma matriz não singular correspondente à matriz original.

6- Encontra a solução da equação a partir da matriz não singular obtida no passo 5.

No que se refere a implementação, Laub usa uma matriz simplética/Hamiltoniana H formada a partir dos dados de entrada do problema. Depois de realizar o balanceamento da matriz H, Laub utiliza uma norma definida como a raiz quadrada da maior somatória das magnitudes absolutas dos elementos da matriz H, por colunas, mais um, que denomina *alpha*. Com este valor *alpha* redefine duas novas matrizes da seguinte forma:

W: Definida como a diagonal da matriz original menos o valor *alpha* calculado;

Z: Definida como a diagonal da matriz original mais o valor *alpha* calculado;

Nós testamos um programa implementado em nosso laboratório, com as mesmas subrotinas que Laub propõe, e a solução obtida - ver definição a seguir - satisfaz as condições que Laub estabelece no seu artigo [3].

```
1- Ler Dados
2- Formar a matriz Hamiltoniana/Simplética
3- Executar a subroutina BALANC
4- Definir as duas matrizes W e Z
5- Executar a subroutina ORTHES e obter uma matriz em forma
   Superior de Hessenberg
6- Executar a subroutina ORTRAN que acumula as transforma-
   ções de redução
7- Executar a subroutina HQR3 para obter a matriz em forma
   Real de Schur e uma matriz de transformação.
8- Calcular a solução da equação de Riccati contínua.
```

A seguir apresentamos um exemplo para resolver uma Equação Algébrica de Riccati contínua utilizando o programa implementado em nosso laboratório:

```
 Matriz Hamiltoniana
    0.0158      0.7338     -0.7672     -0.1007
```

```
   0.7277     0.0679    -0.1007    -0.7353
  -0.7975    -0.2107    -0.0158    -0.7277
  -0.2107    -0.4533    -0.7338    -0.0679
Matriz W formada
   2.3077    -0.7338     0.7672     0.1007
  -0.7277     2.2556     0.1007     0.7353
   0.7975     0.2107     2.3393     0.7277
   0.2107     0.4533     0.7338     2.3914
Matriz Z formada
   2.3393     0.7338    -0.7672    -0.1007
   0.7277     2.3914    -0.1007    -0.7353
  -0.7975    -0.2107     2.3077    -0.7277
  -0.2107    -0.4533    -0.7338     2.2556
Saída da Sub HQR3:
Matriz A - Triangular
   3.0062    -0.3284     0.0508    -0.0135
   0.0000     2.1336     0.2574     0.0641
   0.0000     0.0000     0.4687     0.0512
   0.0000     0.0000     0.0000     0.3326
Matriz V  Transformações
   0.3669    -0.6308     0.1008    -0.6763
   0.2006     0.7263    -0.2538    -0.6065
   0.6763    -0.1008    -0.6308     0.3669
   0.6065     0.2538     0.7263     0.2006

Matriz S em Forma Real de Schur
  -1.1636     0.1216    -0.0614     0.0082
   0.0000    -0.8406    -0.2599    -0.0614
   0.0000     0.0000     0.8406    -0.1216
   0.0000     0.0000     0.0000     1.1636
Solução --- E. A. de Ricatti Contínua ---
```

```
                1.3012    0.9913
                0.9913    1.2102
Teste => AtX + XA - XSX + Q = 0
                0.0000    0.0000
                0.0000    0.0000
```

Em seguida fizemos alterações na definição das matrizes W e Z, adotando como Z a matriz simplética/Hamiltoniana H e W como uma matriz identidade. Os resultados foram totalmente diferentes e não satisfizeram as condições propostas no artigo de Laub. A partir disto, concluímos que o passo fundamental do algoritmo proposto por Laub, na nossa forma de ver, consiste na definição das matrizes W e Z, pois se eliminamos essa definição inicial de matrizes, nunca obtemos a Forma Real de Schur Ordenada que Laub propõe, utilizando sempre o algoritmo por ele proposto.

1.1.2 Características da solução numérica de Bartels e Stewart

Outro dos algoritmos propostos na literatura que foram estudados, foi o *Algoritmo 432* de Bartels e Stewart, [26]. Neste algoritmo estão implementadas uma coleção de subrotinas para resolver a equação matricial simétrica:

$$AX + XA^T = C \tag{1.4}$$

A matriz A é reduzida a Forma Real de Schur Superior A' por uma transformação de similaridade ortogonal U, resultando:

$$A' = U^T A U$$

$$C' = U^T C U$$

$$X' = U^T X U$$

e a equação transformada obtida é resolvida por um procedimento recursivo. A seguir apresentamos uma breve descrição do algoritmo utilizado:

1- Reduzir a matriz A a forma superior de Hessenberg via transformações hermitianas elementares (método de Householder). Na saída a matriz A contém:

   - Um triângulo superior e uma coluna a mais com os elementos da subdiagonal da matriz transformada;
   - Um triângulo inferior e uma linha a mais com os resultados dos cálculos das transformações.

2- Calcular a partir da matriz anterior uma matriz ortogonal U que reduz a matriz original à forma superior de Hessenberg.

3- Achar a Forma Real de Schur superior de uma matriz em forma de Hessenberg via método QR; esta subrotina acumula o produto das transformações usadas na redução. Em particular, a subrotina utilizada neste passo é considerada ineficiente para calcular os autovalores da matriz de Hessenberg superior.

4- Finalmente, utilizar uma outra subrotina para resolver o sistema de equações lineares e dar a solução da respectiva equação matricial.

Modificando um pouco as subrotinas propostas no artigo [26] para resolver a equação algébrica de Riccati contínua, definida como:

$$AX + XA^T - XGX + Q = 0 \tag{1.5}$$

comprovamos que podemos obter a mesma matriz de Hessenberg, e a mesma matriz de transformação obtida pelo MATLAB, para uma matriz Simplética original. Nosso objetivo em utilizar o programa proposto por Bartels e Stewart foi satisfeito, pois temos mais um teste de comprovação para nossa proposta de obtenção da matriz em forma superior de Hessenberg, com a qual obtemos por transformações ortogonais uma matriz em forma real de Schur.

A seguir apresentamos o mesmo exemplo apresentado anteriormente, para comprovar os resultados, utilizando o algoritmo modificado de Bartels e Stewart:

```
Matriz Hamiltoniana
     0.0158       0.7338      -0.7672      -0.1007
     0.7277       0.0679      -0.1007      -0.7353
    -0.7975      -0.2107      -0.0158      -0.7277
    -0.2107      -0.4533      -0.7338      -0.0679
```

O passo seguinte é executar o programa implementado que calcula a matriz em Forma Real de Schur e devolve a matriz de transformação correspondente.

```
Execucao da subrotina ATXPXA modificada que contém chamadas a
outras subrotinas que ajudam na solução:
 Matriz obtida da subrotina HSHLDR
    0.0158    -1.0610    -0.0991     0.0415    -1.1000
    2.2918     0.1159     0.2207     0.1703     0.3373
   -1.0000    -1.7765    -0.9085    -0.2210    -0.4424
   -0.2642    -1.0000    -0.4424     0.7767     0.0000
    3.1610     2.0780     0.0000     0.0000     0.0000
 Matriz obtida da subrotina BCKMLT
 Matriz U de Transformação Ortogonal
    1.0000     0.0000     0.0000     0.0000
    0.0000    -0.6616    -0.5399    -0.5205
    0.0000     0.7250    -0.2832    -0.6278
    0.0000     0.1915    -0.7927     0.5788
 Matriz de Hessenberg calculada
    0.0158    -1.0610    -0.0991     0.0415
   -1.1000     0.1159     0.2207     0.1703
    0.0000     0.3373    -0.9085    -0.2210
    0.0000     0.0000    -0.4424     0.7767
```

```
Matriz Forma Real de Schur
   1.1636     0.0314    -0.0325     0.1163
   0.0000    -1.1636    -0.1215     0.0561
   0.0000     0.0000    -0.8406    -0.2669
   0.0000     0.0000     0.0000     0.8406
Matriz U de Transformações de Schur
  -0.6741    -0.3760    -0.6193    -0.1438
  -0.4631    -0.2069     0.7344    -0.4511
   0.5728    -0.6686    -0.1104    -0.4611
  -0.0555    -0.6073     0.2549     0.7504

Solução --- E. A. de Ricatti Contínua ---
              12.3522    -19.2177
               7.7821    -11.2082
Teste => AtX + XA - XSX + Q = 0
               0.0001   -0.0001
               0.0000    0.0000
```

A programação deste algoritmo para resolver a equação algébrica de Riccati apresenta um problema relacionado com a ordenação da matriz em Forma Real de Schur: como pode-se observar esta matriz tem os elementos na diagonal principal (que correspondem aos autovalores) desordenados.Em consequência não estamos obtendo a solução desejada.

1.2 Processamento Paralelo e Distribuído

Põe-se a seguinte questão: Por que usar processamento paralelo? A resposta está na busca de soluções mais rápidas e precisas e na busca de maior capacidade de processamento.

Nos anos recentes, processamento paralelo e distribuído têm se apresentado como meios bastante promissores para atender as exigências computacionais do futuro. Significativos avanços em algoritmos paralelos e arquiteturas

têm permitido mostrar o potencial das técnicas de computação concorrente numa ampla variedade de problemas.

O contínuo desenvolvimento de pesquisas em diferentes ramos da ciência tem provocado um aumento na demanda por poder computacional, na medida em que aplicações com requerimentos cada vez maiores precisam ser executadas. Para atender esta crescente demanda, pesquisadores na área de computação procuram soluções, tanto em termos de hardware como de software, que viabilizem aplicações com desempenhos cada vez melhores. Uma boa parte das pesquisas conduzidas até hoje para melhorar a performance das aplicações se concentra na área de processamento paralelo. O processamento paralelo baseia-se na percepção de que muitas aplicações podem ser divididas em tarefas menores que podem ser executadas em paralelo usando vários processadores. Uma aplicação composta por várias tarefas que podem ser executadas em paralelo é chamada de *Aplicação Paralela*; ou seja uma *Aplicação Paralela* corresponde a uma coleção de componentes paralelas operando de forma cooperativa para resolver um dado problema.

Os sistemas de Computação Paralela e Distribuída incluem várias áreas da computação, tais como: O projeto de máquinas paralelas, linguagens de programação paralela, desenvolvimento e análise de algoritmos paralelos, e suas aplicações.

Existem várias propostas para processamento paralelo usando computadores conectados em rede, dentre estas ressaltamos:

O PVM (Parallel Virtual Machine) [14] é um sistema que permite simular um computador paralelo usando máquinas conectadas em uma rede. Este computador paralelo é chamado de máquina paralela virtual, e é uma implementação muito simples e eficiente, baseada em troca de mensagens. Atualmente existem implementações do PVM em muitas arquiteturas, tanto para estações de trabalho conectadas a uma rede quanto para computadores paralelos reais, tornando as aplicações que usam PVM extremamente portáveis.

O MPI (Message Passing Interface) [28] é um sistema com objetivos simi-

lares ao PVM. Enquanto o PVM foi um projeto de pesquisa que evoluiu até virar um padrão de fato, o MPI surgiu do esforço de um grupo de engenheiros para desenvolver um padrão de troca de mensagens com o objetivo de aumentar a portabilidade de aplicações paralelas. O MPI tem sido implementado em várias arquiteturas paralelas, e existem algumas implementações que trabalham em conjuntos de computadores conectados por uma rede. Uma das principais limitações do MPI é que não especifica como implementações diferentes podem cooperar na solução de um problema. Isto limita seriamente a sua interoperabilidade.

Neste trabalho pretende-se mostrar uma proposta de metodologia para solucionar a *Equação Algébrica de Riccati Discreta* assim como a implementação que realizamos do método de Schur-Modificado em formas sequencial e paralela.

1.3 Características da Implementação Paralela de Bunse-Gerstner e Fabbender

No artigo [1] apresenta-se uma proposta de implementação para resolver em forma paralela a *Equação Algébrica de Riccati Contínua.* Este algoritmo, um *Método Jacobi-Similar*, constitui uma adaptação do *Método de Jacobi* para matrizes Hamiltonianas não simétricas.

É conhecido por estudos anteriores, que o Método de Jacobi é considerado um candidato para resolver o problema de autovalores para matrizes não simétricas com computação paralela, mas para o caso da computação sequencial o método de Jacobi não deve ser recomendado [9].

Bunse-Gerstner e Fabbender, na sua implementação utilizam transformações de similaridade simpléticas unitárias as quais preservam a estrutura Hamiltoniana da matriz, ou seja o método Jacobi-similar restringe-se à escolha de transformações ortogonais/unitárias que sejam ao mesmo tempo simpléticas. Cada passo da iteração necessita só informação local sobre a

matriz atual, o que possibilita uma implementação paralela eficiente sobre certas arquiteturas paralelas.

No método de solução proposto no artigo [1] para resolver a equação algébrica de Riccati contínua:

$$A^H X + XA - XGX + Q = 0$$

onde $A, Q, G, X \in C^{n \times n}$; $G = G^H$ e $Q = Q^H$, calcula-se a decomposição de Schur de uma matriz Hamiltoniana criada com os dados de entrada do problema. Neste método similarmente ao método proposto por Eberlein em [9], transforma-se iterativamente uma matriz arbitraria à forma de Schur por transformações de similaridade com rotações de Givens.

Similarmente ao método de Jacobi, um passo elementar no método implementado proposto para uma matriz H $(n \times n)$ consiste na escolha de um par de índices (k, l), e nas transformações da matriz H tal que a entrada (k, l) seja anulada por transformações de similaridade com rotações U no plano-(k, l). A matriz U difere da matriz identidade só nos quatro elementos u_{kk}, u_{ll}, u_{kl} e u_{lk}. Estas quatro entradas essenciais formam uma matriz unitária que transforma a correspondente submatriz (4×4) de H à forma de Schur. Ou seja, Bunse-Gerstner e Fabbender adaptam esta estratégia para calcular a forma Hamilton-Schur de uma matriz Hamiltoniana. O algoritmo completo acha-se em [1].

Em relação à implementação paralela, em cada passo da iteração para calcular a forma Hamilton-Schur, só 16 elementos da matriz H são necessários para calcular a seguinte transformação U_{ij}, e só 4 linhas e colunas são envolvidas em cada transformação.

Tem-se uma matriz Hamiltoniana $H \in C^{2n \times 2n}$; em cada iteração vamos resolver um subproblema (4×4). Dado H:

$$H = \begin{bmatrix} A & G \\ Q & -A^H \end{bmatrix}$$

onde $A, G, Q \in C^{n \times n}$, formamos subproblemas de tamanho (4×4).

Por exemplo, eliminando os elementos a_{21}, q_{11}, q_{12}, q_{21}, q_{22} da submatriz $H_1 \in C^{4\times 4}$:

$$H_1 = \begin{bmatrix} a_{11} & a_{12} & g_{11} & g_{12} \\ a_{21} & a_{22} & g_{21} & g_{22} \\ q_{11} & q_{12} & -\hat{a}_1 & -\hat{a}_{12} \\ q_{21} & q_{22} & -\hat{a}_{21} & -\hat{a}_{22} \end{bmatrix}$$

afetamos só as linhas e colunas $1, 2, n+1, n+2$ de H. Em particular, os elementos em A, Q, G com índices (3,3), (3,4), (4,3) e (4,4) não são afetados. Logo, é possível calcular a transformação $\hat{U}_{34}$ para anular os elementos: a_{43}, q_{33}, q_{34}, q_{43}, q_{44} ao mesmo tempo que a transformação $\hat{U}_{12}$ anula apenas: a_{21}, q_{11}, q_{12}, q_{21}, q_{22}. Assim, se a transformação $\hat{U}$ é escolhida tal que $\hat{U} = \hat{U}_{12}\hat{U}_{34}$, então $\hat{U}^H H \hat{U}$ deve zerar os elementos em posição (2,1), (4,3), (n+1,1), (n+2,1), (n+1,2), (n+2,2), (n+3,3), (n+4,3), (n+3,4), (n+4,4), (n+1,n+2) e (n+3,n+4). Mas é impossível calcular $\hat{U}_{12}$ e $\hat{U}_{13}$ ao mesmo tempo pois ambas transformações afetam a primeira linha e coluna de H.

Esta estratégia pode ser representada em um arranjo de N processadores, Figura 1.1, onde a comunicação se dá apenas entre os processadores vizinhos.

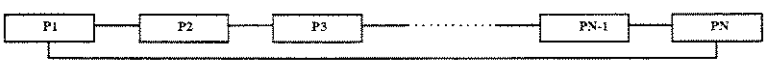


Figura 1.1: Arranjo de Processadores

Inicialmente cada processador P_i têm um índice pivot (k_i, l_i) e as correspondentes quatro colunas de H $(k_i, l_i, n+k_i, n+l_i)$. Cada processador P_i calcula a transformação U_{k_i,l_i}, a qual transforma a matriz $H_{k_i l_i}$ correspondente ao índice pivot (k_i, l_i) à forma Hamilton-Schur. O passo seguinte é transformar a matriz H:

$$\hat{H} = U_N^H U_{N-1}^H ... U_1^H H U_1 U_2 ... U_N$$

onde $U_i = U_{k_i l_i}$. A multiplicação pelo lado esquerdo só afeta as linhas k_i e l_i da matriz H e a multiplicação pelo lado direito só afeta as colunas k_i e l_i.

Todos os métodos até aqui expostos têm a característica comum de utilizar Transformações de Similaridade Ortogonais para achar a solução da equação de Riccati: Eles sempre trabalham com matrizes reais e simétricas, logo os autovetores correspondentes são reais e ortonormais, e a matriz de transformação determinada é ortogonal; assim a Transformação é chamada de Transformação de Similaridade Ortogonal. Pretendemos utilizar uma outra técnica numérica existente para obter a solução da equação de Riccati: *Transformações de Similaridade Elementares Estabilizadas (TSEE)*, pois vamos trabalhar fundamentalmente com matrizes simpléticas não simétricas que podem gerar matrizes de transformação não necessariamente reais, [31].

Este trabalho propõe uma metodologia para a solução da Equação Algébrica de Riccati nas formas sequencial e paralela, que difere dos algoritmos acima descritos, em que nós simplesmente trabalhamos com a matriz Simplética (Hamiltoniana) formada a partir dos dados do problema. As principais contribuições deste trabalho são: A proposta de solução utilizando uma nova metodologia, a implementação das *Transformações de Similaridade Elementares Estabilizadas*, bem como a obtenção da *Forma Real de Schur*, e a proposta de implementação paralela apresentada.

O restante do texto está organizado como segue. O capítulo 2 descreve alguns aspectos da solução numérica adotados na literatura, assim como o método de Schur-Modificado proposto neste trabalho para resolver a equação algébrica de Riccati. O capítulo 3 introduz algumas definições relacionadas com a computação paralela e distribuída, fazemos uma classificação de sistemas distribuídos, apresentamos os fundamentos da computação paralela, uma classificação das arquiteturas paralelas mais utilizadas na prática assim como os modelos de programação de alguns sistemas existentes. No capítulo 4 apresentamos uma implementação paralela de nosso algoritmo. Finalmente, o capítulo 5 contém as conclusões e comentários deste trabalho, bem como sugestões de alguns pontos a serem ainda pesquisados.

Capítulo 2

Um Método de Schur-Modificado para solução da Equação Algébrica de Riccati

Uma nova proposta de metodologia é apresentada para a solução da *Equação Algébrica de Riccati Discreta*:

$$AXA^T - X - A^T X G_1 (G_2 + G_1^T X G_1)^{-1} G_1^T X A + Q = 0$$

para a qual existe, sob certas condições, uma solução simétrica, definida não negativa $X \in R^{n \times n}$, [3].

Na nossa implementação, a matriz simplética H do sistema é transformada à Forma Superior de Hessenberg **(FSH)**, ou seja, reduzimos (H) a uma matriz (Hes) via alguma transformação tal que $hes_{ij} = 0$ para todo $i > j+1$.

2.1 Características Numéricas Gerais

Uma Transformação de Similaridade de uma matriz geral (H) é definida como:

$$H \rightarrow N^{-1}HN \tag{2.1}$$

para alguma matriz de transformação N. Transformações de Similaridade têm uma função importante no cálculo dos autovalores, porque elas mantém os autovalores da matriz original inalterados.

Para matrizes reais, simétricas, a matriz de transformação é ortogonal. A transformação de similaridade é então uma transformação ortogonal na forma:

$$H \rightarrow N^{T}HN \tag{2.2}$$

Para matrizes reais não simétricas, a matriz de transformação não é necessariamente real. A estratégia de todas as rotinas de autosistemas modernas para reduzir a matriz simplética H à forma diagonal é utilizar uma sequência de transformações de similaridade do tipo:

$$H \rightarrow P_1^{-1}HP_1 \rightarrow P_2^{-1}P_1^{-1}HP_1P_2 \rightarrow P_3^{-1}P_2^{-1}P_1^{-1}HP_1P_2P_3 \rightarrow etc. \tag{2.3}$$

Finalmente, o produto das matrizes P_i forma uma matriz de transformação acumulada cujas colunas são os autovetores da matriz original. Para o problema numérico gerado, só precisamos obter uma matriz mais simples e então nela realizar um processo iterativo. Tal estrutura simples, utilizada neste trabalho, é a chamada **Forma de Hessenberg**. A matriz em *Forma Superior de Hessenberg (FSH)* tem zeros em todos os elementos embaixo da diagonal, exceto na primeira subdiagonal, ou seja é uma matriz (Hes) tal

que $hes_{ij} = 0$ para todo $i > j + 1$.

$$Hes = \begin{bmatrix} x & x & x & x & x & x \\ x & x & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \end{bmatrix} \tag{2.4}$$

Existem dois conjuntos de técnicas diferentes para implementar as transformações de similaridade apresentadas na expressão (2.3).

- O primeiro conjunto constrói transformações explícitas individuais P_i destinadas a realizar alguma tarefa específica; por exemplo: podemos zerar um elemento particular fora da diagonal (Transformação Jacobi), ou zerar os elementos de uma linha ou coluna particular (Métodos de Eliminação ou Transformações Householder). Em geral, uma sequência finita destas transformações não triangulariza completamente a matriz. Temos então duas escolhas:

  1. Utilizar uma sequência finita de transformações que levem a uma forma especial (tridiagonal ou Hessenberg) e depois utilizar o segundo conjunto de técnicas expostas a seguir;

  2. Iterar uma sequência finita de transformações simples até obter um desvio da matriz da diagonal que seja desprezível. Este método é computacionalmente ineficiente quando N é maior que 10 aproximadamente.

- O segundo conjunto denomina-se *Métodos de Fatoração.* Suponha que a matriz A pode ser fatorada num fator esquerdo F_L e num fator direito F_R. Então

$$A = F_L F_R$$

ou equivalentemente

$$F_L^{-1} A = F_R$$

Das expressões acima obtemos:

$$F_R F_L = F_L^{-1} A F_L \tag{2.5}$$

que é conhecida como uma transformação de Similaridade da matriz A com a matriz de transformação F_L . Os métodos de fatoração não convergem num número finito de transformações, mas convergem rápida e confiavelmente depois de uma redução inicial por transformações de similaridade simples.

Transformações de uma matriz

Os algoritmos para transformações de matrizes simétricas são altamente satisfatórios na prática. Em contraposição, é mais difícil desenvolver algoritmos igualmente satisfatorios para transformar matrizes não-simétricas. Existem duas razões para isto:

- Os autovalores de uma matriz não simétrica podem ser muito sensíveis a pequenas variações nos elementos da matriz.
- A matriz propriamente dita pode ser defectiva, tal que não exista um conjunto completo de autovetores.

Balanceamento

A sensibilidade dos autovalores a erros por truncamento na execução de alguns algoritmos pode ser reduzida pelo procedimento de balanceamento. Os erros para os autosistemas determinados com procedimentos numéricos são geralmente proporcionais à norma euclidiana da matriz, ou seja, à raiz quadrada da soma dos quadrados dos elementos. Através do balanceamento procura-se igualar as ordens de magnitude dos elementos de uma matriz,

para isto transformações de similaridade são utilizadas para fazer que linhas e colunas da matriz tenham normas comparáveis; desta forma reduzimos a norma geral da matriz e mantemos os autovalores inalterados. No caso das matrizes simétricas o processo de balanceamento não é necessário; mas no caso das matrizes não simétricas é recomendado fazer tal procedimento.

Pensando nisto, é que restringimos nossos dados a matrizes muito bem condicionadas e com ordens de magnitude nos elementos iguais, pois trabalhamos com matrizes não simétricas. Neste trabalho não fazemos análise do balanceamento nem do condicionamento das matrizes; só expomos as restrições que devem satisfazer os dados para que uma solução simétrica, definida não negativa da equação algébrica de Riccati seja obtida.

2.2 Algoritmo de Schur Modificado

O algoritmo proposto e implementado neste trabalho segue basicamente o mesmo fluxo de dados e idéias que o Método de Schur [3]. A ordem dos cálculos das diferentes matrizes necessárias no algoritmo foi mantida, ou seja, primeiro formamos a matriz simplética (para o caso discreto) ou Hamiltoniana (para o caso contínuo) do sistema; achamos uma matriz mais simples em forma de Hessenberg e a partir dela calculamos uma matriz de transformação que conduz a uma matriz em forma real de Schur (FRS). Esta matriz de transformação leva à solução simétrica, definida não negativa da Equação Algébrica de Riccati.

Nossa modificação está na forma de obter a primeira matriz de transformação que produz uma matriz em forma superior de Hessenberg. A redução à forma de Hessenberg pode ser obtida de maneira estável utilizando tanto matrizes elementares estabilizadas como matrizes elementares unitárias.

Nosso trabalho utiliza à redução com matrizes elementares estabilizadas. O método transforma a matriz em questão em outra com os mesmos autovalores, equação (2.6).

$$Hes = N^{-1}HN \tag{2.6}$$

onde N é uma matriz não-singular e não-ortogonal da mesma ordem que H.

A seguir resumiremos o algoritmo para a obtenção da matriz de transformação e da correspondente matriz de Hessenberg, a partir das $TSEE$, apresentando uma descrição passo-a-passo do procedimento.

Parte-se da formação de uma matriz simplética $H \in R^{2n\times 2n}$, não simétrica, a partir dos dados de entrada do problema; a seguir aplicamos as $TSEE$ na matriz simplética para obter uma matriz em forma superior de Hessenberg; junto com esta matriz é formada uma matriz de transformação que nos guia à solução final de Equação Algébrica de Riccati; depois disto obtemos uma matriz em Forma Real de Schur utilizando subrotinas públicas definidas na biblioteca *Scalapack*.

O algoritmo, em pseudocódigo Fortran, para achar a matriz em forma superior de Hessenberg Hes utilizando $TSEE$ pode ser esquematizado como segue, tendo em conta que antes do $k_{ésimo}$ passo a matriz $H = H_1$ foi reduzida à forma de Hessenberg H_k em suas primeiras $k-1$ colunas:

```
DO k = 1, N-2
   1- Inicializar I, T, Tinv, Q, Qinv como matrizes identidade
   2- Achar o maior elemento em módulo da k-ésima coluna,
      começando pelo primeiro elemento embaixo da diagonal.
      Suponhamos que o maior elemento foi denotado por k'
   3- Trocar as linhas k' e (k+1). Para que a matriz de
      permutação seja uma transformação de similaridade,
      também trocamos as colunas k' e (k+1).
   4- Para cada valor i=k+2 até N, calcular o multiplicador:
```

$$n_{i,k+1} = H_{ik}^{(k)} / H_{(k+1),k}^{(k)}$$

```
      4.1- A linha (k+1) vezes o valor do multiplicador calcu-
           lado acima é subtraida da linha i. Para que a ope-
           ração seja uma transformação de similaridade, o
           valor do multiplicador calculado acima, vezes a
```

```
            coluna i, é adicionado a coluna (k+1).
   5- Formar uma matriz de permutação I e uma matriz de trans-
      formação Tinv utilizada para eliminar os elementos apro-
      priados na matriz original.
   6- Criar uma matriz de transformação final Q tal que:
      Q = Tinv * I
   7- Criar uma matriz de transformação final Qinv tal que:
      Qinv = Iinv * T
   8- No final realiza-se Qinv * Hes * Q = H
ENDDO
```

Expressando o procedimento anterior de modo mais sintético, podemos escrever:

- Determinar o máximo valor da $k_{ésima}$ coluna para $k = 1, n-2$:

  $$max|H_{ik}^{(k)}| = |H_{k'k}^{(k)}| \qquad i = k+1, ..., n$$

  onde $k^{'}$ corresponde a linha onde foi achado a maior elemento; com isto se garante que a transformação seja numericamente estável.

- Trocar linhas e colunas com índice k'e (k+1)

- Para cada valor i=k+2 até n, calcular:
  $n_{i,k+1} = H_{ik}^{(k)} / H_{k+1,k}^{(k)}$

  - Para cada valor j=1 até n, calcular:
    $H_{i,j} = H_{i,j} - n_{i,k+1} * H_{k+1,j}$
    $H_{j,k+1} = H_{j,k+1} + n_{i,k+1} * H_{j,i}$

  Se $n_{i,k+1} = 0$ estas duas operações não são realizadas.

Com isto podemos expressar a relação entre as matrizes H_k e H_{k+1} na forma:

$$H_{k+1} = Tinv_{k+1} I_{k+1,k'} H_k Iinv_{k+1,k'} T_{k+1} \tag{2.7}$$

onde $I_{k+1,k'}$ é uma matriz de permutação elementar, ver apêndice C, e T_{k+1} é uma matriz de transformação utilizada para eliminar os elementos apropriados na coluna k; ela é uma matriz identidade onde os elementos abaixo da diagonal são adicionados na coluna $k+1$:

$$(T_{k+1})_{i,k+1} = n_{i,k+1} \qquad i = k+2, ..., n \tag{2.8}$$

e

$$T_1 T_2 ... T_{n-2} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & n_{32} & 1 & & & \\ \cdots & \cdots & \cdots & & & \\ & n_{n-1,2} & n_{n-1,3} & \cdots & 1 & \\ & n_{n,2} & n_{n,3} & \cdots & n_{n,n-1} & 1 \end{bmatrix} = T$$

$$Tinv_1 Tinv_2 ... Tinv_{n-2} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & -n_{32} & 1 & & & \\ \cdots & \cdots & \cdots & & & \\ & -n_{n-1,2} & -n_{n-1,3} & \cdots & 1 & \\ & -n_{n,2} & -n_{n,3} & \cdots & -n_{n,n-1} & 1 \end{bmatrix} = Tinv$$

A Figura 2.1 apresenta as operações em linha e coluna durante a $k_{ésima}$ iteração.

A seguir encontra-se parte da listagem do programa implementado em forma sequencial para achar a matriz em forma superior de Hessenberg (Hes) utilizando $TSEE$. O programa completo foi implementado em Fortran 77, utilizando subrotinas da biblioteca pública Scalapack, e executado numa estação de trabalho Sun do Laboratório de Controle e Sistemas Inteligentes da Faculdade de Engenharia Elétrica e de Computação da UNICAMP.

```
DO K=1,n-2
   CALL INICIALIZA(I,T,Tinv,posic)
```

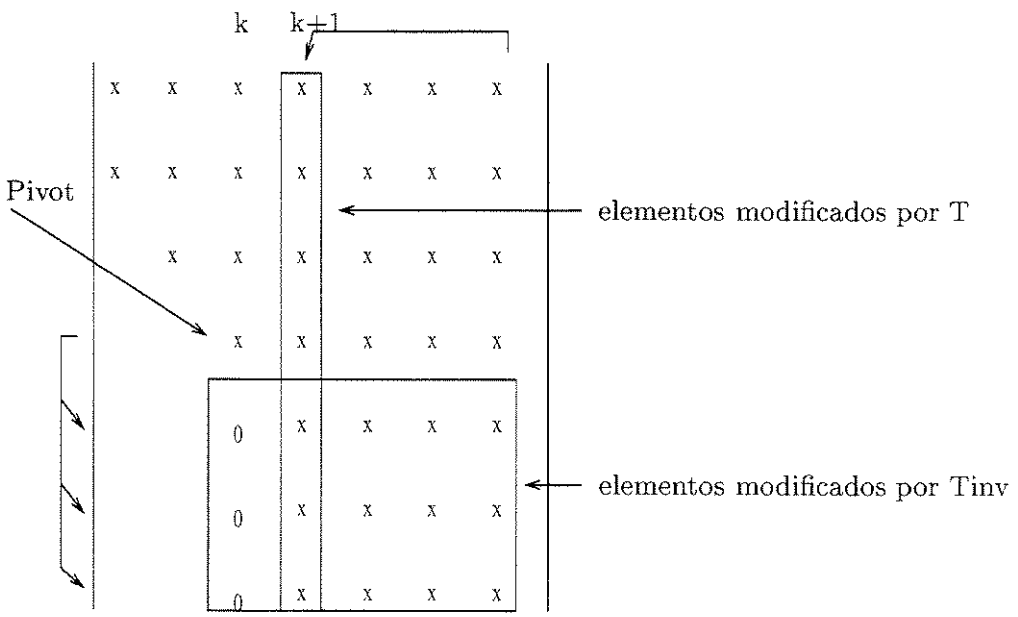


Figura 2.1: Transformações elementares para eliminar os elementos na coluna k

```
CALL CALCULO(Z1,piv,K,posic)
CALL CREAR(Z1,T,Tinv,piv,K)
CALL ATUALIZA(posic,I)
DO l = 1,n
   DO j = 1,n
      IT(j,l) = I(l,j)
   ENDDO
ENDDO
call vezes(Tinv,I,Q11,nz,nz,nn)
call vezes(Q11,Q,QA,nz,nz,nn)
DO l = 1,n
   DO j = 1,n
      Q(l,j) = QA(l,j)
   ENDDO
```

```
      ENDDO
      call vezes(IT,T,PH1,nz,nz,nn)
      call vezes(Qinv,PH1,QA1,nz,nz,nn)
      DO l = 1,n
         DO j = 1,n
            Qinv(l,j) = QA1(l,j)
         ENDDO
      ENDDO
   ENDDO
```

Exemplo 1.- A seguir mostra-se as operações de redução da matriz

$$H = \begin{bmatrix} 0.0158 & 0.7338 & -0.7672 & -0.1007 \\ 0.7277 & 0.0679 & -0.1007 & -0.7353 \\ -0.7975 & -0.2107 & -0.0158 & -0.7277 \\ -0.2107 & -0.4533 & -0.7338 & -0.0679 \end{bmatrix} \tag{2.9}$$

à forma superior de Hessenberg, que só precisa de duas transformações, pois a ordem da matriz é quatro.

Transformação 1:

Passo 1):

Para k = 1, i = 2,3,4;

- Achar
  $max(|H(2,1)|, |H(3,1)|, |H(4,1)|) = 0.7975 = H(3,1)$
  define-se $k' = 3$.

Passo 2):

Trocar linhas k' e $k+1$ e colunas k' e $k+1$, ou seja, linha e coluna 3 (onde achamos o maior elemento em valor absoluto) com linha e coluna 2 (correspondente a (k+1));

$$H = \begin{bmatrix} 0.0158 & -0.7672 & 0.7338 & -0.1007 \\ -0.7975 & -0.0158 & -0.2107 & -0.7277 \\ 0.7277 & -0.1007 & 0.0679 & -0.7353 \\ -0.2107 & -0.7338 & -0.4533 & -0.0679 \end{bmatrix} \tag{2.10}$$

Passo 3):
Para k = 1

- DO i = k+2, n
  $n(i, k+1) = H(i, k)/H(k+1, k)$
  - DO j = 1, n
    $H_1(i, j) = H(i, j) - n(i, k+1) * H(k+1, j)$
    $H_1(j, k+1) = H(j, k+1) + n(i, k+1) * H(j, i)$
  - ENDDO
- ENDDO

Como resultado desta primeira transformação, obtém-se a matriz H_1 transformada:

$$H_1 = \begin{bmatrix} 0.0158 & -1.4634 & 0.7338 & -0.1007 \\ -0.7975 & -0.0158 & -0.2107 & -0.7277 \\ 0.0000 & -0.3713 & -0.1244 & -1.3993 \\ 0.0000 & -0.3339 & -0.3976 & 0.1244 \end{bmatrix} \tag{2.11}$$

Transformação 2:
Passo 1):
Para k = 2, i = 3,4;

- Achar
  $max(|H_1(3,2)|, |H_1(4,2)|) = 0.3713 = H_1(3,2)$
  define-se $k' = 3$.

Passo 2):
Trocar linhas k' e $k+1$ e colunas k' e $k+1$, ou seja linha e coluna 3 (onde

achamos o maior elemento em valor absoluto) com linha e coluna 3 (correspondente a (k+1));

$$H_1 = \begin{bmatrix} 0.0158 & -1.4634 & 0.7338 & -0.1007 \\ -0.7975 & -0.0158 & -0.2107 & -0.7277 \\ 0.0000 & -0.3713 & -0.1244 & -1.3993 \\ 0.0000 & -0.3339 & -0.3976 & 0.1244 \end{bmatrix} \tag{2.12}$$

Passo 3):
Para k = 2

- DO i = k+2, n
  $n(i, k+1) = H_1(i, k)/H_1(k+1, k)$
  - DO j = 1, n
    $H_2(i, j) = H_1(i, j) - n(i, k+1) * H_1(k+1, j)$
    $H_2(j, k+1) = H_1(j, k+1) + n(i, k+1) * H_1(j, i)$
  - ENDDO
- ENDDO

Como resultado desta segunda e última transformação, obtém-se a matriz H_2 transformada:

$$H_2 = \begin{bmatrix} 0.0158 & -1.4634 & 0.6432 & -0.1007 \\ -0.7975 & -0.0158 & -0.8651 & -0.7277 \\ 0.0000 & 0.3713 & -1.3827 & -1.3993 \\ 0.0000 & 0.0000 & 0.9577 & 1.3827 \end{bmatrix} \tag{2.13}$$

Finalmente obtém-se uma matriz em *Forma Superior de Hessenberg*:

$$Hes = \begin{bmatrix} 0.0158 & -1.4634 & 0.6432 & -0.1007 \\ -0.7975 & -0.0158 & -0.8651 & -0.7277 \\ 0.0000 & -0.3713 & -1.3827 & -1.3993 \\ 0.0000 & 0.0000 & 0.9577 & 1.3827 \end{bmatrix} \tag{2.14}$$

e a correspondente matriz de transformação:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.9125 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0.2642 & 0.8993 & 1 \end{bmatrix} \tag{2.15}$$

formada a partir da matriz de permutação I e da matriz de transformação $Tinv$, tal que realiza-se:

$$H = QinvHesQ \tag{2.16}$$

Depois de obter uma matriz Hes em FSH e a correspondente matriz de transformação Q, $Qinv$, o passo seguinte é calcular a matriz em *Forma Real de Schur (FRS)*.

A FRS de uma matriz não pode ser determinada, em geral, por uma sequência finita de transformações. É necessário utilizar métodos de fatoração nos quais os elementos da subdiagonal da matriz vão sendo cada vez menores, utilizando uma sequência infinita de transformações de similaridade. Depois de realizar um número de tais transformações, os elementos da subdiagonal chegam a ser tão pequenos que podem ser desprezados e a matriz resultante pode ser aceita como triangular.

Nosso método, depois de obter uma matriz em forma superior de Hessenberg, acha uma matriz $\mathbf{V_s}$ que transforma a matriz de Hessenberg obtida Hes em FRS com $T_{21} = 0$:

$$S = V_s^{-1} Hes V_s = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \tag{2.17}$$

Para realizar tal processo utilizamos algumas subrotinas definidas na biblioteca pública *Scalapack*. De forma geral, a subrotina utilizada devolve os autovalores da matriz Hes, uma matriz S em FRS e uma matriz V_s com os correspondentes autovetores Schur. Tendo obtido as duas expressões (2.16) e (2.17), podemos fazer uma substituição de (2.16) em (2.17) e obter:

$$S = U^{-1} H U \tag{2.18}$$

onde

$$U = QinvV_s \tag{2.19}$$

Posteriormente precisamos reordenar os autovalores na matriz S em FRS que acabamos de obter, pois a subrotina utilizada não garante um ordenamento dos mesmos. Este reordenamento é feito, com outra subrotina da biblioteca *Scalapack*, passando como parâmetro a matriz S em FRS e a matriz U de transformação. Os autovalores são reordenados de forma tal que T_{11} tem os autovalores λ em ordem descendente e T_{22} tem os autovalores $1/\lambda$.

A seguir, calculamos uma nova matriz U:

$$U = QinvL = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} \tag{2.20}$$

da qual obtemos a solução X simétrica, definida não negativa, [29, 3]:

$$X = U_{21}U_{11}^{-1}$$

onde L é a matriz de transformação (V_s) da equação (2.17) reordenada devolvida pela subrotina. Com estes passos obtém-se finalmente:

$$S = U^{-1}HU = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} \tag{2.21}$$

2.3 Exemplo de Aplicação

Nesta seção é mostrado um exemplo de um sistema discreto a ser resolvido. O objetivo do exemplo é mostrar como os resultados obtidos com nosso algoritmo são idênticos aos resultados obtidos através de outros algoritmos já clássicos na literatura. Para atingir este objetivo, apresentamos o exemplo de Laub [3], onde:

$$A = \begin{bmatrix} 0.9512 & 0.0 \\ 0.0 & 0.9048 \end{bmatrix}$$

$$G_1 = \begin{bmatrix} 4.877 & 4.877 \\ -1.1895 & 3.569 \end{bmatrix}$$

$$G_2 = \begin{bmatrix} 0.33 & 0.0 \\ 0.0 & 3.0 \end{bmatrix}$$

$$Q = \begin{bmatrix} 0.005 & 0.0 \\ 0.0 & 0.02 \end{bmatrix}$$

são os dados do problema e devemos resolver a equação algébrica de Riccati:

$$AXA^T - X - A^T X G_1 (G_2 + G_1^T X G_1)^{-1} G_1^T X A + Q = 0$$

A matriz simplética, a matriz em Forma Real de Schur ordenada, e a matriz de transformação utilizada para achar a solução da equação são apresentadas a seguir:

$$H = \begin{bmatrix} 1.05130 & 0.00000 & 83.68837 & -12.23875 \\ 0.00000 & 1.10522 & -12.86638 & 9.38448 \\ 0.00526 & 0.00000 & 1.36964 & -0.06119 \\ 0.00000 & 0.02210 & -0.25733 & 1.09249 \end{bmatrix}$$

$$S = \begin{bmatrix} 1.9696 & 0.2426 & 6.0438 & -85.3252 \\ 0.0000 & 1.4533 & -7.7243 & 5.4132 \\ 0.0000 & 0.0000 & 0.6881 & -0.0848 \\ 0.0000 & 0.0000 & 0.0000 & 0.5077 \end{bmatrix}$$

$$U = \begin{bmatrix} 0.9722 & -0.2339 & -0.0056 & 0.0094 \\ -0.2339 & -0.9718 & -0.0303 & -0.0040 \\ 0.0094 & -0.0056 & 0.2338 & -0.9722 \\ -0.0087 & -0.0497 & 0.9712 & 0.2338 \end{bmatrix}$$

Como solução final, obtemos uma matriz X simétrica, definida não negativa:

$$X = \begin{bmatrix} 0.01045 & 0.00323 \\ 0.00323 & 0.05041 \end{bmatrix}$$

que coincide com a solução dada por Laub no artigo [3].

Capítulo 3

Computação Paralela e Distribuída: Alguns Aspectos

3.1 Introdução

Dentre as principais motivações que estimularam o aparecimento da Computação Paralela e Distribuída, podemos assinalar:

Desempenho - Dado que diversos agentes estarão operando na execução do programa do usuário, então espera-se que o tempo de processamento seja inversamente proporcional ao número de agentes (isto é, processadores, unidades funcionais, etc). Seria ideal se o desempenho da máquina aumentasse pelo menos linearmente com o número de agentes que fossem adicionados ao computador.

Modularidade - Tanto para o fabricante quanto para o usuário, é importante que a arquitetura possa ser expandida incrementalmente através da adição de módulos: aplicações pertencentes à diversas faixas de demanda computacional poderiam ser atendidas a partir de um mesmo produto.

Tolerância a falhas - Muitas aplicações requerem que o sistema de computação assegure operação contínua na presença de falhas de *hardware*.

Outras razões que estimularam o aparecimento da computação paralela foram: O aparecimento de microprocessadores de 32 bits capazes de apresentar desempenho comparável ao de um supercomputador; o desenvolvimento de linguagens para programação paralela, e pesquisas em algoritmos paralelos.

Neste capítulo trataremos de alguns aspectos importantes relacionados com a computação distribuída, isto é, computação em que os diversos processadores não compartilham memória e, consequentemente, toda a comunicação entre processadores deve ser realizada através da troca de mensagens; e de aspectos relacionados com o processamento paralelo referido à replicação de unidades processadoras ou de sub-sistemas destes. A replicação permite que uma mesma instrução seja executada simultaneamente pelos processadores sobre os conjuntos de dados previamente carregados, ou que múltiplas e independentes unidades processadoras sejam utilizadas por partes distintas de um problema.

Iniciamos a seção 3.2 com uma breve classificação dos sistemas distribuídos em sistemas para *broadcast*, sistemas *ponto-a-ponto* e sistemas *assíncronos*. A seguir na seção 3.3 apresentamos os fundamentos da computação paralela segundo os diferentes níveis hierárquicos de um sistema de computação, dentre os quais podemos mencionar o nível do *hardware*, *sistema operacional* e *nível de linguagem de programação*; aprofundamo-nos no nível hierárquico de linguagens de programação que constitui o tipo de paralelismo utilizado ao longo deste trabalho. Na seção 3.4 fazemos uma classificação de arquiteturas paralelas mais utilizadas na prática e assinalamos o tipo de arquitetura utilizada neste trabalho.

3.2 Características dos Sistemas Distribuídos

Um primeiro critério para classificar os sistemas distribuídos é levar em consideração à forma como é realizada a *comunicação* entre os processadores. Segundo este critério, uma divisão bem ampla pode ser feita entre sistemas distribuídos onde a comunicação é feita por *broadcast*, e aqueles em que a comunicação é realizada de forma *ponto-a-ponto*. Quando a comunicação é realizada por *broadcast*, todos os processadores do sistema podem se comunicar diretamente com todos os outros, através do uso de um ou mais canais de comunicação compartilhados por eles. Contrastando com sistemas distribuídos por *broadcast*, existem aqueles em que a comunicação é feita de forma *ponto-a-ponto*. Em particular, esses sistemas com comunicação *ponto-a-ponto* podem ser representados por grafos conexos em que cada nó representa um processador e cada arco entre dois nós representa um canal de comunicação de uso exclusivo dos dois processadores representados por tais nós. No restante deste trabalho, praticamente todo o texto diz respeito apenas a sistemas distribuídos com comunicação *ponto-a-ponto*.

Um outro aspecto que sugere uma classificação de sistemas para computação distribuída é aquele que trata das bases de tempo dos diversos processadores do sistema. Segundo esse critério, os sistemas distribuídos podem ser síncronos ou assíncronos.

Um sistema distribuído síncrono é aquele em que todos os processadores do sistema funcionam segundo um relógio global comum, a que todos têm acesso. Sistemas distribuídos reais são sistemas assíncronos, isto é, sistemas cuja principal característica é a ausência de uma base de tempo comum a todos os processadores que o compõem. Cada processador possui então um relógio local e independente dos demais. Portanto nosso trabalho refere-se a sistemas cuja estratégia de paralelização tenha um esquema assíncrono neste contexto, para calcular a matriz de Hessenberg. No restante, usamos um esquema síncrono.

3.3 Processamento Paralelo

Uma definição de processamento paralelo amplamente aceita é dada a seguir:

"Processamento paralelo (ou concorrente) é uma forma de processamento da informação em que dois ou mais processos, através de um sistema de comunicação, cooperam na solução de um problema".

Nesta seção abordaremos aspectos relacionados com o processamento paralelo no nível hierárquico de linguagens de programação, pois nossa proposta de trabalho está relacionada com a paralelização de um método numérico formulado sequencialmente para obter uma solução simétrica, definida não negativa da equação algébrica de Riccati.

No modelo de processamento sequencial, um programa especifica uma lista de instruções que são executadas sequencialmente. Em processamento paralelo, um programa paralelo especifica duas ou mais listas de instruções que são executadas concorrentemente como processos paralelos.

Paralelismo pode aparecer em formas diferentes, tais como concorrência; vetorização; multiprogramação onde os diferentes processos compartilham o único processador da máquina; multiprocessamento por exemplo, o modelo multiprocessador *UMA (uniform memory access)* onde os processos são executados em paralelo por uma rede de processadores fortemente acoplados, ou seja o código e os dados de todos os processos ficam armazenados numa memória comum; computação distribuída onde os processos ficam associados a processadores fracamente conectados por uma rede de comunicação, ou seja cada nó de processamento possui sua memória local a partir da qual são executados os processos, entre outras formas.

Um sistema multicomputador com memória distribuída, por exemplo, consiste de múltiplos computadores (nós), interconectados por uma rede de troca de mensagens. Cada nó consiste particularmente de um processador e memória local, etc. A rede de troca de mensagens fornece uma conexão

estática ponto-a-ponto entre os nós, todas as memórias locais são privadas a cada nó e só podem ser acessadas por processadores locais.

Existem basicamente dois caminhos para explorar o paralelismo das arquiteturas no nível hierárquico de linguagem de programação de alto nível: *adaptação* e *(re)programação paralela.* Nosso caminho principal é *adaptação*, pois fazemos uma adaptação do programa sequencial de forma que ele possa tirar proveito das características do paralelismo da máquina onde está sendo executado. Uma vantagem desse esquema é que ele torna compatível o processamento sequencial com o paralelo.

Através do compilador de linguagem de programação Fortran77 e do uso da biblioteca de rotinas paralelas *Scalapack*, nosso programa sequencial, explicado no capítulo 2, é transformado num programa paralelo. Nele os dados do programa são decompostos em grupos que serão distribuídos usando os canais de comunicação que existem entre os diferentes processadores que executarão simultaneamente, um mesmo programa.

Comunicação e Sincronização

A existência de diversas tarefas simultâneas ou paralelas que estão envolvidas numa aplicação global, exige mecanismos de comunicação e de sincronização de modo a permitir uma cooperação efetiva entre as tarefas.

Para coordenar processos paralelos, o sistema operacional precisa prover mecanismos de comunicação entre os processos. Existem dois esquemas básicos de comunicação: memória compartilhada e troca de mensagens.

No primeiro esquema de comunicação as tarefas trocam informação através de variáveis compartilhadas, que são variáveis que podem ser usadas por mais de uma tarefa.

O esquema de comunicação *troca de mensagens* pode ser empregado em sistemas multiprocessadores com ou sem memória compartilhada. As mensagens permitindo a comunicação entre os processos são transmitidas através de segmentos de memória compartilhados pelos processadores ou através de

barramentos de comunicação. Uma possível implementação desse mecanismo de comunicação é aquela provida por duas funções do sistema operacional UNIX:

- send(destino, mensagem).
- receive(fonte, mensagem)

Compete à primeira primitiva enviar mensagens para um processo destino enquanto que a outra primitiva é responsável pela recepção de mensagens produzidas pelo processo fonte. Estas primitivas podem ser implementadas com ou sem bloqueio. No primeiro caso, o processo invocando uma das primitivas permanece bloqueado até que o outro processo invoque a primitiva simétrica. Esse tipo de comunicação também é chamado de comunicação *síncrona*. No segundo caso, também conhecido como comunicação *assíncrona*, os processos prosseguem após a execução de uma das primitivas.

Os mecanismos de *sincronização* são necessários quando tarefas comunicam-se entre si. Uma tarefa ao desejar estabelecer uma comunicação deve realizar uma ação que a outra tarefa possa detectar. Esta ação pode corresponder a atribuição de um valor a uma variável compartilhada, ou ao envio de uma mensagem.

3.4 Classificação de Arquiteturas Paralelas

Os dois tipos de arquiteturas paralelas mais utilizados são os computadores *SIMD* (única instrução, múltiplos dados) e os *MIMD* (múltiplas instruções, múltiplos dados). Num computador *SIMD* todos os processadores executam a mesma instrução simultaneamente sobre diferentes conjuntos de dados. Nos computadores *MIMD* os processadores executam diferentes instruções com diferentes conjuntos de dados de forma independente.

3.4.1 Sistemas de Memória Compartilhada

O fator mais utilizado para classificar os computadores *MIMD* é o método utilizado para compartilhar a informação entre os processadores. Os multiprocessadores com memória compartilhada oferecem ao usuário uma memória global na qual todos os processadores têm acesso. Os processadores comunicam-se através de variáveis compartilhadas em memória e com capacidade de acessar qualquer posição na memória. A memória por sua vez está estruturada em módulos. Nestes sistemas podem ocorrer conflitos de memória quando vários processadores pretendem acessar um mesmo módulo, o que pode provocar uma perda significativa do rendimento. Estes conflitos podem ser evitados por uma serie de técnicas, e podem ser eliminados incorporando-se caches e memórias locais.

Nos sistemas de memória compartilhada também podemos diferenciar entre multiprocessamento simétrico, onde todos os processadores têm igual capacidade para executar programas, sejam de usuário, do sistema operacional ou subrotinas de entrada/saída, e os de processamento não simétrico, onde um único processador, ou um subconjunto deles, pode executar as rotinas do sistema operacional ou dirigir os processos de entrada/saída. Atualmente, os sistemas simétricos de memória compartilhada *Symmetric Multiprocessing (SMP)* são mais utilizados nas aplicações comerciais, pois proporcionam um grande rendimento na execução de aplicações já existentes para sistemas monoprocessador, pois o efeito da rede é transparente nas atuais aplicações.

Nos computadores *MIMD* com memória distribuída cada processador tem seu próprio programa e memória de dados aos quais os demais não podem acessar diretamente. A comunicação dos dados compartilhados é através de troca de mensagens entre os processadores com uso de uma rede de interconexão (paradigma de troca de mensagens).

O tipo de computador que vamos considerar no transcurso deste trabalho para calcular a matriz em forma real de Schur, é um computador *MIMD* com memória distribuída e topologia de malha, onde os processadores são organizados como numa matriz (2-dimensões) e cada processador é conecta-

do diretamente aos seus processadores vizinhos do norte, sul, leste e oeste, Figura 3.1. Cada processador tem sua própria memória local e processa seus próprios dados. A topologia de malha é utilizada pois sua pesquisa tem sido estimulada em vista da correspondência entre malhas e algoritmos orientados a matrizes.

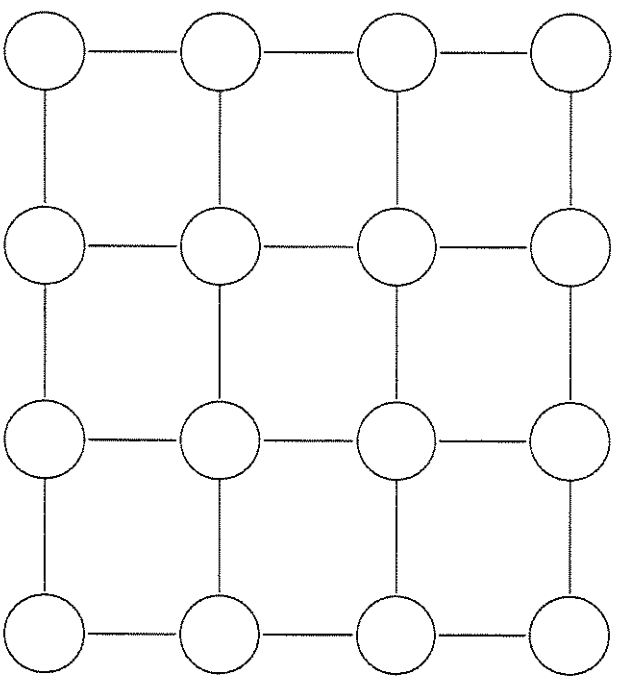


Figura 3.1: Topologia de Malha

Uma qualidade desejável nos algoritmos paralelos é a escalabilidade, ou seja, que o desempenho do algoritmo não dependa do tamanho do sistema nem do tamanho do problema. Por outro lado, deve-se conseguir uma repartição equilibrada da carga computacional entre os processadores. Também é desejável minimizar o volume de comunicação interprocessador, pois pressupõe-se um consumo de tempo importante, que afetará a eficiência dos algoritmo paralelos. Concretamente, deve-se minimizar a redistribuição de dados entre as etapas do algoritmo, pois isto implica um número importante

de comunicações.

A atribuição de dados e computação de um problema nos diferentes processadores constitui um problema matemático conhecido como mapeamento. Em nosso caso o melhor mapeamento é o que minimiza o tempo de execução. Em geral, o tempo de execução de um algoritmo sobre um multicomputador pode-ser decomposto em dois componentes [6]: tempo de computação t_c e o tempo de comunicação t_{com}.

3.5 Características Gerais

São fatores importantes no projeto de um algoritmo com memória distribuída [16]:

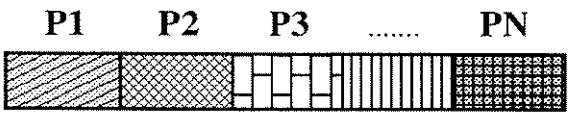
- número de processadores e capacidade das memórias locais;
- como são interconectados os processadores;
- velocidade da computação relativa à velocidade de comunicação entre processadores.

Num multiprocessador cada processador tem um número de identificação *id*; para a topologia de malha nós utilizamos *Proc(i)* para designar o i-ésimo processador .

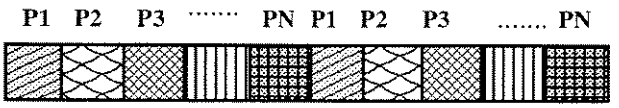
Distribuição de Dados

Os elementos dos vetores podem ser alocados na memória do sistema usando dois tipos de distribuição: por blocos ou cíclica, Figura 3.2. Entende-se por distribuição por blocos aquela onde os elementos do vetor de tamanho N são divididos em P blocos de tamanho $K = N/P$, onde P é o número de processadores. No caso da distribuição cíclica, os elementos são divididos

em partes de tamanho L, (em nosso caso L pode ser igual a um), que são distribuídos ciclicamente sobre os processadores.



Distribuição por blocos



Distribuição cíclica

Figura 3.2: Distribuição de Dados

Nossa distribuição de dados para realizar o cálculo em paralelo da primeira parte de nosso algoritmo é definida por blocos, mas na segunda parte, como estamos utilizando subrotinas paralelas definidas numa biblioteca pública, a distribuição de dados é realizada de forma cíclica.

Estrutura de Dados

Consideremos o cálculo em paralelo, [6], do produto matriz-vetor Ax numa rede de p *processadores*, onde $A \in R^{n \times n}$ e $x \in R^n$. Suponhamos que n é divisível por p e denotemos $k = n/p$. Existem dois métodos básicos para distribuir os elementos da matriz A:

Distribuição por linhas: cada processador i armazena as k linhas da matriz A numeradas $(i-1)k+1$ até ik, assim como o vetor x. O processador i então calcula as k coordenadas do produto Ax;

Distribuição por colunas: cada processador i armazena as k colunas da matriz A numeradas $(i-1)k+1$ até ik, assim como as k coordenadas

do vetor x numeradas $(i-1)k+1$ até ik. O processador i então calcula os correspondentes k termos da soma que define cada coordenada do produto Ax .

Modelo Troca de mensagens

O paradigma de programação *Troca de Mensagens* foi utilizado e algumas das razões, [24], para tanto são :

1. grande número de plataformas que o suportam;
2. programas escritos com estilo troca de mensagens podem rodar sobre:
   - multiprocessadores com memória compartilhada ou distribuída;
   - redes de estações de trabalho;
   - sistemas mono-processador.

Neste modelo a coordenação entre os processadores é realizada pelo envio e recebimento de mensagens. Um processador pode enviar uma mensagem em qualquer momento e não tem que aguardar pelo relógio global do sistema. Esta é a principal diferença com o modelo síncrono.

O modelo troca de mensagens não se importa em como os processadores estão interconectados; qualquer processador pode enviar mensagens para qualquer outro processador, não obstante, na prática é bom ter um arranjo tal que as mensagens sejam enviadas aos processadores mais próximos na rede.

3.6 Modelo de programação

Para definir as características de um modelo de programação determinado é necessário analisar vários fatores. A seguir são listados alguns deles.

- Qual será o suporte para as comunicações entre os componentes?

  - Será permitida comunicação ponto-a-ponto?
  - Será permitida comunicação em grupo?

- Quais serão as características das comunicações? Bloqueante, não bloqueante ou bloqueante com tempo de espera finito?

- Qual será o suporte para troca de mensagens em redes heterogêneas? Existem basicamente 4 formas.

  - Enviar ao receptor uma cópia exata do conteúdo da mensagem. Esta forma é pouco apropriada para redes heterogêneas, pois o receptor tem de converter a representação dos dados no transmissor em sua própria representação. Isto obriga o receptor saber o formato de representação dos dados em cada um dos possíveis transmissores.
  - Converter os dados à representação do receptor. Por razões análogas à forma anterior, esta também é pouco apropriada para redes heterogêneas.
  - Converter os dados a uma representação comum (Ex. XDR, eXternal Data Representation, da Sun) no transmissor e converter da representação comum à nativa no receptor. Desta forma o transmissor e o receptor somente precisam saber como converter a sua representação à representação comum e vice-versa.
  - Uma variação da anterior consiste em o programador descrever a sua mensagem (No. de componentes, tipo de dados de cada componente) e o sistema se encarregar de fazer as conversões necessárias. Esta variante permite otimizações, pois o sistema pode decidir não fazer conversões se o transmissor e o receptor têm a mesma representação de dados.

- Qual será o suporte para a manipulação de tarefas e da plataforma?

  - O sistema terá suporte para criar e destruir tarefas?

– O sistema terá suporte para manipular os computadores onde são executadas as tarefas? Isto inclui a possibilidade de se tirar proveito de novos computadores disponíveis durante a execução ou de se adaptar à perda de computadores que estavam trabalhando.

3.7 Algumas Plataformas para a Implementação de Aplicações Paralelas

Nesta seção serão apresentadas algumas plataformas para a implementação de aplicações paralelas, bem como as abordagens seguidas nas mesmas para cada um dos pontos acima.

PVM (Parallel Virtual Machine)

O PVM baseia-se no conceito de máquina paralela virtual [14]. A máquina paralela virtual está formada pelos computadores participantes no processo paralelo. O programador pode adicionar ou eliminar dinamicamente computadores da máquina virtual. O sistema inclui também funções para a manipulação de tarefas, que permitem que qualquer tarefa da aplicação possa criar e destruir tarefas dinamicamente.

O PVM tem suporte para comunicações ponto-a-ponto. As operações de envio de mensagens retornam assim que a mensagem é colocada na rede. As operações de recepção podem ser bloqueantes, não bloqueantes ou bloqueantes com tempo finito. As mensagens enviadas são sempre colocadas em buffers, que são explicitamente manipulados pelo programador. É tarefa do programador empacotar e desempacotar explicitamente as mensagens. Neste processo ocorre a conversão de representação de dados. Implicitamente os dados são convertidos à representação XDR, embora o programador possa especificar outros tipos de conversão.

Além de comunicações ponto-a-ponto, o PVM tem suporte para comunicações em grupo. Os grupos têm carácter dinâmico, isto é, um processo pode

entrar e sair de um grupo em qualquer momento. O PVM oferece operações de broadcast, multicast e de redução globais (por exemplo, mínimo, máximo, soma, etc).

MPI (Message Passing Interface)

Uma implementação MPI (Message Passing Interface) é um conjunto de especificações para comunicação entre processadores, com objetivos similares aos do PVM [28]. O MPI não tem suporte para a criação e destruição de tarefas. O número de tarefas é fixo desde o início do processamento, assim como o número de máquinas participantes. O sistema também não inclui suporte para o conceito de máquina paralela virtual.

O MPI oferece um conjunto extremamente flexível de funções para a troca de mensagens ponto-a-ponto. Tanto as funções para enviar mensagens quanto as de receber podem ser bloqueantes ou não bloqueantes. A semântica das funções de envio de mensagens bloqueantes é escolhida pelo programador entre 4 possibilidades: *standard* (o sistema decide se armazena a mensagem em um buffer ou se a envia diretamente), *buffered* (o sistema sempre armazena a mensagem em um buffer), *synchronous* (a função não retorna até a mensagem ser recebida) e *ready* (a função retorna com sucesso somente se alguma outra tarefa já anunciou que vai receber a mensagem).

Além das funções para a comunicação ponto-a-ponto, o MPI oferece também um grande conjunto de funções para as comunicações em grupo: *broadcast, multicast, scatter* (um processo envia uma informação diferente a cada um dos outros processos do grupo), *gather* (um processo recebe uma informação diferente de cada um dos outros processos do grupo), *all to all scatter* e *all to all gather* (todos os processos enviam/recebem informações diferentes dos outros processos do grupo) e operações de redução globais (mínimo, máximo, soma, produto, OR, XOR, mínimo e máximo locais, etc). As funções para comunicação em grupo são sempre bloqueantes, para simplificar a implementação. O programador pode escolher entre empacotar explicitamente a mensagem ou descrever a sua estrutura. O sistema escolhe a melhor representação para

enviar a mensagem através da rede.

O MPI suporta diferentes topologias de comunicação entre os processos. O programador pode, por exemplo, organizar seus processos em forma de árvore. A partir desse ponto, cada processo pode se comunicar com seu pai e seus filhos na árvore sem especificar explicitamente o identificador de tarefa. Os processos podem formar grupos. Os grupos de processos têm caráter estático, ou seja, para formar e destruir um grupo os processos têm que se sincronizar.

3.8 Sumário

Podemos assinalar que os *Sistemas Distribuídos* são redes de processadores fracamente conectados por enlaces de comunicação. Possuem um número variado de conjuntos processador mais memória local, com a diferença de que não existe uma memória comum. Toda a comunicação entre tarefas em processadores diferentes deve ser realizada através da troca de mensagens. Nos sistemas distribuídos, as tarefas são alocadas estaticamente a cada conjunto *(processador, memória)*. Cada processador utiliza sua memória local para o armazenamento de dados e intercambia informação com outros processadores em grupos de bits denominados *pacotes* utilizando para isto os enlaces de comunicação da rede.

A implementação dos mecanismos de *comunicação* e *sincronização* através de troca de mensagens dá-se pelo envio *(send)* e recebimento *(receive)* de mensagens ao invés de uma leitura ou escrita de variáveis compartilhadas. A *comunicação* ocorre porque uma tarefa ao receber uma mensagens obtém dados enviados por outra tarefa, enquanto que a *sincronização* é obtida porque uma mensagen somente pode ser recebida após a mesma ter sido enviada o que restringe a ordem na qual estes dois eventos devem ocorrer.

Neste trabalho utilizaremos como mecanismo de comunicação o esquema síncrono com transferência de dados entre dois processos, ou seja comunicação ponto-a-ponto bloqueante.

Capítulo 4

Solução Paralela do algoritmo de Schur-Modificado

4.1 Introdução

Neste trabalho a obtenção da solução paralela da equação algébrica de Riccati foi dividida em duas partes: A primeira corresponde a formação da matriz em forma de Hessenberg via *TSEE*, utilizando como plataforma de programação paralela o *MPI*, com memória distribuída, comunicação síncrona, e a semântica das funções para enviar e receber mensagens é bloqueante com modo de programação $SPMD$; a segunda parte corresponde à obtenção da matriz em Forma Real de Schur que foi implementada utilizando as subrotinas da biblioteca pública *Scalapack* [22].

A *Scalapack* é uma biblioteca de alto desempenho com rotinas de Álgebra Linear para computadores *MIMD* com paradigma de troca de mensagens e memória distribuída em redes de estações de trabalho suportando *PVM* e/ou *MPI*. Utiliza também como parte de suas rotinas a biblioteca sequencial *BLAS*, a paralela *PBLAS*, os subprogramas de comunicação da algebra linear

básica *BLACS* e as rotinas da *LAPACK*.

Similarmente ao *LAPACK*, as rotinas da *Scalapack* são baseadas em algoritmos com particionamento em blocos, o que minimiza a frequência de movimento de dados entre os diferentes níveis da memória hierárquica. Os algoritmos são apresentados como processos e a comunicação pode ser ponto-a-ponto ou em grupo quando necessário. De forma geral, a *Scalapack* foi desenvolvida e testada para o caso de um processo por processador.

Dos vários algoritmos paralelos que têm sido propostos, um dos que receberam maior atenção recentemente é o baseado em multiplicação de matrizes, pois a multiplicação de matrizes de alta ordem é forte candidata à paralelização [17]. Nos interessa a implementação paralela do método de Schur para o caso de matrizes reais não simétricas utilizando *TSEE*: método de Schur-Modificado paralelo.

Na subseção 4.1.1 consideramos a paralelização do algoritmo de Schur-Modificado para matrizes densas, não simétricas sobre um sistema de memória distribuída; o modo de programação paralela utilizado é *SPMD*, ou seja todos os processadores vão executar o mesmo programa, porém podem executar tarefas diferentes sobre dados diferentes. A estratégia de paralelização aplicada segue um esquema assíncrono; descreve-se as principais subrotinas utilizadas na solução do problema apresentado. Em particular é descrito como o programa fonte sequencial, desenvolvido no capítulo 2, é alterado, sendo incluídas as primitivas do paralelismo apresentadas na biblioteca *Scalapack* e na plataforma *MPI*, as quais implementam as primitivas de criação, comunicação e sincronização de processos paralelos; desta maneira realizamos o procedimento de formação paralela das matrizes de Hessenberg e Forma Real de Schur; na subseção 4.1.2 apresentamos os resultados experimentais da implementação do algoritmo paralelo e ainda expomos o algoritmo de programação paralela desenvolvido.

4.1.1 Paralelização do Algoritmo

A implementação paralela da primeira parte de nosso algoritmo, ou seja, a correspondente a formar a matriz de Hessenberg via *TSEE* é baseada fundamentalmente em operações do tipo produto matriz-matriz, as quais representam maior custo computacional; essas operações são efetuadas acessando os elementos da matriz por linhas; desta maneira formamos blocos que são distribuídos entre os diferentes processadores participantes do processamento. Neste caso é possível paralelizar o produto tal que cada processador calcula os valores dos elementos diferentes do vetor resultante. O método para acessar aos elementos da matriz é a *Distribuição por linhas.* Ou seja, os P processadores que participam do processamento paralelo podem ser apresentados como um arranjo linear uni-dimensional, onde os elementos são distribuídos em blocos. Por exemplo, vamos distribuir uma matriz (6×6) em 4 processadores; os processadores são numerados de 0 até $P-1$ e as linhas da matriz de 1 até N $(N = 6)$. A Tabela 4.1 apresenta uma distribuição de blocos por linha, onde cada linha ou submatriz formada é nomeada por um número de processador.

0
1
2
3

Tabela 4.1: Armazenagens por linha

Neste caso temos que cada processador armazena no máximo (NB) número de linhas, onde $\lceil NB = N/P \rceil$ e a linha k é armazenada no processador $\lfloor k/NB \rfloor$. A distribuição atribui blocos de linhas de tamanho (NB) nos processadores sucessivos. Se o valor de P é divisível pelo valor de N, então cada processador vai receber blocos do mesmo tamanho, de forma a garantir um melhor balanceamento de carga.

Com o modo de programação *SPMD* cada processador participante realiza uma operação de multiplicação ($C = A * B$) com os dados enviados pelo processador principal, ou seja os dados são divididos e cada processador calcula no máximo (NB) blocos da matriz resultante na multiplicação matriz-matriz que realiza. O processador principal, encarregado de distribuir os dados e realizar seus cálculos envia no máximo (NB) blocos da matriz A (NB, N) e a matriz B (N, N) para cada processador participante do processo e recebe uma matriz de dimensão (NB,N) com o produto final de cada processador. A Figura 4.1, ilustra o procedimento descrito para um sistema de ordem (6×6) onde os processadores recebem as linhas seguintes:

- $\mathrm{P}_0 = linha\mathrm{A}_{1n}$ da matriz (processador principal)
- $\mathrm{P}_1 = linhas\mathrm{A}_{2n}$ e A_{3n} da matriz
- $\mathrm{P}_2 = linhas\mathrm{A}_{4n}$ e A_{5n} da matriz
- $\mathrm{P}_3 = linha\mathrm{A}_{6n}$ da matriz

No Apêndice A é apresentada a subrotina paralela implementada para realizar a multiplicação de duas matrizes.

A implementação paralela da segunda parte do nosso algoritmo, a correspondente a achar a *Forma Real de Schur* é baseada na utilização das subrotinas definidas na biblioteca pública *Scalapack*. Esta biblioteca atualmente está escrita em linguagem Fortran77 utilizando o paradigma de programação de *troca de mensagens*, Figura 4.2, onde vários processadores, cada um com sua memória local, executam tarefas em paralelo.

Na implementação estão presentes os mecanismos básicos da biblioteca *Scalapack* utilizada, tais como criação da topologia malha, comunicação entre tarefas e entrada-saída de dados. A *Scalapack* utiliza uma distribuição de dados com blocos cíclicos de dimensão-2 nos quais a matriz, dividida em blocos de linhas e colunas ($MB \times NB$), é atribuída ao mesmo processador. A distribuição de dados cíclica é parametrizada por quatro valores: P, Q, r e c onde ($P \times Q$) é a definição da topologia malha utilizada e ($r \times c$) é

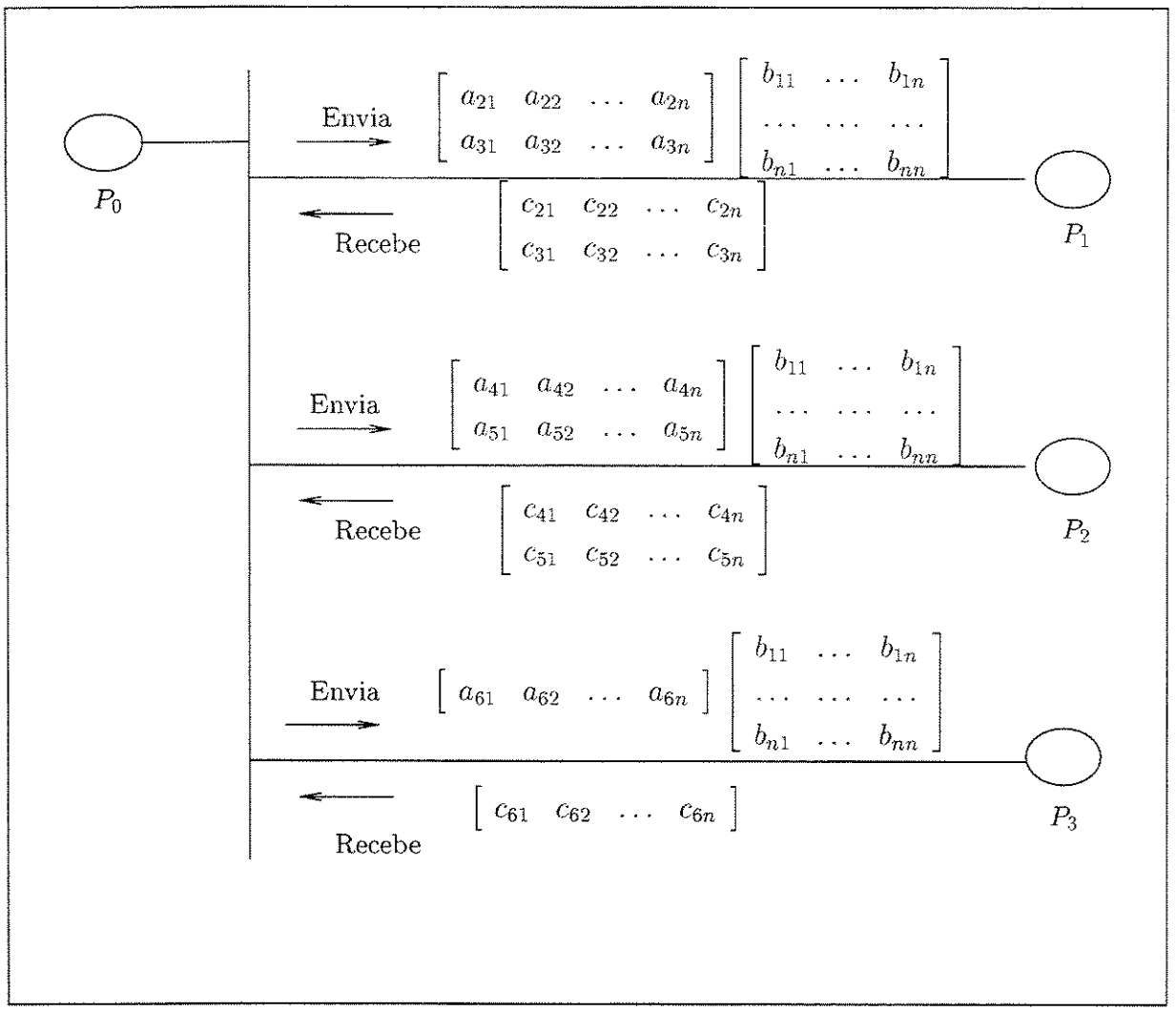


Figura 4.1: Processo Paralelo de Distribuição de matrizes

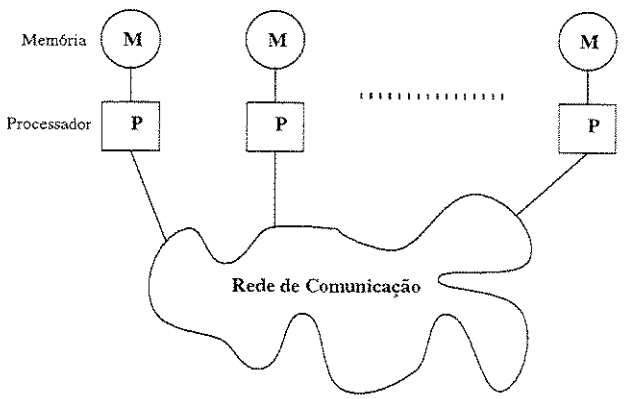


Figura 4.2: Paradigma de programação de Troca de Mensagens

a dimensão do bloco. Supõe-se que os processadores estão arranjados logicamente como uma malha de P linhas e Q colunas; assim o número de processadores utilizados corresponde ao produto $R = P \times Q$.

Após o particionamento da matriz $(N \times N)$ em blocos $(MB \times NB)$, o primeiro bloco é associado ao processador com coordenadas $(RSRC, CSRC)$, onde $RSRC$ e $CSRC$ correspondem respectivamente aos números dos processos linha e coluna aos quais o primeiro bloco da matriz é atribuído. Por exemplo, a entrada (I, J): cada elemento da matriz global, está localizada no processador definido pelas coordenadas (p_r, p_c) do bloco local (l, m) na posição (x, y) dada por:

$$(l, m) = (\lfloor (I-1)/(P * MB) \rfloor, \lfloor (J-1)/(Q * NB) \rfloor),$$

$$(p_r, p_c) = ((RSRC + \lfloor (I-1)/MB \rfloor) mod P, (CSRC + \lfloor (J-1)/NB \rfloor) mod Q),$$

$$(x, y) = (mod(I-1, MB) + 1, mod(J-1, NB) + 1).$$

A seguir apresentamos na Figura 4.3 o mapeamento de uma matriz A (6×6) particionada em blocos (2×2) e na Figura 4.4 o ponto de vista da distribuição da matriz (6×6) numa topologia malha (2×2), onde podemos observar que a entrada local $A(2, 3)$ definida no processador de coordenadas $(0, 0)$ contém o valor da entrada da matriz global a_{25}.

$$\left[\begin{array}{cc|cc|cc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ \hline a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{array}\right]$$

Figura 4.3: Matriz 6×6 particionada em blocos 2×2

$$\left[\begin{array}{c|cccc|cc} & \multicolumn{4}{c|}{0} & \multicolumn{2}{c}{1} \\ \hline & a_{11} & a_{12} & a_{15} & a_{16} & a_{13} & a_{14} \\ 0 & a_{21} & a_{22} & a_{25} & a_{26} & a_{23} & a_{24} \\ & a_{51} & a_{52} & a_{55} & a_{56} & a_{53} & a_{54} \\ & a_{61} & a_{62} & a_{65} & a_{66} & a_{63} & a_{64} \\ \hline 1 & a_{31} & a_{32} & a_{35} & a_{36} & a_{33} & a_{34} \\ & a_{41} & a_{42} & a_{45} & a_{46} & a_{43} & a_{44} \end{array}\right]$$

Figura 4.4: Matriz 6×6 numa Topologia Malha 2×2

Existe uma subrotina da *Scalapack* denominada *BLACS-GRIDINIT* que realiza o mapeamento dos elementos da matriz global em cada processador por linha, ou seja vai distribuindo os elementos ordenados por linha, veja a definição na Tabela 4.2, onde quatro processos são distribuídos numa malha (2×2) com uma distribuição por linha:

Para um exemplo de ordem (20×20), os parâmetros da subrotina que inicializam a topologia malha vão estar definidos como: MB=NB=6, P=Q=2 e LLD=12 que corresponde ao número máximo de linhas/colunas em cada bloco da submatriz; para o exemplo que estamos apresentando os parâmetros

0	1
2	3

Tabela 4.2: Quatro processos mapeados numa malha (2×2)

estão definidos como: MB=NB=6, P=Q=2 e LLD=6.

4.1.2 Resultados Experimentais

Os resultados que apresentamos foram obtidos com diferentes valores dos parâmetros. Por exemplo, para uma matriz de ordem (6×6) que definimos a seguir os parâmetros são os seguintes: N=6, P=3, NB=2. Com este tamanho de matriz, obtemos tempos de execução paralela de cada uma das partes do programa maiores do que os tempos respectivos obtidos na implementação sequencial, veja Tabela 4.3; mas isso é diferente quando utilizamos P=2.

	SEQUENCIAL	PARALELO	
Atividade	1-proc	2-proc	3-proc
Leitura dos dados	0.1256	0.0296	0.0323
Matriz Hessenberg	0.1699	0.1060	0.2049
Forma Real de Schur	0.2228	0.1412	0.3623
Cálculo de X	0.0007	0.0077	0.0131
Tempo Total de Execução	0.5190	0.2845	0.6126

Tabela 4.3: Tempos de Execução das partes do Algoritmo numa matriz (6×6)

Primeiramente fazemos a compilação do programa escrito em linguagem *Fortran77* e os respectivos enlaces com as subrotinas paralelas utilizadas; para isto criamos um arquivo, que denominamos *Make* que inclui os endereços onde estão as diferentes bibliotecas de subrotinas utilizadas no programa. A seguir executamos o programa paralelo obtido da compilação *laub-disc*

com suporte *MPI* numa rede de estações de trabalho utilizando neste caso particular apenas 3 processadores.

```
annabell@urano[58] make
annabell@urano[59] /softs/mpi/bin/mpirun -np 4 laub-disc
```

Os resultados obtidos são armazenados num arquivo de saída. Neste caso particular vamos apresentar os resultados do algoritmo para resolver o sistema discreto de ordem (6×6) descrito acima.

```
 Matriz Simplética
    1.2897     0.0843     0.0679     0.0983    -0.0768     0.0048
   -0.0866     1.2912    -0.0772    -0.0881     0.0699    -0.0055
    0.0564     0.0354     1.2686     0.0072    -0.0067     0.0015
    1.2897     0.0843     0.0679     0.8723    -0.0270    -0.0310
   -0.0866     1.2912    -0.0772    -0.1374     0.8399    -0.0248
    0.0564     0.0354     1.2686    -0.0372     0.0375     0.7905
Matriz de Hessenberg: (Hes)
    1.2897     0.1009    -0.0375     0.0462     0.0320     0.0679
    1.2897     0.8700     0.0139     0.0268    -0.0038     0.0679
    0.0000    -0.2265     1.5603     1.3258    -0.0561    -0.0727
    0.0000     0.0000    -0.0692     0.6067    -0.0059    -0.0327
    0.0000     0.0000     0.0000    -0.5674     1.3009     1.2802
    0.0000     0.0000     0.0000     0.0000    -0.0205     0.7246
Matriz de Schur Ordenada:
    1.5839     0.1986     0.0103    -0.9185     0.6041     0.4631
    0.0000     1.3350     0.0441    -0.6486     0.2116    -1.0269
    0.0000     0.0000     1.2581    -0.2611    -1.5437    -0.0462
    0.0000     0.0000     0.0000     0.6314     0.0912     0.0428
    0.0000     0.0000     0.0000     0.0000     0.7948    -0.0100
    0.0000     0.0000     0.0000     0.0000     0.0000     0.7490
Matriz de Transformação QA final:QA=Qinv*L1
    0.2637     0.2908     0.1644    -0.8180    -0.2661    -0.2810
```

```
-0.4319    0.0931    0.1601    0.3801   -0.7654   -0.5969
 0.0278    0.2641   -0.2679   -0.1533    1.0840   -0.4373
 0.4632    0.6073    0.4901    0.3894    0.1043    0.1178
-0.8681    0.2984    0.3841   -0.0446   -0.0857   -0.0702
-0.0188    0.6373   -0.7627    0.2915   -0.3541   -0.3174
Solução --- E. A. de Ricatti ---------

 2.2504    0.2837   -0.2788
 0.2837    2.1863    0.0472
-0.2788    0.0472    2.7041
```

Com nossa implementação conseguimos provar que estamos obtendo os resultados propostos para este trabalho: obter uma solução simétrica, definida não negativa da equação algébrica de Riccati com *TSEE*, embora não tenhamos bons tempos de execução para todos os casos. A seguir apresentamos outros exemplos teste com diferentes dimensões nas matrizes.

Quando testamos com matrizes de ordens maiores, por exemplo (50×50) obtemos, Tabela 4.4, que os tempos de execução paralela de cada uma das partes do programa são bem menores do que os tempos respectivos obtidos na implementação sequencial; assim podemos concluir que a paralelização surte melhores resultados quando o problema envolve cálculos com matrizes de ordens bem maiores.

	SEQUENCIAL	PARALELO	
Atividade	1-proc	2-proc	3-proc
Leitura dos dados	0.3092	0.2642	0.2869
Matriz Hessenberg	59.0045	8.3596	10.3933
Forma Real de Schur	1.8914	0.9707	1.3932
Tempo Total de Execução	61.2051	9.5945	12.0734

Tabela 4.4: Tempos de Execução das partes do Algoritmo numa matriz (50×50)

Contrariamente, temos um exemplo teste com uma matriz de ordem

(100×100),na qual os resultados não são favoráveis, ou seja obtemos tempos de execução paralela de cada uma das partes do programa bem maiores do que os tempos respectivos obtidos na implementação sequencial, veja Tabela 4.5; achamos que a causa deste problema esteja na definição do tamanho dos pacotes no sistema operacional ou na grande comunicação entre os processadores para obter o resultado final do problema.

	SEQUENCIAL	PARALELO	
Atividade	1-proc	2-proc	3-proc
Leitura dos dados	0.2954	0.9317	1.0464
Matriz Hessenberg	57.8246	85.4671	106.0137
Forma Real de Schur	1.7721	1.8458	3.0281
Tempo Total de Execução	59.8921	88.2446	110.0882

Tabela 4.5: Tempos de Execução das partes do Algoritmo numa matriz (100×100)

A seguir apresentamos parte do algoritmo implementado em forma paralela para achar a matriz em *Forma Superior de Hessenberg* e a matriz em *Forma Real de Schur* com as quais obtemos a solução da equação algébrica de Riccati.

```
INICIALIZAR MPI:
  MPI_INIT
  MPI_COMM_RANK
  MPI_COMM_SIZE
Se Processador Principal fazer
   INICIALIZAR MATRIZES
   LER DADOS INICIAIS
Finalizar
CALCULAR MATRIZ DE HESSENBERG
INICIALIZAR A TOPOLOGIA DE MALHA:
    DEFINIR P,Q
```

```
DEFINIR O DESCRITOR DAS MATRIZES
    DEFINIR DIMENSÃO DO BLOCO
    ORDEM DAS MATRIZES
    IDENTIFICADOR ONDE O PRIMEIRO BLOCO É ATRIBUIDO
CALCULAR A MATRIZ EM FORMA REAL DE SCHUR
Se Processador Principal fazer
   ACHAR SOLUÇÃO DA EQUAÇÃO
   APRESENTAR OS RESULTADOS
Finalizar
```

Capítulo 5

Conclusões e Comentário

Neste trabalho apresentamos uma proposta de metodologia para solução da *Equação Algébrica de Riccati* em formas sequencial e paralela. A implementação sequencial assim como a proposta de implementação paralela foi baseada fundamentalmente nas rotinas definidas em bibliotecas públicas de álgebra linear: *LAPACK*, *SCALAPACK*, *PBLAS* e *BLACS*.

O algoritmo de Schur-Modificado proposto, além de numericamente estável, é mais eficiente quanto ao número de operações de multiplicação realizadas que o algoritmo proposto por Laub para resolver a *Equação Algébrica de Riccati*.

Com base nos exemplos teste executados com nosso algoritmo podemos dizer que nosso algoritmo esta restricto a utilização de matrizes com ordens de magnitude iguais nos elementos, de forma a garantir que os autovalores originais da matriz não tenham variação após ter aplicado algum procedimento numérico; nos exemplos que rodamos com matrizes muito mal balanceadas, obtivemos resultados não desejados, como: solução não simétrica e definida não negativa e autovalores da matriz simplética e da matriz em forma superior de Hessenberg diferentes. Embora, possamos incluir em nosso algoritmo o processo de balanceamento para poder trabalhar com qualquer tipo de matrizes este caso não foi analisado neste trabalho e o propomos como trabalho futuro; só testamos o algoritmo para matrizes não simétricas com ordens de

magnitude iguais nos elementos e muito bem condicionadas; com as quais os resultados foram ótimos.

Como contra-exemplo do condicionamento, testamos uma matriz simplética não simétrica, bem balanceada de ordem (6×6) muito mal condicionada e obtivemos resultados corretos, ou seja mantivemos inalterados os autovalores originais da matriz.

Em muitos algoritmos e Sistemas Paralelos/Distribuídos o tempo consumido pela comunicação entre processadores é uma fração do tempo total necessário para resolver o problema.

Em relação a comparação em tempo de execução dos algoritmos implementados em formas sequencial e paralela, podemos dizer que o processamento paralelo nem sempre foi melhor que o processamento sequencial quanto ao tempo de execução, pois no processamento paralelo existe uma alta comunicação de dados entre os diferentes processadores, o que implica que a solução possa ser mais demorada; mas propiciou uma menor carga computacional para os processadores envolvidos.

Em nossa implementação concluímos que devido a utilização de uma rede de estações de trabalho, e a forma de implementar o algoritmo paralelo, o tempo de execução obtido, para alguns casos foi bem maior do que o tempo de execução obtido no algoritmo sequencial, por causa da comunicação entre os diferentes processadores participantes do procedimento.

Sugerimos como trabalhos futuros:

- Testar nosso algoritmo numa máquina paralela.
- Realizar implementação diferente da que fizemos para o nosso algoritmo, procurando reduzir a comunicação em favor da computação.
- Utilizar uma arquitetura paralela com memória compartilhada e comparar os resultados em quanto ao tempo de execução global do algoritmo paralelo.
- Estudar o efeito das transformações de similaridade não ortogonais em

matrizes não simétricas em relação ao condicionamento das matrizes.

- Estudar o efeito do processo de balanceamento no método de solução proposto neste trabalho.

Bibliografia

[1] Angelika Bunse-Gerstner and Heike Fabbender. A Jacobi-Like Method for Solving Algebraic Riccati Equations on Parallel Computers . *IEEE Transactions on Automatic Control*, 42(8), August 1997.

[2] Angelika Bunse-Gerstner and Volker Mehrmann . A Symplectic QR Like Algorithm for the Solution of the Real Algebraic Riccati Equations . *IEEE Transactions on Automatic Control*, Ac-31(12), December 1986.

[3] Alan J. Laub. A Schur Method for Solving Algebraic Riccati Equations. *IEEE Transactions on Automatic Control*, AC-24(6), December 1979.

[4] A.C. Baleeiro Alves. Uma Introdução à Programação Paralela. Relatório de curso. Publicação interna, FEEC - UNICAMP, 1993.

[5] W.F. Arnold and Alan J. Laub. Generalized Eigenproblem Algorithms and Software for Algebraic Riccati Equations . *Proceedings of the IEEE*, 72(12), December 1984.

[6] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall International Editions, 1989.

[7] C. P. Bottura and J. Tarcisio Costa Filho. On Parallel Programming Environments and Multilevel Optimization. *14 IFIP Conference on Systems Modelling and Optimization. Leipzig, GDR*, July 3-7 1970.

[8] V.C. Barbosa e E.S.T.Fernandez C. Luis de Amorim. *Uma Introdução à Computação Paralela e Distribuída.* VI Escola de Computação. Publicação interna, UNICAMP, 1988.

[9] P.J. Eberlein. On the Schur Decomposition of a Matrix for Parallel Computation. *IEEE Transactions on Computer*, C-36(2), February 1987.

[10] A.M. Fartes Filho. Análise e Computação de Formas Canônicas para Sistemas Lineares Multivariáveis. Tese de Mestrado - UNICAMP, 1977.

[11] J. Tarcisio Costa Filho. Processamento Paralelo de Algoritmos de Controle Ótimo Hierárquico. Tese de Mestrado - UNICAMP, 1988.

[12] J. Tarcisio Costa Filho. Proposta para Computação Assíncrona Paralela e Distribuída de Estruturas Especiais de Jogos Dinâmicos. Tese de Doutorado - UNICAMP, 1992.

[13] A.L.M. França. Computação Paralela Aplicada à Engenharia Elétrica. Notas de aula. Publicação interna, FEEC - UNICAMP.

[14] A. Geist, A. Beguelin, J.J. Dongarra, and et all. *PVM: Parallel Virtual Computer. A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[15] G.H. Golub and J.H. Wilkinson. Ill-Conditioned Eigensystem and the Computation of the Jordan Canonical Form. *SIAM Review*, 18(4), October 1976.

[16] G.H. Golub and Charles F. Van Loan. *Matrix Computations , 2 edition* . The Johns Hopkins University Press, 1989.

[17] D.Watkins Greg Henry and J.J. Dongarra. A Parallel Implementation of the Nonsymetric QR Algorithm for Distributed Memory Architectures. INTERNET, March 1997.

[18] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* International Editions, 1993.

[19] J.D. Gardiner and Alan J. Laub. Parallel Algorithms for Algebraic Riccati Equations. *Int. Contr.*, 54:1317–1333, 1991.

[20] A. Jennings and J.J. McKeown. *Matrix Computations , Second edition* . John Wiley Sons Ltd., 1993.

[21] J.J. Dongarra and M.A. Sidani. A Parallel Algorithm for the Nonsymmetric Eigenvalue Problem . Technical report, Number ORNL/TM-12003, ORNL, Oak Ridge Tennessee, Fevereiro 1991.

[22] A. Cleary L.S. Blackford, J.Choi and et all. *Scalapack Users' Guide.* SIAM, 1997.

[23] R.S. Martin and J.H. Wilkinson. *Similarity Reduction of a General Matrix to Hessenberg Form.* Prepublished in Numer. Math. 12, 349-368, 1968.

[24] T. Harding N. MacDonald, E. Minty and S. Brown. Writing Message-Passing Parallel Programs with MPI. Edinburgh Parallel Computing Centre.

[25] R. Bastos Quirino. Filtro de Kalman: Hierarquização e Computação Paralela. Tese de Mestrado - UNICAMP, 1990.

[26] R.H. Bartels and G.W. Stewart. Algorithm 432 Solution of the Matrix Equation AX+XB=C. *Communications of the ACM*, 15(9), September 1972.

[27] U. Schendel and B.W. Conolly. *Introduction to Numerical methods for Parallel Computers.* Ellis Horwood Limited, 1984.

[28] M. Snir, S. Otto, D.Walker S. H-Lederman, and J.J. Dongarra. *MPI: The Complete Reference.* The MIT Press, 1996.

[29] D.R. Vaughan. A Nonrecursive Algebraic Solution for the Discrete Riccati Equation. *IEEE Transactions on Automatic Control*, AC-15(5), October 1970.

[30] Vlad Ionescu, Cristian Oară and Martin Weiss. General Matrix Pencil Techniques for the Solution of Algebraic Riccati Equations: A unified approach. *IEEE Transactions on Automatic Control*, 42(8), August 1997.

[31] W.T. Vetterling William H. Press, S.A. Teukolsky and B.P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

Apêndice A

Subrotina de Multiplicação Paralela Implementada

Neste apêndice encontra-se parte do código da subrotina de multiplicação de matrizes implementada em forma paralela para resolver a *Equação Algébrica de Riccati Discreta.*

```
      SUBROUTINE MM_pall(a,nra,nca,b,ncb,c,nump,itaf)
      include 'mpif.h'
      parameter (MASTER = 0)
      parameter (FROM_MASTER = 1)
      parameter (FROM_WORKER = 2)
      integer  numtasks,taskid,numworkers,source,dest,mtype,
     &          cols,avecol,extra, offset,i,j,k,ierr
      integer nra, nca, ncb, nump, itaf
      integer status(MPI_STATUS_SIZE)
      real*4 a(NRA,NCA), b(NCA,NCB), c(NRA,NCB)
      numworkers = nump - 1
                              Tarefa mestre
C ---------------------------------------------------------------
      if (itaf.eq. MASTER) then
```

```
      avecol = NCB/numworkers
      extra = mod(NCB, numworkers)

      offset = 1
      mtype = FROM_MASTER
      do 50 dest=1, numworkers
        if (dest .le. extra) then
          cols = avecol + 1
        else
          cols = avecol
        endif
        call MPI_SEND( offset, 1, MPI_INTEGER, dest, mtype,
     &                 MPI_COMM_WORLD, ierr )
        call MPI_SEND( cols, 1, MPI_INTEGER, dest, mtype,
     &                 MPI_COMM_WORLD, ierr )
        call MPI_SEND( a, NRA*NCA, MPI_REAL, dest, mtype,
     &                 MPI_COMM_WORLD, ierr )
        call MPI_SEND( b(1,offset), cols*NCA, MPI_REAL,
     &                 dest, mtype, MPI_COMM_WORLD, ierr )
        offset = offset + cols
 50     continue
      mtype = FROM_WORKER

      do 60 i=1, numworkers
        source = i
        call MPI_RECV( offset, 1, MPI_INTEGER, source,
     &                 mtype, MPI_COMM_WORLD, status, ierr )
        call MPI_RECV( cols, 1, MPI_INTEGER, source,
     &                 mtype, MPI_COMM_WORLD, status, ierr )
        call MPI_RECV( c(1,offset), cols*NRA, MPI_REAL,
     &                 source, mtype, MPI_COMM_WORLD, status, ierr )
```

```
 60      continue
       endif
                              Tarefa escravo
C -----------------------------------------------------------------
       if (itaf .GT. MASTER) then
         mtype = FROM_MASTER
         call MPI_RECV( offset, 1, MPI_INTEGER, MASTER,
     &                  mtype, MPI_COMM_WORLD, status, ierr )
         call MPI_RECV( cols, 1, MPI_INTEGER, MASTER,
     &                  mtype, MPI_COMM_WORLD, status, ierr )
         call MPI_RECV( a, NRA*NCA, MPI_REAL, MASTER,
     &                  mtype, MPI_COMM_WORLD, status, ierr )
         call MPI_RECV( b, cols*NCA, MPI_REAL, MASTER,
     &                  mtype, MPI_COMM_WORLD, status, ierr )
         do 100 k=1, cols
           do 100 i=1, NRA
             c(i,k) = 0.0
             do 100 j=1, NCA
               c(i,k) = c(i,k) + a(i,j) * b(j,k)
  100    continue
         mtype = FROM_WORKER
         call MPI_SEND( offset, 1, MPI_INTEGER, MASTER, mtype,
     &                  MPI_COMM_WORLD, ierr )
         call MPI_SEND( cols, 1, MPI_INTEGER, MASTER, mtype,
     &                  MPI_COMM_WORLD, ierr )
         call MPI_SEND( c, cols*NRA, MPI_REAL, MASTER,
     &                   mtype, MPI_COMM_WORLD, ierr )
       endif
       RETURN
       end
```

Apêndice B

Terminologia Utilizada no Texto

- **TSEE:** Transformações de Similaridade Elementares Estabilizadas;
- **Nó:** Um processador de um computador;
- **id:** Identificador de processos;
- $\lceil - \rceil$ Parte inteira superior do argumento;
- $\lfloor - \rfloor$ Parte inteira inferior do argumento;
- **Mensagem:** Uma quantidade de dados ou instruções que é passada entre nós e o host;
- **Programa Elementar:** Programa constituinte de um programa paralelo que é executado num único nó;

Apêndice C

Definições e fatos básicos

Notação

Neste trabalho, $A \in F^{m \times n}$ denota uma matriz mxn com coeficientes no corpo F. Este corpo pode ser constituido por números reais R ou de números complexos C. Outras notações:

A^T **ou** A^H denota transposta e conjugada transposta, respectivamente;

A^{-T} denota $(A^T)^{-1} = (A^{-1})^T$;

A^+ denota pseudo-inversa da matriz $A \in R^{n \times n}$ (Moore-Penrose);

$D(A)$ denota o conjunto dos n autovalores (spectrum) da matriz $A \in R^{n \times n}$;

A matriz $A \in R^{2n \times 2n}$ é particionada em blocos nxn como

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

onde A_{ij} refere-se aos blocos individuais.

Revisão de Álgebra Linear

Definição 1: $A \in R^{n\times n}$ é ortogonal se $A^T = A^{-1}$.

Definição 2: $A \in C^{n\times n}$ é unitária se $A^H = A^{-1}$.

Seja

$$J = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix} \in R^{2n\times 2n} \tag{C.2}$$

onde I denota a matriz identidade de ordem n. Note que $J^T = J^{-1} = -J$.

Definição 3: $A \in R^{2n\times 2n}$ é Hamiltoniana se $J^{-1}A^TJ = -A$.

Definição 4: $A \in R^{2n\times 2n}$ é Simplética se $J^{-1}A^TJ = A^{-1}$.

As transformações elementares consistem em:

- Multiplicação de uma linha/coluna por um número real diferente de zero.
- Troca de duas linhas/colunas.
- Adicionar o produto de uma linha/coluna e um número a outra linha/coluna.

Estas transformações são feitas utilizando matrizes elementares, que devem ser quadradas e não singulares. Exemplo:

1. $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ 2. $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Os objetivos destas matrizes são reduzir o número de operações computacionais e obter uma matriz triangular como resultado.

Para matrizes reais, *Simétrica* significa o mesmo que *Hermitiana* para matrizes complexas e *Ortogonal*, para matrizes reais significa o mesmo que *Unitária* para matrizes complexas.

Os autovalores de uma matriz Hermitiana são todos reais. Em particular, os autovalores de uma matriz simétrica real são todos reais. Contrariamente o conjunto dos autovalores de uma matriz não simétrica e real pode incluir valores reais mas também pode incluir pares de valores complexos conjugados: por outro lado, autovalores de uma matriz complexa não Hermitiana podem ser complexos.

Apêndice D

O ambiente de simulação

Entende-se por ambiente de simulação a arquitetura de computadores que foi utilizada para o processamento do sistema teste de exemplo. As simulações foram realizadas num computador paralelo virtual formado por computadores conectados numa rede de área local com máquinas *workstations-Sun* do tipo SPARC-20, SPARC-4, SPARC-Classic e SPARC-2. Esta rede pertence ao DMCSI e sua configuração esta apresentada na figura (D.1). Esta rede é constituída de duas sub redes que possuem como ponto de ligação (*Gateway*) entre elas a máquina *Papagaio*. As simulaçòes foram realizadas na sub rede LCSI, utilizando as máquinas *aguia*, que é a servidora desta rede e é o ponto de ligação com a rede da FEEC, *Saturno*, *Urano* e outras.

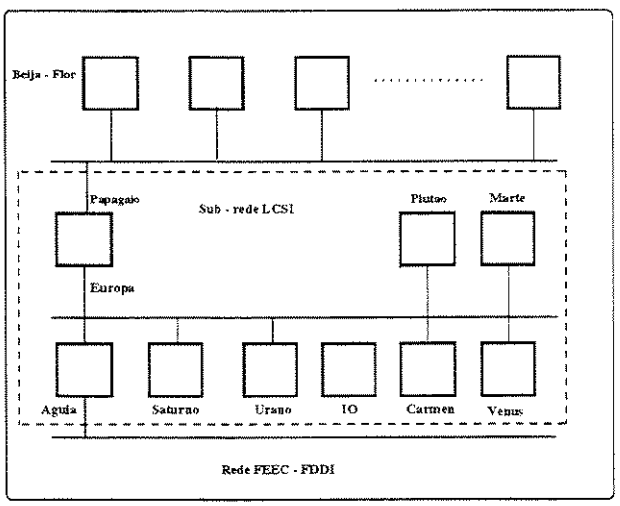


Figura D.1: Rede do DMCSI