



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E
DE COMPUTAÇÃO



UM MÉTODO PARA O CÁLCULO DA INVERSA DE MATRIZES EM BLOCOS COM USO LIMITADO DE MEMÓRIA

Iria Caline Saraiva Cosme

Orientador: Prof. Dr. Samuel Xavier de Souza

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Doutor em Ciências.

Número de Ordem do PPgEEC: D218
Natal/RN, Maio de 2018

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Cosme, Iria Caline Saraiva.

Um método para o cálculo da inversa de matrizes em blocos com uso limitado de memória / Iria Caline Saraiva Cosme. - 2018.
108 f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Programa de Pós-Graduação em Engenharia Elétrica e de Computação. Natal, RN, 2018.

Orientador: Prof. Dr. Samuel Xavier de Souza.

1. Algoritmo - Tese. 2. Matrizes em bloco - Tese. 3. Baixo consumo de memória - Tese. 4. Complemento de Schur - Tese. 5. Inversão de matrizes de grande porte - Tese. I. Souza, Samuel Xavier de. II. Título.

RN/UF/BCZM

CDU 004.421



Universidade Federal do Rio Grande do Norte
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE
COMPUTAÇÃO

ATA Nº 218

Aos quatro dias do mês de maio do ano de dois mil e dezoito, foi realizada a 218ª sessão de defesa de tese de doutorado do Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN, na qual a doutoranda Iria Caline Saraiva Cosme apresentou o trabalho que tem como título: Inversão de matrizes em bloco de forma recursiva com uso limitado de memória. A sessão teve início às 13h30min, tendo a banca examinadora sido constituída pelos seguintes participantes: Samuel Xavier De Souza (Dr. UFRN, Orientador), Adriaio Duarte Doria Neto (Dr. UFRN, Examinador Interno ao Programa), Daniel Aloise (Dr. UFRN, Examinador Interno ao Programa), Luiz Affonso Henderson Guedes De Oliveira (Dr. UFRN, Examinador Interno ao Programa), Aurélio Ribeiro Leite De Oliveira (Dr. UNICAMP, Examinador Externo à Instituição) e Francisco Chagas De Lima Junior (Dr. UERN, Examinador Externo à Instituição). Após a apresentação do trabalho e o exame pela banca, o doutorando foi considerado aprovado, tendo sido lavrada a presente ata, que vai assinada pelos examinadores e pelo doutorando. A versão final da tese deverá ser entregue ao programa, no prazo máximo de 60 dias, contendo as modificações sugeridas pela banca examinadora e constante na folha de correção anexa. Conforme o Artigo 49 da Resolução 197/2013 - CONSEPE, o candidato não terá o título se não cumprir as exigências acima.

Aurelio RL de Oliveira

Dr. AURELIO RIBEIRO LEITE DE OLIVEIRA, UNICAMP

Examinador Externo à Instituição

Francisco Chagas de Lima Junior

Dr. FRANCISCO CHAGAS DE LIMA JUNIOR, UERN

Examinador Externo à Instituição

Adriaio Duarte Doria Neto

Dr. ADRIAIO DUARTE DORIA NETO, UFRN

Examinador Interno

Daniel Aloise

Dr. DANIEL ALOISE, UFRN

Examinador Interno

Luiz Affonso Henderson Guedes de Oliveira
Dr. LUIZ AFFONSO HENDERSON GUEDES DE OLIVEIRA, UFRN

Examinador Interno

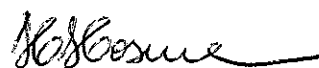


Universidade Federal do Rio Grande do Norte

**PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE
COMPUTAÇÃO**


Dr. SAMUEL XAVIER DE SOUZA, UFRN

Presidente


IRIA CALINE SARAIVA COSME

Doutorando

Resumo

A inversão de matrizes de ordem extremamente alta tem sido uma tarefa desafiadora devido ao processamento e à capacidade de memória limitados dos computadores convencionais. Em um cenário em que os dados não cabem na memória, é oportuno considerar a troca de mais tempo de processamento por menos uso de memória para permitir a computação da inversa matricial, o que seria proibitivo de outra forma.

Sendo assim, este trabalho apresenta um novo algoritmo para o cálculo da inversa de matrizes particionadas em blocos com uso reduzido de memória. O algoritmo funciona de forma recursiva para inverter um bloco de uma matriz $M^{k \times k}$, com $k \geq 2$, com base na divisão de M , sucessivamente, em matrizes de menor ordem. Este algoritmo, denominado BRI (do inglês, *Block Recursive Inversion*), calcula um bloco da matriz inversa por vez para limitar o uso da memória durante todo o processamento.

Considerando que o baixo consumo de memória, proporcionado pelo BRI, é contrabalanceado por um maior tempo de processamento, este trabalho também discorre sobre uma implementação paralela, em OpenMP, do algoritmo a fim de reduzir o tempo de processamento e ampliar sua aplicabilidade. Além disso, uma melhoria no algoritmo sequencial é proposta.

Como aplicação prática, o algoritmo proposto foi utilizado no processo de validação cruzada para Máquinas de Vetor de Suporte por Mínimos Quadrados (LS-SVM, do inglês, *Least Squares Support Vector Machines*). Este procedimento computacional utiliza o cálculo da matriz inversa para encontrar os rótulos esperados das amostras de testes na validação cruzada.

Os resultados experimentais com o BRI demonstraram que, apesar do aumento da complexidade computacional, matrizes, que de outra forma excederiam o limite de uso da memória, podem ser invertidas usando esta técnica.

Palavras-chave: Matrizes em bloco. Baixo consumo de memória. Complemento de Schur. Inversão de matrizes de grande porte.

Abstract

The inversion of extremely high order matrices has been a challenging task because of the limited processing and memory capacity of conventional computers. In a scenario in which the data does not fit in memory, it is worth to consider exchanging more processing time for less memory usage in order to enable the computation of the inverse, which otherwise would be prohibitive.

Therefore, this work introduces a novel algorithm to compute the inverse of block partitioned matrices with a reduced memory footprint. The algorithm works recursively to invert one block of a $k \times k$ block matrix M , with $k \geq 2$, based on the successive splitting of M into lower order matrices. This algorithm, called Block Recursive Inverse (BRI), computes one block of the inverse at a time to limit memory usage during the entire processing.

Considering that the low memory consumption, provided by the BRI, is counterbalanced by longer processing time, this work also discusses a parallel implementation of the algorithm in OpenMP to reduce the running time and to extend its applicability. Additionally, an improvement in the sequential algorithm is proposed.

As a practical application, the proposed algorithm was applied in the cross-validation process for Least Squares Support Vector Machines (LS-SVM). This computational procedure uses the inverse matrix calculation to find the expected labels of the test samples in the cross-validation.

Experimental results with BRI show that, despite increasing computational complexity, matrices that otherwise would exceed the memory-usage limit can be inverted using this technique.

Keywords: Block matrices. Low memory usage. Schur Complement. Large-scale matrix inversion.

Sumário

Sumário	i
Lista de Figuras	iii
Lista de Tabelas	v
Lista de Símbolos e Abreviaturas	vii
1 Introdução	1
1.1 Motivação	1
1.2 Relevância da Pesquisa	4
1.3 Objetivos	6
1.4 Organização da tese	7
2 Estado da Arte	9
2.1 Trabalhos Relacionados	9
3 Referencial Teórico	13
3.1 Matriz em Blocos	13
3.1.1 Terminologia	14
3.2 Complemento de Schur	15
3.2.1 Definição	16
3.2.2 Fórmula da inversão de matrizes em blocos 2×2	17
3.3 Memória	18
4 Método de Inversão Recursiva de Matrizes em Blocos	21
4.1 Algoritmo de Inversão Recursiva por Blocos	21
4.1.1 Procedimento Recursivo Progressivo	22
4.1.2 Procedimento Recursivo Retroativo	26
4.2 Um Exemplo: Inversa de matrizes em blocos 4×4	31
4.3 Implementação	36

4.4	Análise de Complexidade	37
4.4.1	Análise do Custo de Memória	38
4.5	Resultados Experimentais	39
4.5.1	Uso de Memória	40
4.5.2	Tempo de Execução	42
5	BRI Aplicado à Validação Cruzada de LS-SVM	43
5.1	Máquinas de Vetor de Suporte por Mínimos Quadrados	45
5.1.1	Classificação de Padrões em LS-SVM	45
5.1.2	Funções de <i>Kernel</i>	48
5.2	Validação Cruzada (<i>Cross-Validation</i>)	49
5.3	Trabalhos Relacionados	50
5.4	Algoritmo BRI aplicado à Validação Cruzada de LS-SVM	52
5.4.1	Definições	52
5.4.2	Algoritmo CV usando BRI	54
5.5	Implementação	56
5.6	Análise de Complexidade	58
5.6.1	Análise do Custo de Memória	59
5.7	Resultados Experimentais	60
5.7.1	Uso de Memória	61
5.7.2	Tempo de Execução	63
6	BRI Paralelo para Arquiteturas de Memória Compartilhada	65
6.1	Programação Paralela	66
6.1.1	OpenMP	68
6.1.2	Análise de Desempenho de Algoritmos Paralelos	69
6.2	Trabalhos Relacionados	70
6.3	Otimização do Algoritmo BRI	71
6.3.1	Resultados Experimentais	72
6.4	Paralelização do Algoritmo BRI	73
6.4.1	Resultados Experimentais	76
7	Considerações Finais	85
	Referências bibliográficas	87

Lista de Figuras

3.1	Hieraquia de memória de um sistema computacional. Fonte: [Maziero 2014]	18
4.1	Vetor linha e vetor coluna.	28
4.2	Quadros $M\langle_A, M\langle_B, M\langle_C$ e $M\langle_D$ resultantes das divisões de uma matriz M	32
4.3	Divisões ocorridas no quadro $M\langle_B$, no Procedimento Recursivo Progressivo.	33
4.4	Procedimento Recursivo Retroativo nos quadros originados por $M\langle_B$ até a obtenção de $\langle\langle M\langle_B \rangle\rangle$	35
4.5	Procedimento Recursivo Progressivo	36
4.6	Procedimento Recursivo Retroativo	36
4.7	Uso de memória para a computação da inversa de matrizes $m \times m$ usando decomposição LU e o algoritmo BRI com diferentes números de blocos.	41
4.8	Tempo de execução da computação da inversa de matrizes $m \times m$ usando decomposição LU e o algoritmo BRI com diferentes números de blocos.	42
5.1	Hiperplano de separação ótimo.	46
5.2	Maximização da margem de separação. Os vetores em destaque correspondem aos vetores de suporte.	47
5.3	Método de validação cruzada utilizando l -partições.	50
5.4	Comparação do pico do uso de memória na validação cruzada de LS-SVM com o algoritmo BRI e com a inversão LU, para diferentes l partições.	62
5.5	Comparação do tempo de execução da validação cruzada de LS-SVM eficiente utilizando o algoritmo BRI e a inversão LU, para diferentes l partições.	63
6.1	Arquitetura com memória compartilhada Fonte [Pacheco 2011].	67
6.2	Arquitetura com memória compartilhada [Pacheco 2011].	67
6.3	Tempo de execução da computação da inversa de matrizes $m \times m$ usando o algoritmo BRI original e uma versão otimizada.	73
6.4	Ilustração da árvore de chamadas recursivas do algoritmo BRI.	74

6.5	Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 2$, utilizando diferentes grupos de <i>threads</i> (p).	78
6.6	Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 3$, utilizando diferentes grupos de <i>threads</i> (p).	79
6.7	Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 4$, utilizando diferentes grupos de <i>threads</i> (p).	80
6.8	Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 5$, utilizando diferentes grupos de <i>threads</i> (p).	81
6.9	Curvas do <i>speedup</i> em função do tamanho de grupo de <i>threads</i>	82
6.10	Curvas da eficiência em função do tamanho de grupo de <i>threads</i>	83
6.11	Eficiência para diferentes matrizes de entrada com $k = 3$	83
6.12	Eficiência para diferentes matrizes de entrada com $k = 4$	84
6.13	Eficiência para diferentes matrizes de entrada com $k = 5$	84

Lista de Tabelas

5.1	Algumas funções usadas como kernel	48
6.1	Tempos de processamento do BRI sequencial e do BRI paralelo	77

Lista de Símbolos e Abreviaturas

l-fold CV: Validação Cruzada de *l*-partições – do inglês, *l-Fold Cross Validation*

UFRN: Universidade Federal do Rio Grande do Norte

API: Interface de Programação – do inglês, *Application Programming Interface*

BRI: Algoritmo de Inversão Recursiva de Matrizes em Blocos – do inglês, *Block Recursive Inversion algorithm*

crossBRI: Algoritmo *l-fold* VC eficiente com uso do BRI

crossLU: Algoritmo *l-fold* VC eficiente tradicional, com uso da inversa por fatoração LU

CV: Validação Cruzada – do inglês, *Cross Validation*

GPU: Unidade de Processamento Gráfico – do inglês, *Graphics Processing Unit*

LMO-CV: Validação Cruzada 'deixe-mais-fora' – do inglês, *Leave-More-Out Cross Validation*

LNO-CV: Validação Cruzada 'deixe-N-fora' – do inglês, *Leave-N-Out Cross Validation*

LOO-CV: Validação Cruzada 'deixe-um-de-fora' – do inglês, *Leave-one-out Cross Validation*

LS-SVM: Máquina de Vetor de Suporte por Mínimos Quadrados – do inglês, *Least Squares Support Vector Machine*

MIMD: Múltiplas Instruções, Múltiplos Dados – do inglês, *Multiple Instruction, Multiple Data*

MISD: Múltiplas Instruções, Único Dado – do inglês, *Multiple Instruction, Single Data*

Monte-Carlo LNO-CV: Monte-Carlo Validação Cruzada 'deixe-mais-fora' – do inglês, *Monte-Carlo Leave-More-Out Cross Validation*

MSE: Erro Quadrático Médio – do inglês, *Mean Squared Error*

NPAD: Núcleo de Processamento de Alto Desempenho

RAM: Memória de Acesso Aleatório – do inglês, *Random-Access Memory*

SIMD: Instrução Única, Múltiplos Dados – do inglês, *Single Instruction, Multiple Data*

SISD: Instrução Única, Único Dado – do inglês, *Single Instruction, Single Data*

SVM: Máquina de Vetor de Suporte – do inglês, *Support Vector Machine*

Capítulo 1

Introdução

A evolução dos computadores, associada aos investimentos em tecnologia digital e microeletrônica, deu origem a equipamentos computacionais mais compactos e com maior desempenho. Paulatinamente, o uso do computador se mostrou bastante eficiente em aplicações que requerem alto poder computacional como, por exemplo, previsão do tempo, processamento de imagens e alinhamento de sequências de DNA. Desta forma, os *softwares* foram se tornando mais complexos e, conseqüentemente, demandando mais recursos computacionais, especialmente, memória para armazenamento e processamento de informações.

O século XXI tem se destacado em como a humanidade aprendeu o valor de armazenar grandes quantidades de dados, a fim de fazer previsões e tomar decisões mais assertivas [Chandorkar et al. 2015]. Atualmente, trabalhar com grandes conjuntos de dados (popularmente denominado *big data*) é uma situação cada vez mais comum devido aos avanços nas tecnologias de comunicação e sensores, bem como à evolução de dados digitais. Conforme Chen et al. (2014), essa nova tendência tecnológica representa os conjuntos de dados que não poderiam ser armazenados, gerenciados e processados por ferramentas tradicionais de software e hardware em um tempo tolerável. Isto tem se tornado uma tarefa desafiadora porque os dados manipulados excedem os tamanhos convencionais e são frequentemente coletados a uma velocidade superior à capacidade de processamento e armazenamento em memória dos computadores convencionais.

Nesse cenário atual, métodos novos e aprimorados para o tratamento de dados vêm sendo cada vez mais estimulados.

1.1 Motivação

A inversão de matrizes é uma tarefa computacional necessária em muitas aplicações científicas, como processamento de sinais, análise de redes complexas, estatísti-

cas [McCullagh & Nelder 1989] e em alguns problemas envolvendo autovalores [Byers 1987], para citar apenas alguns. Existem algoritmos que podem ser usados na inversão de matrizes comumente disponíveis para matrizes não-singulares, como eliminação gaussiana, Gauss-Jordan, decomposição LU e decomposição de Cholesky. Porém, a maioria deles é computacionalmente intensiva em uso de memória e processador. Por exemplo, calcular o inverso de uma matriz $n \times n$ com o método Gauss-Jordan possui complexidade computacional de $O(n^3)$ e complexidade de armazenamento de memória de $O(n^2)$. Este fato pode tornar proibitiva a aplicabilidade de tais métodos em matrizes de grande porte, principalmente, porque os dados simplesmente poderiam não caber na memória.

Em problemas de classificação (ou regressão) [An et al. 2007], a título de exemplo, inverter matrizes de ordem extremamente alta pode exceder a capacidade da memória do computador apenas para armazenar sua inversa. Muitos algoritmos foram, então, desenvolvidos com o objetivo de acelerar o processamento de cálculos com matrizes de dados de grande porte.

Neste contexto, uma abordagem que tem se mostrado bastante promissora é a utilização de matrizes particionadas em blocos a fim de diminuir a complexidade e, consequentemente, reduzir o tempo de processamento de cálculos matriciais. Conforme Golub & Van Loan (2013), algoritmos formulados no nível de bloco são tipicamente ricos em multiplicação entre matrizes, a qual é uma operação desejável em muitos ambientes computacionais de alto desempenho. Considerando que, às vezes, a estrutura de blocos de uma matriz é recursiva, as entradas em blocos possuem uma similaridade explorável com a matriz no geral. Este tipo de conexão é a base para algoritmos de produtos matriz-vetor, ditos mais rápidos, tais como transformadas rápidas de Fourier, transformadas trigonométricas e transformadas *Wavelet* [Golub & Van Loan 2013].

As matrizes em blocos podem surgir, naturalmente, devido à ordenação de equações e variáveis em uma ampla variedade de aplicações científicas e de engenharia, como na equação Navier-Stokes incompressível [Elman et al. 2008], aproximação numérica de equações diferenciais parciais elípticas por elementos finitos mistos [Brezzi & Fortin 1991], controle ótimo [Betts 2001], redes elétricas [Björck 1996] e no algoritmo de Strassen [Strassen 1969] para a multiplicação rápida de matrizes. Através do uso do conceito de matrizes em blocos, Strassen conseguiu reduzir a complexidade dos algoritmos triviais de multiplicação de matrizes de $O(n^3)$ para $O(n^{2,807})$, sendo n a ordem das matrizes de entrada. De acordo com Golub & Van Loan (2013), algoritmos que manipulam matrizes no nível do bloco são muitas vezes mais eficientes porque são mais abundantes em operações BLAS de nível 3 e, portanto, podem ser implementados de forma recur-

siva¹.

Existem alguns artigos que tratam a inversão matricial e/ou resolução de sistemas lineares através de matrizes subdividas em blocos, a fim de distribuir os cálculos envolvidos em partes menores que podem ser mais tratáveis em cálculos de grande porte. Em [Lu & Shiou 2002], os autores fornecem fórmulas para a inversão de matrizes em blocos 2×2 aplicadas em matrizes triangulares em blocos e várias matrizes estruturadas, como matrizes hamiltonianas, hermitianas e centro-hermitianas. A inversão das matrizes circulantes de blocos foi amplamente investigada em [Baker et al. 1988] e [Vescovo 1997]. Além disso, em [Karimi & Zali 2014], os autores propõem um novo pré-condicionador triangular de blocos para resolver sistemas de equações lineares particionados em blocos.

Do mesmo modo, os algoritmos recursivos têm sido aplicados em [Baker et al. 1988], [Madsen 1983] e [Tsitsas et al. 2007] para a inversão de casos particulares de matrizes, tais como matrizes circulantes. Madsen (1983) propõe um método recursivo para a decomposição LU de uma matriz circulante simétrica real. Em [Baker et al. 1988], um algoritmo recursivo é aplicado para calcular a primeira linha de blocos da inversa de uma matriz circulante de blocos com blocos circulantes. E, finalmente, em [Tsitsas et al. 2007], os autores propõem um algoritmo recursivo baseado na inversão de matrizes em blocos $k \times k$ para casos de matrizes com blocos circulantes com base na diagonalização anterior de cada bloco circulante.

Além disso, Tsitsas (2010) propõe um método eficiente para a inversão de matrizes com blocos U -diagonalizáveis (sendo U uma matriz unitária fixa) utilizando a U -diagonalização de cada bloco e, subsequentemente, um procedimento de transformação de similaridade. Esta abordagem permite obter o inverso de matrizes com blocos U -diagonalizáveis sem ter que assumir a invertibilidade dos blocos envolvidos no procedimento, desde que determinadas condições sejam atendidas.

Apesar desses numerosos esforços, a inversão de grandes matrizes densas é ainda um desafio para grandes conjuntos de dados. Em um cenário no qual os dados não cabem na memória, trocar mais tempo de processamento por menos uso de memória poderia ser uma alternativa para permitir o cálculo da inversa que de outra forma seria proibitivo.

Motivado pelas considerações precedentes, a presente tese propõe um método recursivo para a inversão de uma matriz M em blocos $k \times k$, $M \in \mathbb{R}^{m \times m}$, com blocos quadrados de ordem b . A ideia básica desse algoritmo, chamado de Inversão Recursiva por Bloco (BRI), está na determinação de um bloco da matriz inversa por vez. O método baseia-se na

¹Operações BLAS de nível 3, categorizadas pela biblioteca BLAS (do inglês, *Basic Linear Algebra Subprograms*), disponibilizam rotinas altamente otimizadas para operações entre matrizes, tais como multiplicação entre duas matrizes e fatoração LU. Em geral, atuam em cima de $O(n^2)$ dados de entrada, o que garante um volume de cálculos maior que o número de acessos à memória.[Kågström et al. 1998]

divisão sucessiva da matriz original em quatro matrizes quadradas de uma ordem inferior, denominadas quadros. Para isso, são consideradas duas etapas, a saber, o procedimento recursivo progressivo e o procedimento recursivo retroativo. O procedimento recursivo progressivo termina após $k - 2$ passos, quando os quadros resultantes têm blocos 2×2 . Posteriormente, no procedimento recursivo retroativo para cada 4 quadros gerados, operações são realizadas reduzindo-os a um único bloco. Diferentemente daqueles propostos em [Baker et al. 1988, Madsen 1983, Tsitsas et al. 2007], esse algoritmo recursivo pode ser aplicado para inverter qualquer matriz $M \in \mathbb{R}^{m \times m}$, desde que M seja não-singular e que todas as submatrizes que precisam ser invertidas no procedimento recursivo retroativo também sejam não-singulares.

1.2 Relevância da Pesquisa

Como apresentado anteriormente, muitas aplicações científicas dependem da inversão de matrizes para atingir seus objetivos. Contudo, no atual contexto em que é cada vez mais comum a manipulação de grandes quantidades de dados, a tarefa de calcular a inversa de matrizes de ordem extremamente alta tem se tornado um grande desafio tanto em relação a tempo de processamento, quanto em relação ao consumo de memória do sistema computacional.

Segundo Liu et al. (2016), os algoritmos de inversão de matrizes comumente disponíveis não são aplicáveis para inverter matrizes de grande porte, muitas vezes necessárias em aplicações de *big data* emergentes. Por exemplo, em redes sociais (como Facebook e Twitter), sites de comércio eletrônico (como Amazon e Ebay) e provedores de serviços de vídeo *on-line* (como o Youtube e o Netflix), vários tipos de matrizes, incluindo matrizes de seguidores, matrizes de transação e matrizes de classificação, contêm milhões de usuários e itens distintos. A inversão dessas matrizes de grande porte é uma operação fundamental para medição de proximidade, previsão de links e tarefas de recomendação personalizada.

Observa-se também que mesmo que algoritmos convencionais de inversão matricial executem experimentos em conjuntos de dados de grande porte, a escalabilidade desses métodos tem se mostrado restrita à memória disponível em uma única máquina. Quando se trata de projetar um cálculo com uma matriz de grande porte, não é suficiente simplesmente minimizar os FLOPS². É necessário também que seja dada atenção especial

²Em computação, as operações de ponto flutuante por segundo – FLOPS (do inglês, *FL*oating *point* *Operations Per Second*) são uma unidade de medida que serve para mensurar a capacidade de processamento de um computador, neste caso, a quantidade de operações de ponto flutuante.

à forma como as unidades aritméticas interagem com o sistema de memória subjacente [Golub & Van Loan 2013]. Sendo assim, a maneira de se estruturar os dados em uma matriz torna-se uma parte importante do processo, porque nem todos os *layouts* de matriz são ajustáveis à memória disponível. Neste contexto, o armazenamento dos dados afeta a eficiência mais do que o volume da aritmética real.

Como uma maneira de abordar este problema, foi implementado um novo algoritmo recursivo para a inversão de matrizes densas de alta ordem, $M \in \mathbb{R}^{m \times m}$, a partir dos conceitos de complemento de Schur e matrizes em blocos $k \times k$, com blocos quadrados de ordem b , que limita o uso de memória durante todo o seu processamento. Essa abordagem de inversão matricial conduz a um *trade-off* entre uso da memória e tempo de processamento, induzindo ao fato que é possível trocar o aumento no tempo de processamento pela diminuição no uso de memória ao variar a quantidade de blocos, durante todo o cálculo da inversa e, conseqüentemente, permitir a inversão de matrizes muito grandes que, de outra forma, não caberiam na memória.

A implementação do algoritmo proposto, que possibilita o processo de inversão matricial a partir do conceito de blocos da matriz em questão, permite que este processo possa ser realizado por processamento paralelo. Neste caso, é possível utilizar processadores individuais para efetuar os cálculos com matrizes menores (blocos) a fim de, posteriormente, combinar os resultados. Isso, além de claramente reduzir o tempo de execução do algoritmo, possibilita efetuar os cálculos aritméticos necessários trabalhando com matrizes menores na memória de alta velocidade do sistema computacional [Anton & Busby 2006], o que resulta na possibilidade de se alcançar velocidade adicional ao processo de inversão como um todo.

Buscando um enfoque prático e objetivando analisar os benefícios do algoritmo proposto, este foi aplicado no método de validação cruzada do tipo l -partições (do inglês, *l-folds*) para Máquinas de Vetor de Suporte por Mínimos Quadrado (LS-SVM), proposto por [An et al. 2007], cujo processo é centrado no cálculo da inversa da matriz de *kernel* (K_Y^{-1}) e na resolução de l sistemas de equações, onde l é o número de partições, para estimar os rótulos previstos nos subconjuntos de testes omitidos, sem a necessidade de treinar os classificadores para cada subconjunto de treinamento. Contudo, esta matriz de *kernel* se apresenta como sendo de alta ordem para grandes conjuntos de dados de entrada. Sendo assim, a escalabilidade deste método fica restrita à memória disponível pelo sistema computacional. Então, a fim de possibilitar um melhor ajuste dos dados na memória, o algoritmo proposto nesta tese, foi aplicado para calcular tanto os blocos diagonais, C_{kk} quanto os demais blocos da matriz K_Y^{-1} , durante a execução do algoritmo de validação cruzada para grandes conjuntos de amostras de entrada em LS-SVMs.

A análise dos algoritmos propostos nesta tese envolve uma abordagem teórica e experimental, a partir da qual os programas implementados foram testados variando os tamanhos e composição do conjunto dos dados de entrada. O desempenho do algoritmo foi mensurado a partir do tempo de execução, bem como do pico de memória utilizada durante a execução do mesmo. Contudo, tendo em vista que a proposta desta tese objetiva o uso limitado de memória, buscou-se uma redução da memória utilizada em detrimento da quantidade de processamento necessária.

1.3 Objetivos

O principal objetivo desta tese foi investigar e propor uma nova abordagem recursiva de blocos para dividir o excessivo cálculo envolvido no processo de inversão de uma matriz de alta ordem em um conjunto de pequenas tarefas que pudessem ser armazenadas e manipuladas individualmente. Desta forma, o algoritmo proposto poderá ser aplicado em contextos que demandem a manipulação de matrizes de grande porte, mesmo em sistemas computacionais com restrições de memória. Para tanto, considerou-se uma troca entre o aumento no tempo total de processamento pela redução no uso de memória, ao variar a quantidade de blocos da matriz.

Para alcançar este objetivo geral, os seguintes objetivos específicos foram buscados:

- Introduzir conceitos matemáticos subjacentes para o processo de inversão de matriz de alto desempenho como, por exemplo, estrutura de dados orientada a blocos para um acesso eficiente aos elementos da matriz, e aplicação do conceito de complemento de Schur;
- Implementar e adaptar o algoritmo de validação cruzada *l-fold* para LS-SVM, proposto por An et al. (2007), para fazer uso do algoritmo do cálculo da matriz inversa proposta, de maneira a permitir a sua efetiva execução com grandes números de amostras de dados que, de outra forma, excederia o limite de uso da memória disponível;
- Avaliar experimentalmente a complexidade computacional e custo de memória dos algoritmos envolvidos;
- Investigar os algoritmos analisados em relação ao uso de memória e tempo de processamento;
- Investigar a redução do tempo de processamento do algoritmo proposto através do uso de técnicas de paralelização;
- Analisar a eficiência do algoritmo paralelo utilizando métricas de paralelismo.

1.4 Organização da tese

Para um melhor entendimento do escopo do assunto abordado, organizamos a escrita da tese como se segue. O Capítulo 2 apresenta o estado da arte em relação às técnicas de inversão de matrizes relacionando-o à temática em análise. O Capítulo 3 expõe definições e notações básicas acerca da subdivisão de matrizes em blocos, além de introduz o conceito de complemento de Schur e outras notações relevantes para esta tese.

O novo algoritmo recursivo em blocos é tratado no Capítulo 4, juntamente com detalhes de sua implementação. Além disso, é realizada uma análise da complexidade do algoritmo proposto, incluindo uma investigação do consumo de memória. Resultados experimentais também são discutidos.

Numa abordagem prática, no Capítulo 5, o algoritmo proposto é aplicado ao processo de validação cruzada *l-fold* das LS-SVMs. Os resultados experimentais dão margem às discussões.

O Capítulo 6 aborda uma otimização do algoritmo proposto, além de tratar de uma implementação paralelizada do mesmo. Também é feita uma análise de desempenho através dos resultados obtidos nos testes. Por fim, o Capítulo 7 expõe algumas conclusões e considerações finais.

Capítulo 2

Estado da Arte

Um sistema linear pode ser expresso na forma: $Ax = b$, sendo A uma matriz $n \times n$, x um vetor $n \times 1$, contendo as incógnitas do sistema, e b um vetor $n \times 1$ de termos independentes. Neste caso, o sistema é composto por n equações com n incógnitas. Sistemas Lineares cujo valor de n é elevado são ditos de grande porte [Claudio & Marins 2000]. Esta equação pode ser resolvida calculando a inversa da matriz A , denotada por A^{-1} . Assim, sendo A invertível, obtém-se $x = A^{-1} \times b$. Além disso, a tarefa computacional de inversão de matrizes é essencial em muitas aplicações científicas, tais como foram citadas no Capítulo 1.

Desta forma, no presente capítulo, são apresentados os trabalhos científicos que têm abordado o assunto da inversa matricial relacionando-os à temática em análise.

2.1 Trabalhos Relacionados

Para matrizes gerais, existem alguns algoritmos de inversão matricial comumente disponíveis, como eliminação de Gauss, método de Gauss-Jordan [Althoen & McLaughlin 1987] e decomposição LU [Press et al. 2007]. No entanto, esses algoritmos são computacionalmente intensivos, os quais requerem um número cúbico de operações e armazenamento de memória de $O(n^2)$. Portanto, eles não são aplicáveis para inverter matrizes arbitrariamente grandes, muitas vezes necessárias em aplicações emergentes de *big data*.

Devido ao seu papel fundamental na computação científica, a inversão de matrizes é amplamente suportada em diversos softwares de análise numérica como o Matlab, LAPACK [Anderson et al. 1990], LINPACK [Dongarra & Luszczek 2011] e R¹. Embora esses softwares forneçam recursos básicos de inversão de matrizes para resolver equações lineares, eles apresentam problemas de desempenho quando a ordem da matriz, a ser invertida, se torna muito grande.

¹Para mais informações, acesse <https://www.r-project.org/>

Objetivando dividir os cálculos envolvidos em partes menores que possam ser mais tratáveis, alguns trabalhos lidam com a inversão matricial e/ou resolução de sistemas lineares a partir de matrizes subdividas em blocos.

Em [Lu & Shiou 2002], os autores derivam várias fórmulas para a inversão de matrizes em blocos 2×2 , considerando três diferentes partições, e as aplicam para obter inversas de matrizes triangulares em bloco e de várias matrizes estruturadas, como matrizes hamiltonianas, hermitianas e centro-hermitianas. Vescovo (1997) apresenta dois procedimentos alternativos para derivar as fórmulas de inversão de matrizes circulantes de blocos, baseadas em uma abordagem segundo a Transformada Discreta de Fourier. Em [Karimi & Zali 2014], por sua vez, os autores propõem um novo pré-condicionador triangular de blocos para resolver sistemas de equações lineares particionados em blocos.

Do mesmo modo, os algoritmos recursivos têm sido aplicados em [Baker et al. 1988], [Madsen 1983] e [Tsitsas et al. 2007] para a inversão de casos particulares de matrizes, tais como matrizes circulantes. Madsen (1983) propõe um método recursivo para a decomposição LU de uma matriz circulante simétrica real. Em [Baker et al. 1988], um algoritmo recursivo é aplicado para determinar a primeira linha de blocos da inversa de uma matriz circulante de blocos com blocos circulantes. Já em [Tsitsas et al. 2007], os autores propõem uma inversão recursiva para matrizes com blocos circulantes baseado na diagonalização de cada bloco circulante por meio da Transformada Discreta de Fourier e, posterior, aplicação de um algoritmo recursivo para a inversão da matriz em blocos, determinada pelos autovalores de cada bloco diagonal.

Além desses, Tsitsas (2010) propôs um método eficiente para a inversão de matrizes com blocos U -diagonalizáveis (sendo U uma matriz unitária fixa) utilizando a U -diagonalização de cada bloco e, subsequentemente, um procedimento de transformação de similaridade. Esta abordagem permite obter o inverso de matrizes com blocos U -diagonalizáveis sem ter que assumir a invertibilidade dos blocos envolvidos no procedimento, desde que determinadas condições sejam atendidas.

Entretanto, todos esses algoritmos são aplicados para casos particulares de matrizes e não trabalham com matrizes gerais.

Outra forma de abordar o problema de inverter matrizes de ordem extremamente alta tem sido utilizando técnicas de computação paralela. Assim, Lau et al. (1996) apresentaram dois algoritmos baseados na eliminação de Gauss para inverter matrizes simétricas, positivas definidas e esparsas em computadores paralelos, e os implementou em computadores SIMD (Instrução Única, Múltiplos Dados – do inglês, *Single Instruction, Multiple Data*) e MIMD (Múltiplas Instruções, Múltiplos Dados – do inglês, *Multiple Instruction, Multiple Data*). Bientinesi et al. (2008) introduziram um algoritmo para inverter uma

matriz definida positiva simétrica baseada na fatoração de Cholesky. No entanto, esses algoritmos não se aplicam para matrizes gerais.

Mahfoudhi et al. (2017) projetaram dois algoritmos baseados na fatoração LU recursiva e no método de Strassen para inverter matrizes quadradas densas em uma arquitetura de computadores com processadores *do inglês, multicore*, ou seja, com múltiplos núcleos de processamento. Os resultados experimentais mostraram que as implementações apresentaram um bom desempenho. Contudo, no retorno da recursividade, os blocos manipulados por ambos algoritmos aumentam progressivamente de tamanho até obter a ordem de $\frac{n}{2}$, o que não é desejável considerando que n é de alta ordem.

De outro modo, em [Ezzatti et al. 2011], os autores implementaram um algoritmo de inversão de matrizes em blocos baseado na eliminação de Gauss-Jordan em uma arquitetura híbrida consistindo em um (ou mais) processadores *multicore* conectados a várias GPUs (Unidades de Processamento Gráfico).

No contexto de tecnologias de computação em nuvem, algumas plataformas de processamento de dados foram desenvolvidas para processamento de dados em larga escala, tais como MapReduce [Dean & Ghemawat 2008] e Spark [Zaharia et al. 2010]. Assim, consequentemente, surgiram pesquisas a fim de desenvolver funcionalidades de Álgebra Linear nessas plataformas. Xiang et al. (2014) propuseram e implementaram uma técnica recursiva de blocos para inversão de matrizes em MapReduce baseada na fatoração LU, enquanto que, Liu et al. (2016) apresentaram um algoritmo para obter inversas recursivas de matrizes de grande porte em *clusters*, também baseado em fatoração LU, porém implementada em Spark. Contudo, assim como em [Mahfoudhi et al. 2017], no retorno da recursividade, os blocos aumentam progressivamente de tamanho até obter uma ordem de, igual ou aproximadamente, $\frac{n}{2} \times \frac{n}{2}$.

O uso de computadores *multicore* para computação paralela de propósito geral tem sido bastante abordado na literatura, principalmente com o foco no *speedup* que é possível obter nessas máquinas. Contudo, a evolução da memória principal tem sido inferior à dos processadores, considerando que, em média, eles evoluem 10% e 60% ao ano, respectivamente [Patterson & Hennessy 2013]. Deste modo, a capacidade dos processadores é constantemente limitada por acessos à memória, que possuem frequência de operação inferior, forçando, então, o processador a ficar ocioso. Além disso, a limitação do espaço disponível da própria memória reduz o desempenho do algoritmo executado.

Por outro lado, o uso de *clusters* de computadores podem ser uma alternativa para os casos de limitação de memória [Sadashiv & Kumar 2011]. Desta forma, o uso de computadores comuns, com memórias distribuídas, possibilitaria o aumento de desempenho desses algoritmos. Entretanto, a infraestrutura necessária para acomodar uma grande

quantidade de máquinas termina por gerar outras dificuldades, como por exemplo, problema de espaço para instalação dessas máquinas, bem como o elevado consumo de energia por parte do sistema computacional.

Sendo assim, é salutar considerar a possibilidade de abordar esse problema de inversão de matrizes densas arbitrariamente grandes a partir de algoritmos que limitem o uso de memória durante todo o seu processamento.

Capítulo 3

Referencial Teórico

A técnica de subdivisão de matrizes é, geralmente, utilizada a fim de isolar partes da matriz que podem ser importantes em problemas particulares ou para particionar uma matriz de grande porte em blocos menores que podem ser mais tratáveis em cálculos de grande escala.

Em vista disso, neste capítulo, são apresentados conceitos e notações básicas acerca da subdivisão de matrizes. Além disso, a definição do complemento de Schur e outras notações básicas são apresentadas, uma vez que serão, frequentemente, usadas na especificação do algoritmo BRI, no Capítulo 4. A hierarquia de memória dos sistemas computacionais também é apreciada conforme o contexto em questão.

3.1 Matriz em Blocos

Segundo [Anton & Busby 2006], uma matriz pode ser dividida em submatrizes ou blocos de várias formas inserindo retas tracejadas entre linhas e colunas selecionadas. Exemplificando, considere que uma matriz $T(3 \times 4)$ qualquer seja subdividida em quatro blocos T_{11} , T_{12} , T_{21} e T_{22} , a saber:

$$T = \left[\begin{array}{ccc|c} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ \hline t_{31} & t_{32} & t_{33} & t_{34} \end{array} \right] = \left[\begin{array}{cc} T_{11} & T_{12} \\ T_{21} & T_{22} \end{array} \right] \quad (3.1)$$

onde

$$\begin{aligned} T_{11} &= \left[\begin{array}{ccc} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \end{array} \right], & T_{12} &= \left[\begin{array}{c} t_{14} \\ t_{24} \end{array} \right], \\ T_{21} &= \left[\begin{array}{ccc} t_{31} & t_{32} & t_{33} \end{array} \right] \text{ e } & T_{22} &= \left[\begin{array}{c} t_{34} \end{array} \right]. \end{aligned} \quad (3.2)$$

Neste caso, considera-se T como sendo matriz em blocos 2×2 cujas entradas são, por sua vez, matrizes.

As matrizes em blocos suportam uma fácil obtenção de fatorações matriciais, permitindo a detecção de padrões em uma computação que possa estar obscurecida no nível escalar. Algoritmos formulados no nível de bloco são tipicamente ricos em multiplicação entre matrizes, que é uma operação desejável em muitos ambientes computacionais de alto desempenho. Além disso, na maioria das vezes, a estrutura de blocos de uma matriz é recursiva, o que significa que as entradas em blocos têm uma semelhança explorável com a matriz geral. Este tipo de conexão é a base para algoritmos de produtos matriz-vetor, ditos mais rápidos, tais como transformadas rápidas de Fourier, transformadas trigonométricas e transformadas *Wavelet* [Golub & Van Loan 2013].

Considerando os blocos como entradas numéricas, é possível aplicar operações matemáticas diretamente entre os blocos das matrizes tratadas. Por exemplo, sejam as matrizes em blocos:

$$T = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \quad \text{e} \quad U = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ U_{21} & U_{22} & U_{23} \end{bmatrix}. \quad (3.3)$$

Se os tamanhos dos blocos das matrizes T e U satisfizerem as restrições necessárias para as operações efetuadas, então a operação multiplicação em blocos será efetuada como apresentada a seguir [Anton & Busby 2006]:

$$\begin{aligned} TU &= \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ U_{21} & U_{22} & U_{23} \end{bmatrix} \\ &= \begin{bmatrix} T_{11}U_{11} + T_{12}U_{21} & T_{11}U_{12} + T_{12}U_{22} & T_{11}U_{13} + T_{12}U_{23} \\ T_{21}U_{11} + T_{22}U_{21} & T_{21}U_{12} + T_{22}U_{22} & T_{21}U_{13} + T_{22}U_{23} \end{bmatrix}. \end{aligned} \quad (3.4)$$

3.1.1 Terminologia

Seja $M \in \mathbb{R}^{m \times m}$ uma matriz não-singular composta por blocos quadrados $M_{\alpha\beta}$ de mesma ordem

$$M = \begin{bmatrix} M_{11} & \cdots & M_{1k} \\ \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{kk} \end{bmatrix}, \quad (3.5)$$

onde $M_{\alpha\beta}$ representa o bloco (α, β) . A partir desta notação, o bloco $M_{\alpha\beta}$ tem dimensão $b \times b$, com $b = \frac{m}{k}$, e $M = (M_{\alpha\beta})$ é uma matriz em blocos $k \times k$.

Assumindo que M^{-1} é a matriz inversa de M , então,

$$M^{-1} = \begin{bmatrix} N_{11} & \cdots & N_{1k} \\ \vdots & \ddots & \vdots \\ N_{k1} & \cdots & N_{kk} \end{bmatrix}. \quad (3.6)$$

Atribuindo $k = 2$, obtém-se uma matriz em blocos 2×2 como segue:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad (3.7)$$

onde $A := M_{11}$, $B := M_{12}$, $C := M_{21}$ e $D := M_{22}$. Sendo M^{-1} a matriz inversa de M , então,

$$M^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}, \quad (3.8)$$

onde $E := N_{11}$, $F := N_{12}$, $G := N_{21}$ e $H := N_{22}$.

3.2 Complemento de Schur

Embora as discussões sobre as manifestações implícitas de matrizes de complemento de Schur reportarem aos anos de 1800, apenas em 1968, o termo complemento de Schur e sua notação foram introduzidos pela matemática Emilie Virginia Haynsworth (1916-1985) em [Haynsworth 1968b] e [Haynsworth 1968a]. Esse termo foi atribuído em homenagem ao famoso matemático Issai Schur (1875-1941), que, no ano de 1917, havia publicado o lema que estabeleceu a fórmula do determinante de Schur [Schur 1917]:

$$\det(L) = \det(P) \cdot \det(S - RP^{-1}Q), \quad (3.9)$$

onde P é uma submatriz não-singular da matriz particionada L , de ordem $2n \times 2n$, assim definida:

$$L = \begin{bmatrix} P & Q \\ R & S \end{bmatrix}, \quad (3.10)$$

sendo P, Q, R, S quatro submatrizes de ordem $n \times n$.

Desde então, o complemento de Schur tem desempenhado um importante papel em Estatística, Análise Numérica, Álgebra Linear e em diversas outras áreas da Matemática e suas aplicações [Zhang 2005].

3.2.1 Definição

Dada uma matriz particionada em blocos, 2×2 , o complemento de Schur de uma submatriz desta matriz corresponde a uma relação dela com as demais submatrizes, definida a seguir.

Suponha que M seja uma matriz em blocos 2×2 , quadrada, como definida em (3.7). Sendo A uma submatriz principal não-singular de M , o complemento de Schur de A em M [Zhang 2005], denotado por (M/A) , é definido como sendo a matriz:

$$(M/A) = D - CA^{-1}B. \quad (3.11)$$

De um modo mais geral, conforme [Haynsworth 1968a], o complemento de Schur de qualquer submatriz principal não-singular S de M é definido como sendo a matriz obtida, primeiro, permutando simultaneamente as linhas e as colunas de M , de modo a posicionar S no canto superior esquerdo de M e, então, calculando a matriz (M/A) de (3.11).

Desta forma, sendo B não-singular, ao posicioná-la no canto superior esquerdo em M , temos:

$$M' = \begin{bmatrix} B & A \\ D & C \end{bmatrix}. \quad (3.12)$$

Então, aplicando (3.11), obtemos a fórmula do complemento de Schur de B em M :

$$(M/B) = C - DB^{-1}A. \quad (3.13)$$

E, finalmente, é possível definir de forma semelhante os seguintes complementos de Schur:

$$(M/C) = B - AC^{-1}D, \quad (3.14)$$

como sendo o complemento de Schur de C em M . E,

$$(M/D) = A - BD^{-1}C, \quad (3.15)$$

o complemento de Schur de D em M ; desde que, C , e D sejam não-singulares em (3.14), e (3.15), respectivamente [Brezinski 1988].

3.2.2 Fórmula da inversão de matrizes em blocos 2×2

Sejam a matriz M e o bloco superior da diagonal principal, A em (3.7), não-singulares. Então, a fórmula da diagonalização de Aitken consiste em [Zhang 2005]:

$$\begin{aligned} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} I & -A^{-1}B \\ 0 & I \end{bmatrix} &= \begin{bmatrix} A & 0 \\ 0 & M/A \end{bmatrix}, \\ \begin{bmatrix} A & B \\ C & D \end{bmatrix} &= \begin{bmatrix} I & 0 \\ CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & (M/A) \end{bmatrix} \begin{bmatrix} I & A^{-1}B \\ 0 & I \end{bmatrix}, \end{aligned} \quad (3.16)$$

onde I é a matriz identidade e 0 é a matriz nula.

Da fórmula (3.9), segue que a matriz quadrada M é não-singular se, e somente se, o complemento de Schur $(M/A) = D - CA^{-1}B$ for não-singular. Então, obtém-se a fórmula da inversão de Banachiewicz [Banachiewicz 1937] para a inversa de uma matriz particionada:

$$\begin{aligned} M^{-1} &= \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} I & -A^{-1}B \\ 0 & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & (M/A)^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} A^{-1} & -A^{-1}B(M/A)^{-1} \\ 0 & (M/A)^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} A^{-1} + A^{-1}B(M/A)^{-1}CA^{-1} & -A^{-1}B(M/A)^{-1} \\ -(M/A)^{-1}CA^{-1} & (M/A)^{-1} \end{bmatrix}. \end{aligned} \quad (3.17)$$

Suponha, a seguir, que o bloco inferior da diagonal principal, D em (3.7), seja não-singular. Então, a matriz M é não-singular se, e somente se, o complemento de Schur $(M/D) = A - BD^{-1}C$ for não-singular, como definido em (3.9). Seguindo a mesma forma apresentada anteriormente, M pode ser assim decomposta:

$$M = \begin{bmatrix} I & BD^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} (M/D) & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ D^{-1}C & I \end{bmatrix}. \quad (3.18)$$

E a inversa de M pode ser escrita como:

$$\begin{aligned} M^{-1} &= \begin{bmatrix} I & 0 \\ -D^{-1}C & I \end{bmatrix} \begin{bmatrix} (M/D)^{-1} & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & -BD^{-1} \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} (M/D)^{-1} & -(M/D)^{-1}BD^{-1} \\ -D^{-1}C(M/D)^{-1} & D^{-1} + D^{-1}C(M/D)^{-1}BD^{-1} \end{bmatrix}. \end{aligned} \quad (3.19)$$

A formulação em (3.19) é bem conhecida e tem sido amplamente utilizada no tratamento de inversas de matrizes em blocos [Noble & Daniel 1988].

3.3 Memória

A memória do computador é definida como sendo o espaço de trabalho do sistema operacional, no qual são mantidos os processos, *threads*¹, bibliotecas compartilhadas e canais de comunicação, além do próprio *kernel* do sistema, com seu código e suas estruturas de dados [Maziero 2014]. Ela é, comumente, organizada em uma hierarquia.

Observando um sistema computacional típico, pode-se identificar os vários locais onde os dados são armazenados, conforme apresentados na Figura 3.1. No nível mais alto, estão os registradores e o *cache* interno do processador (denominado *cache* L1). Em seguida, vem o *cache* externo da placa mãe (*cache* L2) e, posteriormente, a memória principal (RAM). Além disso, discos rígidos e unidades de armazenamento externas (por exemplo, *pendrives*, CD-ROMs, DVD-ROMs e fitas magnéticas) são considerados memórias externas, pois também possuem a função de armazenamento de dados [Stallings 2010].

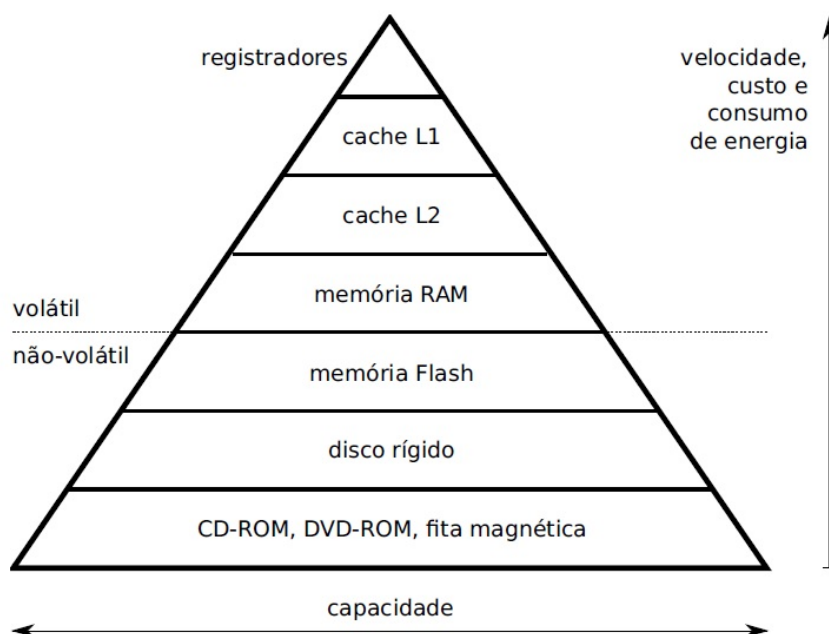


Figura 3.1: Hierarquia de memória de um sistema computacional. Fonte: [Maziero 2014]

¹*Threads* são fluxos de execução de um mesmo processo que compartilham código e dados entre si. É uma forma de o processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente [Tanenbaum 2003].

Esses componentes de hardware são construídos usando diversas tecnologias e por isso têm características distintas, como a capacidade de armazenamento, a velocidade de operação, o consumo de energia e o custo por *byte* armazenado. Pela Figura 3.1 é possível perceber que, quanto mais próxima ao processador, a memória se torna mais rápida, mais cara e, conseqüentemente, possui tamanho menor.

O projeto geral da hierarquia de memória varia de sistema para sistema. No entanto, duas considerações sempre se aplicam:

- Cada nível na hierarquia tem uma capacidade limitada e por razões econômicas esta capacidade, geralmente, se torna menor à medida que subimos a hierarquia.
- Existe um custo, por vezes relativamente alto, associado à movimentação de dados entre dois níveis na hierarquia.

Para acessar dados (ou instruções) contidos na memória, a CPU (Unidade Central de Processamento) segue um nível de prioridade. Quando a CPU deseja aceder a um dado que acredita estar no local de armazenamento, primeiramente ele verifica a memória *cache*. Caso o dado não seja encontrado, busca-se, então, na memória RAM e, posteriormente, no disco rígido. Tão logo o dado seja encontrado em algum nível superior de memória, a CPU pára a verificação e níveis inferiores da hierarquia não são consultados. Assim, quando a *cache* é consultada pela CPU e o dado está disponível, isso é classificado como *cache hit* (acerto do *cache*). Se o mesmo não estiver acessível na *cache*, então, classifica-se como *cache miss* (erro do *cache*).

A localização dos dados na memória afeta diretamente o tempo de execução do programa, pois acessar dados na memória *cache* resultará em um tempo de processamento muito menor do que acessar dados na memória RAM ou, até mesmo, na memória de disco rígido.

Um problema constante nos sistemas computacionais diz respeito à disponibilidade de memória física. Os programas têm se tornado cada vez maiores e mais processos tendem a serem executados simultaneamente, ocupando toda a memória disponível. Além disso, a crescente manipulação de grandes massas de dados contribui mais ainda para esse problema, uma vez que seu tratamento exige grandes quantidades de memória livre. Considerando que a memória RAM é um recurso caro e que consome uma quantidade significativa de energia, aumentar sua capacidade nem sempre é uma opção factível [Maziero 2014].

Desta forma, uma implementação eficiente de um algoritmo requer também uma capacidade de raciocinar sobre o encaixe dos dados entre os vários níveis de armazenamento. Considerando uma computação matricial, a manipulação dos dados estruturados em blo-

cos permite que eles possam ser armazenados sem exceder a capacidade de memória disponível para a sua execução.

Capítulo 4

Método de Inversão Recursiva de Matrizes em Blocos

O algoritmo de Inversão Recursiva de Matrizes em Blocos (BRI) é um método que pode ser aplicado para inverter qualquer matriz $M \in \mathbb{R}^{m \times m}$, desde que M seja não-singular e que todas as submatrizes que necessitam ser invertidas no procedimento também sejam não-singulares. O BRI retorna um bloco da matriz inversa ao aplicar, recursivamente, operações pré-definidas e complementos de Schur nas submatrizes resultantes da matriz de entrada.

Neste capítulo, o algoritmo BRI é definido, bem como é apresentado o processo de inversão de uma matriz em blocos 4×4 , a fim de exemplificar sua operação. Além disso, é realizada uma análise de complexidade do algoritmo com relação ao uso de memória e tempo de execução. E, por fim, são discutidos os resultados experimentais.

4.1 Algoritmo de Inversão Recursiva por Blocos

Considere a matriz M ,

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix},$$

e a matriz M^{-1} ,

$$M^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

definidas em (3.7) e em (3.8), respectivamente.

É importante notar que, de acordo com a Fórmula (3.19), apresentada na Subseção

3.2.2, a saber:

$$M^{-1} = \begin{bmatrix} (M/D)^{-1} & -(M/D)^{-1}BD^{-1} \\ -D^{-1}C(M/D)^{-1} & D^{-1} + D^{-1}C(M/D)^{-1}BD^{-1} \end{bmatrix},$$

para obter E basta calcular a inversa do complemento de Schur de D em M .

Seguindo esta premissa, observou-se que, para calcular o bloco N_{11} da inversa da matriz em blocos $k \times k$, M , com $k > 2$, deve-se dividir sucessivamente M em 4 submatrizes de mesma ordem, denominados quadros, até que os quadros resultantes sejam matrizes em blocos 2×2 . Em seguida, para cada quadro 2×2 resultante, o complemento de Schur deve ser aplicado na direção oposta da recursão, reduzindo o quadro para um único bloco, que deve ser combinado com os outros 3 blocos do ramo de recursão para formar quadros 2×2 que devem ser reduzidos a blocos únicos, sucessivamente. Finalmente, N_{11} é obtido calculando o inverso do último bloco resultante do procedimento de recursão reversa. Para isso, todas as submatrizes que necessitam ser invertidas no algoritmo proposto também devem ser não-singulares.

Para uma melhor compreensão, o algoritmo será explicado considerando duas etapas, nomeadamente *Procedimento Recursivo Progressivo* e *Procedimento Recursivo Retroativo*. Para tanto, considere (3.5) como sendo a matriz de entrada em blocos, a saber:

$$M = \begin{bmatrix} M_{11} & \cdots & M_{1k} \\ \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{kk} \end{bmatrix}.$$

4.1.1 Procedimento Recursivo Progressivo

Passo 1: Divida a entrada M , uma matriz em blocos $k \times k$, em quatro matrizes em blocos $(k-1) \times (k-1)$, denominados quadros e simbolizados por $M\rangle_A$, $M\rangle_B$, $M\rangle_C$ e $M\rangle_D$, excluindo uma das linhas de blocos de M e uma das colunas de blocos de M , tal como segue.

O quadro $M\rangle_A$ resulta da remoção da linha de blocos mais inferior e da coluna de blocos mais à direita de M ; $M\rangle_B$ resulta da remoção da linha de blocos mais inferior e da coluna de blocos mais à esquerda de M ; $M\rangle_C$ resulta da remoção da linha de blocos mais superior e da coluna de blocos mais à direita de M e, finalmente, $M\rangle_D$ resulta da remoção da linha de blocos mais superior e da coluna de blocos mais à esquerda de M , como apresentado em (4.1).

$$\begin{aligned}
M\rangle_A &= \begin{bmatrix} M_{11} & \cdots & M_{1(k-1)} \\ \vdots & \ddots & \vdots \\ M_{(k-1)1} & \cdots & M_{(k-1)(k-1)} \end{bmatrix}, \\
M\rangle_B &= \begin{bmatrix} M_{12} & \cdots & M_{1k} \\ \vdots & \ddots & \vdots \\ M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix}, \\
M\rangle_C &= \begin{bmatrix} M_{21} & \cdots & M_{2(k-1)} \\ \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{k(k-1)} \end{bmatrix}, \\
M\rangle_D &= \begin{bmatrix} M_{22} & \cdots & M_{2k} \\ \vdots & \ddots & \vdots \\ M_{k2} & \cdots & M_{kk} \end{bmatrix}.
\end{aligned} \tag{4.1}$$

Passo 2: Permute linhas de blocos e/ou colunas de blocos até que o bloco M_{22} atinja sua posição original, a saber, segunda linha de blocos e segunda coluna de blocos. Neste caso, o quadro $M\rangle_A$ não sofre alteração. Em $M\rangle_B$, é necessário permutar as duas colunas de bloco mais à esquerda. Em $M\rangle_C$, é necessário permutar as duas linhas de bloco mais superiores. E, finalmente, em $M\rangle_D$, basta permutar as duas colunas de bloco mais à esquerda e as duas linhas de bloco mais superiores. Então, aplicando essas permutações,

resulta em

$$\begin{aligned}
 M\backslash_A &= \begin{bmatrix} M_{11} & M_{12} & \cdots & M_{1(k-1)} \\ M_{21} & M_{22} & \cdots & M_{2(k-1)} \\ M_{31} & M_{32} & \cdots & M_{3(k-1)} \\ \vdots & \ddots & \vdots & \\ M_{(k-1)1} & M_{(k-1)2} & \cdots & M_{(k-1)(k-1)} \end{bmatrix}, \\
 M\backslash_B &= \begin{bmatrix} M_{13} & M_{12} & \cdots & M_{1k} \\ M_{23} & M_{22} & \cdots & M_{2k} \\ M_{33} & M_{32} & \cdots & M_{3k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(k-1)3} & M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix}, \\
 M\backslash_C &= \begin{bmatrix} M_{31} & M_{32} & \cdots & M_{3(k-1)} \\ M_{21} & M_{22} & \cdots & M_{2(k-1)} \\ M_{41} & M_{42} & \cdots & M_{4(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k1} & M_{k2} & \cdots & M_{k(k-1)} \end{bmatrix} \text{ e} \\
 M\backslash_D &= \begin{bmatrix} M_{33} & M_{32} & \cdots & M_{3k} \\ M_{23} & M_{22} & \cdots & M_{2k} \\ M_{43} & M_{42} & \cdots & M_{4k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k3} & M_{k2} & \cdots & M_{kk} \end{bmatrix}.
 \end{aligned} \tag{4.2}$$

Em seguida, divida cada um dos quadros em (4.2), $M\backslash_A$, $M\backslash_B$, $M\backslash_C$ e $M\backslash_D$ em quatro quadros $(k-2) \times (k-2)$, excluindo uma de suas linhas de blocos e uma de suas colunas de blocos, conforme instruído no Passo 01. Por exemplo, dividir o quadro $M\backslash_D$ resulta

em

$$\begin{aligned}
 M_{\rangle D \rangle A} &= \begin{bmatrix} M_{33} & M_{32} & \cdots & M_{3(k-1)} \\ M_{23} & M_{22} & \cdots & M_{2(k-1)} \\ M_{43} & M_{42} & \cdots & M_{4(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(k-1)3} & M_{(k-1)2} & \cdots & M_{(k-1)(k-1)} \end{bmatrix}, \\
 M_{\rangle D \rangle B} &= \begin{bmatrix} M_{32} & M_{34} & \cdots & M_{3k} \\ M_{22} & M_{24} & \cdots & M_{2k} \\ M_{42} & M_{44} & \cdots & M_{4k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(k-1)4} & M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix}, \\
 M_{\rangle D \rangle C} &= \begin{bmatrix} M_{23} & M_{22} & \cdots & M_{2(k-1)} \\ M_{43} & M_{42} & \cdots & M_{4(k-1)} \\ M_{53} & M_{52} & \cdots & M_{5(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k3} & M_{k2} & \cdots & M_{k(k-1)} \end{bmatrix} \mathbf{e} \\
 M_{\rangle D \rangle D} &= \begin{bmatrix} M_{22} & M_{24} & \cdots & M_{2k} \\ M_{42} & M_{44} & \cdots & M_{4k} \\ M_{52} & M_{54} & \cdots & M_{5k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k2} & M_{k4} & \cdots & M_{kk} \end{bmatrix}.
 \end{aligned} \tag{4.3}$$

Passo i : Para cada um dos quadros $(k - (i - 1)) \times (k - (i - 1))$ resultantes do passo anterior (Passo $i - 1$), $M_{\rangle A}^{i-1}$, $M_{\rangle B}^{i-1}$, $M_{\rangle C}^{i-1}$ e $M_{\rangle D}^{i-1}$, faça a permutação de linhas de blocos e/ou colunas de blocos e, em seguida, gere mais quatro quadros $(k - i) \times (k - i)$, $M_{\rangle A}^i$, $M_{\rangle B}^i$, $M_{\rangle C}^i$ e $M_{\rangle D}^i$, excluindo uma de suas linhas de blocos e uma de suas colunas de blocos de forma análoga ao que foi feito no Passo 2 para os quadros resultantes do Passo 1. Repita o Passo i até $i = k - 2$.

O número sobrescrito ao símbolo " \rangle " indica a quantidade desses símbolos existentes, incluindo aqueles que não estão representados. Assim, denota $M_{\rangle x}^y$ com $y \in \mathbb{N}^*$ e $x \in \{A, B, C, D\}$.

4.1.2 Procedimento Recursivo Retroativo

Passo 1: Para cada um dos quadros (matrizes em blocos 2×2) resultantes do Passo $k - 2$ do Procedimento Recursivo Progressivo, apresentado na Subseção 4.1.1, M_A^{k-2} , M_B^{k-2} , M_C^{k-2} e M_D^{k-2} , aplique o complemento de Schur de M_{22} para gerar os blocos: $\langle M_A^{k-2} \rangle$, $\langle M_B^{k-2} \rangle$, $\langle M_C^{k-2} \rangle$ e $\langle M_D^{k-2} \rangle$, usando as Equações (3.15), (3.14), (3.13) e (3.11), nesta ordem. O símbolo " \langle " indica que o complemento de Schur foi calculado no respectivo quadro.

Passo 2: Obtenha $\langle M_A^{k-3} \rangle$, $\langle M_B^{k-3} \rangle$, $\langle M_C^{k-3} \rangle$ e $\langle M_D^{k-3} \rangle$, combinando os quatro blocos que foram originados a partir do mesmo ramo da recursão, no Procedimento Recursivo Progressivo, relatado na Subseção 4.1.1, da seguinte maneira:

$$\langle M_A^{k-3} \rangle = \begin{bmatrix} \langle M_A^{k-3} \rangle_A & \langle M_A^{k-3} \rangle_B \\ \langle M_A^{k-3} \rangle_C & \langle M_A^{k-3} \rangle_D \end{bmatrix}, \quad (4.4)$$

$$\langle M_B^{k-3} \rangle = \begin{bmatrix} \langle M_B^{k-3} \rangle_A & \langle M_B^{k-3} \rangle_B \\ \langle M_B^{k-3} \rangle_C & \langle M_B^{k-3} \rangle_D \end{bmatrix}, \quad (4.5)$$

$$\langle M_C^{k-3} \rangle = \begin{bmatrix} \langle M_C^{k-3} \rangle_A & \langle M_C^{k-3} \rangle_B \\ \langle M_C^{k-3} \rangle_C & \langle M_C^{k-3} \rangle_D \end{bmatrix}, \quad (4.6)$$

$$\langle M_D^{k-3} \rangle = \begin{bmatrix} \langle M_D^{k-3} \rangle_A & \langle M_D^{k-3} \rangle_B \\ \langle M_D^{k-3} \rangle_C & \langle M_D^{k-3} \rangle_D \end{bmatrix}. \quad (4.7)$$

Em seguida, para cada uma dessas matrizes em blocos 2×2 , calcule o complemento de Schur aplicando (3.15), (3.14), (3.13) e (3.11), nesta ordem, para gerar: $\langle \langle M_A^{k-3} \rangle \rangle$, $\langle \langle M_B^{k-3} \rangle \rangle$, $\langle \langle M_C^{k-3} \rangle \rangle$ e $\langle \langle M_D^{k-3} \rangle \rangle$.

Passo i : Considerando cada quatro ramos da recursão, repita o passo anterior gerando quadros $\langle {}^i M_A^{k-(i+1)} \rangle_{i-1}$, $\langle {}^i M_B^{k-(i+1)} \rangle_{i-1}$, $\langle {}^i M_C^{k-(i+1)} \rangle_{i-1}$ e $\langle {}^i M_D^{k-(i+1)} \rangle_{i-1}$, até $i = k - 2$ e, assim, obter $\langle {}^{k-2} M_A \rangle^{k-3}$, $\langle {}^{k-2} M_B \rangle^{k-3}$, $\langle {}^{k-2} M_C \rangle^{k-3}$ e $\langle {}^{k-2} M_D \rangle^{k-3}$.

Passo $k - 1$: Monte $\langle {}^{k-2} M \rangle^{k-2}$ a partir de $\langle {}^{k-2} M_A \rangle^{k-3}$, $\langle {}^{k-2} M_B \rangle^{k-3}$, $\langle {}^{k-2} M_C \rangle^{k-3}$ e $\langle {}^{k-2} M_D \rangle^{k-3}$, como segue:

$$\langle {}^{k-2} M \rangle^{k-2} = \begin{bmatrix} \langle {}^{k-2} M_A \rangle^{k-3} & \langle {}^{k-2} M_B \rangle^{k-3} \\ \langle {}^{k-2} M_C \rangle^{k-3} & \langle {}^{k-2} M_D \rangle^{k-3} \end{bmatrix}. \quad (4.8)$$

E, finalmente, calcule o complemento de Schur de $\langle^{k-2}M\rangle^{k-2}$ em relação a $\langle^{k-2}M\rangle_D^{k-3}$ usando (3.15). Assim, o inverso da matriz correspondente a N_{11} condiz com:

$$N_{11} = (\langle^{k-2}M\rangle^{k-2} / \langle^{k-2}M\rangle_D^{k-3})^{-1}. \quad (4.9)$$

O Algoritmo 1 apresenta o pseudocódigo resumido do módulo principal do algoritmo BRI com o retorno do bloco superior esquerdo da matriz inversa de M , a saber, o bloco N_{11} . A recursividade do algoritmo BRI é gerada por quatro funções, nomeadamente, BRI_A , BRI_B , BRI_C e BRI_D , descritas pelos Algoritmos 2, 3, 4 e 5, respectivamente. Durante a execução da recursividade retroativa, essas funções retornam os complementos de Schur (M/D) , (M/C) , (M/B) e (M/A) , respectivamente.

É válido observar que as funções recursivas BRI_A , BRI_B , BRI_C e BRI_D recebem, como parâmetro de entrada, a quantidade de blocos por linha (ou colunas) da matriz (ou quadro) a ser dividida no Procedimento Recursivo Progressivo. Além disso, elas recebem também dois vetores, denominados linha e coluna, que armazenam os índices das linhas de blocos e os índices das colunas de blocos da matriz M .

Durante a geração dos quadros, na recursividade progressiva, as operações de exclusão e/ou permutação de linhas e/ou colunas de M serão simulados ao se manipular os dados desses vetores. Desta maneira, não é necessário armazenar a matriz de entrada, M , inteira na memória principal nem particioná-la efetivamente. Do contrário, seria computacionalmente custoso e demandaria um consumo maior de memória. A matriz original poderá, então, ser acessada apenas quando essas funções forem calcular os respectivos complementos de Schur.

A passagem dos vetores linha e coluna para as funções BRI_A , BRI_B , BRI_C e BRI_D pode ser por valor ou por referência, sendo preferível por referência a fim de economizar memória.

Conforme o Algoritmo 1, esses vetores são inicializados com valores de 1 a k . A cada descida na recursividade, para a obtenção dos quatro quadros x_A , x_B , x_C e x_D , é extraído um elemento tanto do vetor linha quanto do vetor coluna, conforme a remoção de uma linha de blocos e uma coluna de blocos, descrita no Passo 1 do Procedimento Recursivo Progressivo. Este processo está ilustrado na Figura 4.1.

Vale ressaltar que, com uma permutação adequada de linhas e colunas, pode-se posicionar qualquer bloco no canto superior esquerdo de M e obter o inverso correspondente a este bloco em relação a M .

Uma vez que a dimensão m e o número de blocos k de M são arbitrários, para os casos em que a ordem b de cada bloco seja $b = \frac{m}{k}$, com $b \notin \mathbb{N}$, é possível considerar uma matriz

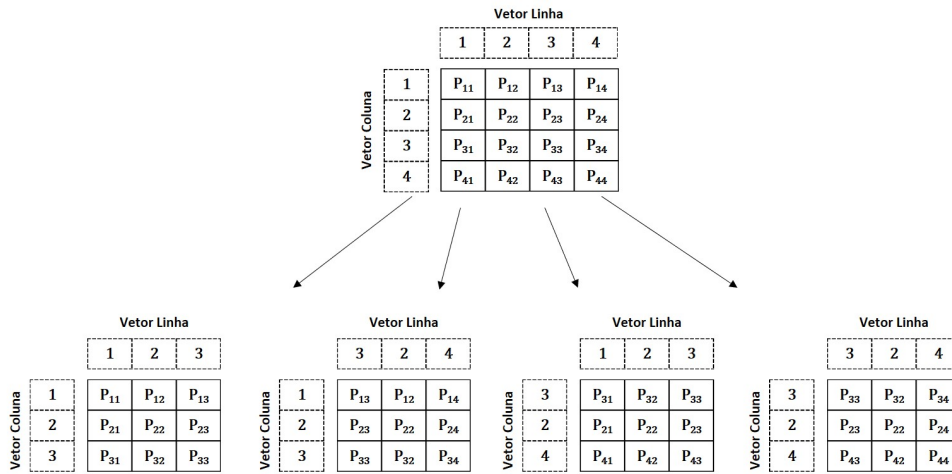


Figura 4.1: Vetor linha e vetor coluna.

Algorithm 1 Algoritmo de Inversão Recursiva de Blocos

Entrada: k (quantidade de blocos por linha), b (ordem do bloco), $M = (M_{\alpha\beta})_{1 \leq \alpha, \beta \leq k}$ (matriz de entrada)

1. $vet : linha[1..k]$ de inteiro
2. $vet : coluna[1..k]$ de inteiro
3. $N_{11} = inv(BRI_A(b, linha, coluna))$

Saída: N_{11}

Algorithm 2 Função recursiva BRI_A

Entrada: k (quantidade de blocos por linha), $coluna$ (vetor coluna de blocos), $linha$ (vetor linha de blocos)

1. **function** $BRI_A(k, linha, coluna)$
2. **if** $k > 2$ **then**
3. $M_A \leftarrow BRI_A(k-1, linha[1:k-1], coluna[1:k-1])$
4. $M_B \leftarrow BRI_B(k-1, linha[1:k-1], coluna[2:k])$
5. $M_C \leftarrow BRI_C(k-1, linha[2:k], coluna[1:k-1])$
6. $M_D \leftarrow BRI_D(k-1, linha[2:k], coluna[2:k])$
7. $(M/D) \leftarrow M_A - M_B * inv(M_D) * M_C$
8. **else**
9. $(M/D) \leftarrow A - B * inv(D) * C$
10. **end if**
11. **end function**

Saída: (M/D)

Algorithm 3 Função recursiva BRI_B

Entrada: k (quantidade de blocos por linha), $coluna$ (vetor coluna de blocos), $linha$ (vetor linha de blocos)

```

1. function  $BRI_B(k, linha, coluna)$ 
2.   if  $k > 2$  then
3.      $temp \leftarrow coluna[2]$ 
4.      $coluna[2] \leftarrow coluna[1]$ 
5.      $coluna[1] \leftarrow temp$ 
6.      $M_A \leftarrow BRI_A(k-1, linha[1:k-1], coluna[1:k-1])$ 
7.      $M_B \leftarrow BRI_B(k-1, linha[1:k-1], coluna[2:k])$ 
8.      $M_C \leftarrow BRI_C(k-1, linha[2:k], coluna[1:k-1])$ 
9.      $M_D \leftarrow BRI_D(k-1, linha[2:k], coluna[2:k])$ 
10.     $(M/C) \leftarrow M_B - M_A * inv(M_C) * M_D$ 
11.  else
12.     $(M/C) \leftarrow B - A * inv(C) * D$ 
13.  end if
14. end function

```

Saída: (M/C)

Algorithm 4 Função recursiva BRI_C

Entrada: k (quantidade de blocos por linha), $coluna$ (vetor coluna de blocos), $linha$ (vetor linha de blocos)

```

1. function  $BRI_C(k, linha, coluna)$ 
2.   if  $k > 2$  then
3.      $temp \leftarrow linha[2]$ 
4.      $linha[2] \leftarrow linha[1]$ 
5.      $linha[1] \leftarrow temp$ 
6.      $M_A \leftarrow BRI_A(k-1, linha[1:k-1], coluna[1:k-1])$ 
7.      $M_B \leftarrow BRI_B(k-1, linha[1:k-1], coluna[2:k])$ 
8.      $M_C \leftarrow BRI_C(k-1, linha[2:k], coluna[1:k-1])$ 
9.      $M_D \leftarrow BRI_D(k-1, linha[2:k], coluna[2:k])$ 
10.     $(M/B) \leftarrow M_C - M_D * inv(M_B) * M_A$ 
11.  else
12.     $(M/B) \leftarrow C - D * inv(B) * A$ 
13.  end if
14. end function

```

Saída: (M/B)

Algorithm 5 Função recursiva BRI_D

Entrada: k (quantidade de blocos por linha), $coluna$ (vetor coluna de blocos), $linha$ (vetor linha de blocos)

```

1. function  $BRI_D(k, linha, coluna)$ 
2.   if  $k > 2$  then
3.      $temp \leftarrow linha[2]$ 
4.      $linha[2] \leftarrow linha[1]$ 
5.      $linha[1] \leftarrow temp$ 
6.      $temp \leftarrow coluna[2]$ 
7.      $coluna[2] \leftarrow coluna[1]$ 
8.      $coluna[1] \leftarrow temp$ 
9.      $M_A \leftarrow BRI_A(k-1, linha[1:k-1], coluna[1:k-1])$ 
10.     $M_B \leftarrow BRI_B(k-1, linha[1:k-1], coluna[2:k])$ 
11.     $M_C \leftarrow BRI_C(k-1, linha[2:k], coluna[1:k-1])$ 
12.     $M_D \leftarrow BRI_D(k-1, linha[2:k], coluna[2:k])$ 
13.     $(M/A) \leftarrow M_D - M_C * inv(M_A) * M_B$ 
14.  else
15.     $(M/A) \leftarrow D - C * inv(A) * B$ 
16.  end if
17. end function

```

Saída: (M/A)

Γ como uma matriz aumentada de M como segue.

Seja m a ordem de uma matriz quadrada $M \in \mathbb{R}^{m \times m}$, k o número arbitrário de blocos que se quer particionar a matriz de entrada do algoritmo BRI e l o número inteiro mínimo com $\frac{m+l}{k} \in \mathbb{N}$. A matriz aumentada de M é

$$\Gamma = \begin{bmatrix} M & 0 \\ 0^T & I \end{bmatrix}, \quad (4.10)$$

com $\Gamma = M$ para $\frac{m}{k} \in \mathbb{N}$, onde 0 é a matriz nula ($m \times l$) e I a matriz identidade de ordem l . Dessa forma, a matriz Γ pode ser particionada em k blocos e a inversa M^{-1} de M será obtida aplicando o algoritmo BRI à matriz Γ .

Assim, temos que o inverso de Γ é:

$$\Gamma^{-1} = \begin{bmatrix} M^{-1} & 0 \\ 0^T & I \end{bmatrix}. \quad (4.11)$$

4.2 Um Exemplo: Inversa de matrizes em blocos 4×4

Para esclarecer o funcionamento do algoritmo BRI, esta seção apresenta o processo de inversão de uma matriz em blocos 4×4 .

A ideia básica do algoritmo recursivo, apresentado na Seção 4.1, para a inversão de matrizes em blocos $k \times k$, reside no fato de que, em cada Passo i da Subseção 4.1.1, as matrizes envolvidas são divididas em quatro matrizes em blocos $(k-i) \times (k-i)$ até obter matrizes em blocos 2×2 .

Considere que uma matriz M , não-singular, de ordem $4b \times 4b$ possa ser particionada em 4×4 blocos de ordem b , como segue:

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \quad (4.12)$$

e M^{-1} seja sua matriz inversa:

$$M^{-1} = \begin{bmatrix} N_{11} & N_{12} & N_{13} & N_{14} \\ N_{21} & N_{22} & N_{23} & N_{24} \\ N_{31} & N_{32} & N_{33} & N_{34} \\ N_{41} & N_{42} & N_{43} & N_{44} \end{bmatrix}. \quad (4.13)$$

Agora, considere quatro matrizes quadradas de ordem $3b$ geradas a partir de M pelo processo de excluir uma de suas linhas de bloco e uma das suas colunas de bloco, conforme mostrado no Passo 1 da Subseção 4.1.1. Depois de posicionar o bloco M_{22} na sua posição original em M (Passo 2), obtém-se os seguintes quadros:

$$\begin{aligned} M\rangle_A &= \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}, & M\rangle_B &= \begin{bmatrix} M_{13} & M_{12} & M_{14} \\ M_{23} & M_{22} & M_{24} \\ M_{33} & M_{32} & M_{34} \end{bmatrix}, \\ M\rangle_C &= \begin{bmatrix} M_{31} & M_{32} & M_{33} \\ M_{21} & M_{22} & M_{23} \\ M_{41} & M_{42} & M_{43} \end{bmatrix}, & \text{e } M\rangle_D &= \begin{bmatrix} M_{33} & M_{32} & M_{34} \\ M_{23} & M_{22} & M_{24} \\ M_{43} & M_{42} & M_{44} \end{bmatrix}, \end{aligned} \quad (4.14)$$

conforme ilustrado na Figura 4.2.

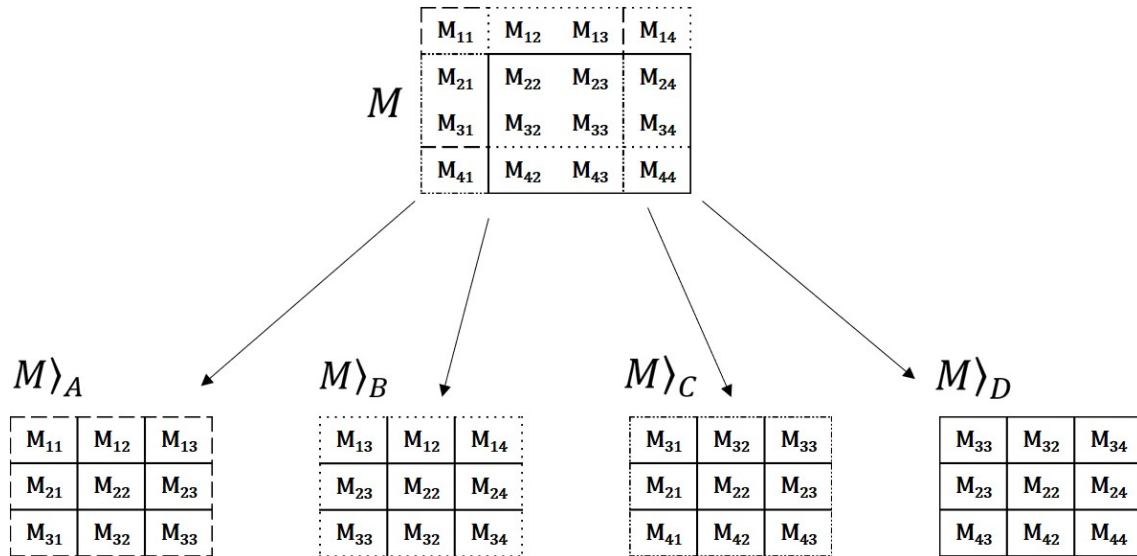


Figura 4.2: Quadros $M\rangle_A$, $M\rangle_B$, $M\rangle_C$ e $M\rangle_D$ resultantes das divisões de uma matriz M .

Posteriormente, de forma recursiva, cada um dos quadros $M\rangle_A$, $M\rangle_B$, $M\rangle_C$ e $M\rangle_D$ serão divididos em quatro quadros 2×2 , aplicando as regras discutidas na Subseção 4.1.1 (Passos 2 e i).

Dividindo, por exemplo, o quadro $M\rangle_B$, cuja ilustração encontra-se na Figura 4.3, os

seguintes quadros são obtidos:

$$\begin{aligned} M_{\rangle B \rangle A} &= \begin{bmatrix} M_{13} & M_{12} \\ M_{23} & M_{22} \end{bmatrix}, & M_{\rangle B \rangle B} &= \begin{bmatrix} M_{12} & M_{14} \\ M_{22} & M_{24} \end{bmatrix}, \\ M_{\rangle B \rangle C} &= \begin{bmatrix} M_{23} & M_{22} \\ M_{33} & M_{32} \end{bmatrix}, & \text{e } M_{\rangle B \rangle D} &= \begin{bmatrix} M_{22} & M_{24} \\ M_{32} & M_{34} \end{bmatrix}. \end{aligned} \quad (4.15)$$

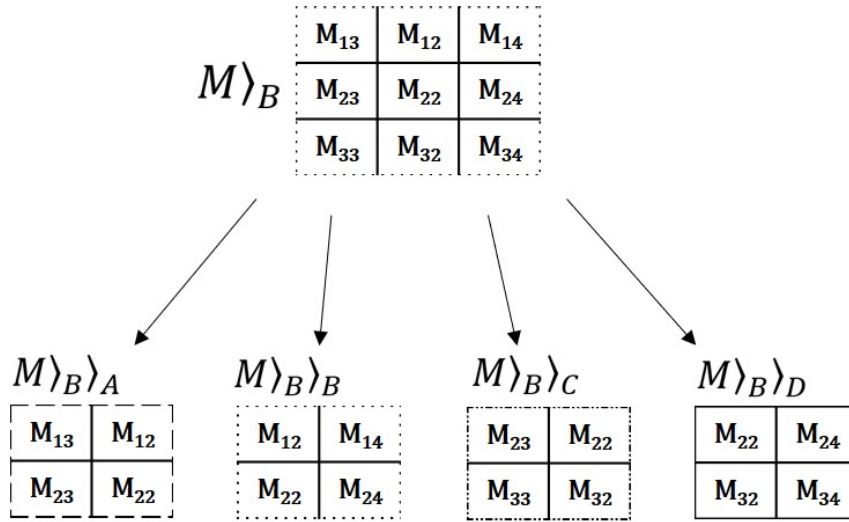


Figura 4.3: Divisões ocorridas no quadro $M_{\rangle B}$, no Procedimento Recursivo Progressivo.

O Procedimento Recursivo Progressivo completo, aplicado na matriz M , encontra-se ilustrado na Figura 4.5. É oportuno observar que os dados manipulados pelo algoritmo BRI pode ser convenientemente estruturados em árvore¹, onde cada quadro resultante se constitui como sendo um nó raiz de subárvores (ou ramos) geradas a partir dele próprio.

Em seguida, conforme o Passo 1 do Procedimento Recursivo Retroativo, na Subseção 4.1.2, assumindo M_{22} como sendo não-singular e aplicando as Equações (3.15), (3.14), (3.13) e (3.11), nesta ordem, o complemento de Schur de M_{22} para cada um dos quadros em (4.15) são calculados:

$$\langle M \rangle_{B \rangle A} = (\langle M \rangle_{B \rangle A} / M_{22}) = M_{13} - M_{12} M_{22}^{-1} M_{23}; \quad (4.16)$$

$$\langle M \rangle_{B \rangle B} = (\langle M \rangle_{B \rangle B} / M_{22}) = M_{14} - M_{12} M_{22}^{-1} M_{24}; \quad (4.17)$$

$$\langle M \rangle_{B \rangle C} = (\langle M \rangle_{B \rangle C} / M_{22}) = M_{33} - M_{32} M_{22}^{-1} M_{23}; \quad (4.18)$$

¹Árvores são estruturas de dados em que a relação entre os dados (denominados nós) é de hierarquia ou composição. As subárvores dessa estrutura são também denominadas de ramos. [Veloso et al. 1986]

$$\langle M \rangle_B \rangle_D = (\langle M \rangle_B \rangle_D / M_{22}) = M_{34} - M_{32} M_{22}^{-1} M_{24}. \quad (4.19)$$

Executando o Passo 2 do mesmo procedimento retroativo, Subseção 4.1.2, produz-se:

$$\langle M \rangle_A \rangle = \begin{bmatrix} \langle M \rangle_A \rangle_A & \langle M \rangle_A \rangle_B \\ \langle M \rangle_A \rangle_C & \langle M \rangle_A \rangle_D \end{bmatrix}, \quad (4.20)$$

$$\langle M \rangle_B \rangle = \begin{bmatrix} \langle M \rangle_B \rangle_A & \langle M \rangle_B \rangle_B \\ \langle M \rangle_B \rangle_C & \langle M \rangle_B \rangle_D \end{bmatrix}, \quad (4.21)$$

$$\langle M \rangle_C \rangle = \begin{bmatrix} \langle M \rangle_C \rangle_A & \langle M \rangle_C \rangle_B \\ \langle M \rangle_C \rangle_C & \langle M \rangle_C \rangle_D \end{bmatrix} \text{ e} \quad (4.22)$$

$$\langle M \rangle_D \rangle = \begin{bmatrix} \langle M \rangle_D \rangle_A & \langle M \rangle_D \rangle_B \\ \langle M \rangle_D \rangle_C & \langle M \rangle_D \rangle_D \end{bmatrix}. \quad (4.23)$$

Assumindo M_{22} , $\langle M \rangle_A \rangle_D$, $\langle M \rangle_B \rangle_C$, $\langle M \rangle_C \rangle_B$ e $\langle M \rangle_D \rangle_A$ como sendo não-singulares e aplicando as Equações (3.15), (3.14), (3.13) e (3.11), nesta ordem, o complemento de Schur calculado para cada um dos quadros resulta em:

$$\begin{aligned} \langle \langle M \rangle_A \rangle &= (\langle M \rangle_A \rangle / \langle M \rangle_A \rangle_D) \\ &= \langle M \rangle_A \rangle_A - \langle M \rangle_A \rangle_B \langle M \rangle_A \rangle_D^{-1} \langle M \rangle_A \rangle_C; \end{aligned} \quad (4.24)$$

$$\begin{aligned} \langle \langle M \rangle_B \rangle &= (\langle M \rangle_B \rangle / \langle M \rangle_B \rangle_C) \\ &= \langle M \rangle_B \rangle_B - \langle M \rangle_B \rangle_A \langle M \rangle_B \rangle_C^{-1} \langle M \rangle_B \rangle_D; \end{aligned} \quad (4.25)$$

$$\begin{aligned} \langle \langle M \rangle_C \rangle &= (\langle M \rangle_C \rangle / \langle M \rangle_C \rangle_B) \\ &= \langle M \rangle_C \rangle_C - \langle M \rangle_C \rangle_D \langle M \rangle_C \rangle_B^{-1} \langle M \rangle_C \rangle_A; \end{aligned} \quad (4.26)$$

$$\begin{aligned} \langle \langle M \rangle_D \rangle &= (\langle M \rangle_D \rangle / \langle M \rangle_D \rangle_A) \\ &= \langle M \rangle_D \rangle_D - \langle M \rangle_D \rangle_C \langle M \rangle_D \rangle_A^{-1} \langle M \rangle_D \rangle_B. \end{aligned} \quad (4.27)$$

A Figura 4.4 exibe o Procedimento Recursivo Retroativo aplicado nos quadros $M \rangle_B \rangle_x$ com $x \in \{A, B, C, D\}$, resultantes das divisões realizadas no ramo do quadro $M \rangle_B$, até a obtenção de $\langle \langle M \rangle_B \rangle$, conforme Equação (4.25). Observe que a aplicação do complemento de Schur em uma matriz em blocos 2×2 gera um único bloco. Os quatro blocos originados,

no Procedimento Recursivo Progressivo, a partir do mesmo ramo da recursão deverão ser agrupados gerando uma nova matriz em blocos 2×2 . Este processo deverá ser repetido até restar um único bloco, conforme descrito na Subseção 4.1.2.

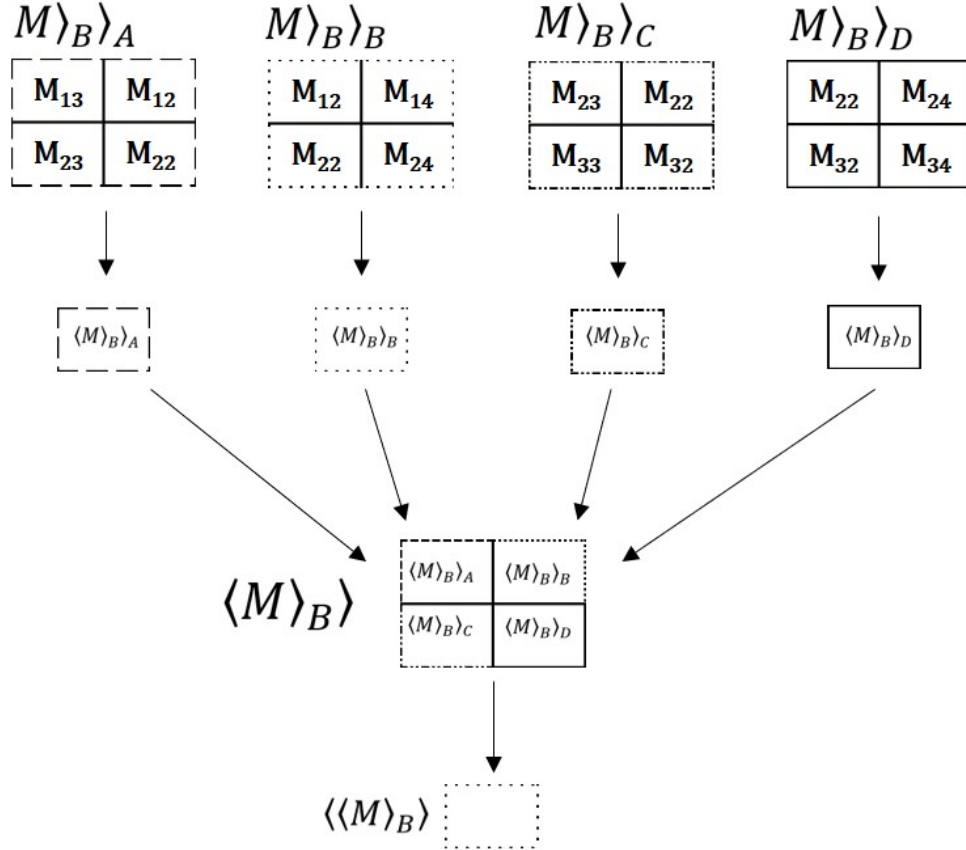


Figura 4.4: Procedimento Recursivo Retroativo nos quadros originados por M_B até a obtenção de $\langle\langle M \rangle_B\rangle$.

Finalmente, seguindo o Passo $k - 1$ da Subseção 4.1.2, seja $\langle\langle M \rangle\rangle$ uma matriz em blocos gerada pelas matrizes resultantes de (4.24), (4.25), (4.26) e (4.27).

$$\langle\langle M \rangle\rangle = \begin{bmatrix} \langle\langle M \rangle_A\rangle & \langle\langle M \rangle_B\rangle \\ \langle\langle M \rangle_C\rangle & \langle\langle M \rangle_D\rangle \end{bmatrix} \quad (4.28)$$

Portanto, $\langle\langle M \rangle\rangle$ é uma matriz em blocos 2×2 gerada a partir da matriz em blocos 4×4 original, como mostrado na Figura 4.6.

Sendo assim, para obter N_{11} , basta calcular a inversa do complemento de Schur de $\langle\langle M \rangle_D\rangle$ em relação a $\langle\langle M \rangle\rangle$, seguindo a mesma fórmula usada para obter E , mostrado no canto superior esquerdo de (3.19), a saber $(M/D)^{-1} = (A - BD^{-1}C)^{-1}$. Aplicando esta

fórmula, obtém-se:

$$\begin{aligned} N_{11} &= (\langle\langle M \rangle\rangle / \langle\langle M \rangle_D \rangle)^{-1} \\ &= (\langle\langle M \rangle_A \rangle - \langle\langle M \rangle_B \rangle \langle\langle M \rangle_D \rangle^{-1} \langle\langle M \rangle_C \rangle)^{-1}. \end{aligned} \quad (4.29)$$

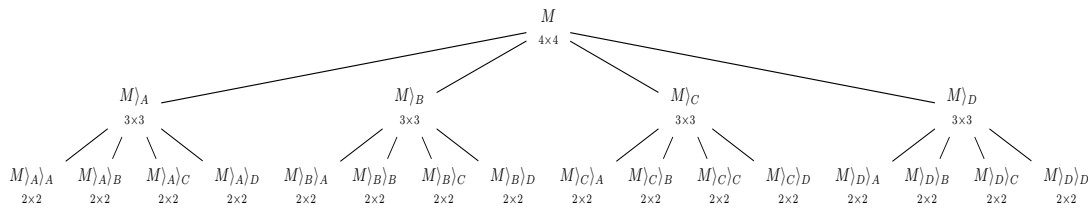


Figura 4.5: Procedimento Recursivo Progressivo

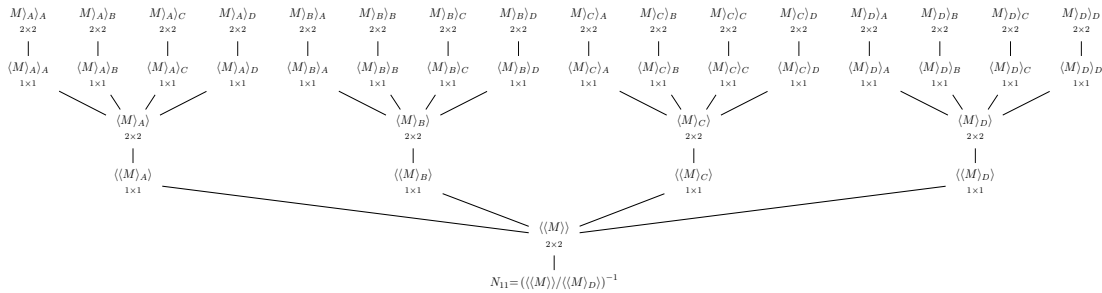


Figura 4.6: Procedimento Recursivo Retroativo

4.3 Implementação

O algoritmo de Inversão Recursiva de Matrizes em Blocos, proposto na Seção 4.1, foi implementado em linguagem C++, usando a biblioteca de álgebra linear Armadillo [Sanderson & Curtin 2016] para a inversão dos blocos de ordem b no Procedimento Recursivo Retroativo.

Devido à estrutura recursiva do algoritmo, é possível armazenar apenas alguns blocos $b \times b$ para executar o cálculo por inteiro. Esta característica permite a inversão de uma matriz de, praticamente, qualquer tamanho a ser calculada usando uma quantidade de memória que se eleva apenas com os níveis da recursão. De fato, se o Procedimento Recursivo Retroativo for realizado sequencialmente, ramo por ramo, e se assumirmos

que a inversão de uma matriz $b \times b$ pode ser calculada no mesmo espaço de memória e consome apenas três vezes o tamanho da matriz $b \times b$, pode-se provar que cada ramo precisa apenas $O(b^2)$ de armazenamento de memória para completar sua computação (ver Subseção 4.4.1).

Além disso, para problemas nos quais a matriz a ser invertida pode ser obtida elemento por elemento, como no treinamento de LS-SVMs [An et al. 2007], não é necessário armazenar a matriz original. Esta pode ser auferida, elemento por elemento, a partir de uma função aplicada à matriz de amostras de entrada que, embora necessite ser armazenada, é frequentemente muitas ordens de magnitude menor. Para esta aplicação, é possível que nem seja necessário calcular todos os blocos da inversa, uma vez que o procedimento de validação cruzada necessita apenas dos blocos mais próximos da diagonal principal.

No entanto, considerando que o BRI calcula um único bloco da matriz inversa, mais especificamente o bloco superior esquerdo, para calcular a inversa completa, é necessário executar o algoritmo k^2 vezes com permutações adequadas de linhas e colunas a fim de obter todos os blocos.

4.4 Análise de Complexidade

A análise de complexidade do BRI pode ser desenvolvida com base na quantidade de operações de complemento de Schur que devem ser aplicadas às matrizes em blocos 2×2 em cada etapa do Procedimento Recursivo Retroativo, descrito na Subseção 4.1.2. O Procedimento Recursivo Progressivo não contém operações aritméticas. Considere os seguintes fatos básicos.

- (a) O custo computacional do produto XY de dois blocos X e Y de ordem b é $O(b^3)$;
- (b) O custo computacional da inversa X^{-1} de um bloco X de ordem b é $O(b^3)$;
- (c) A definição dos complementos de Schur em (3.11), (3.13), (3.14) e (3.15) requer, em cada uma, 1 inversão de matriz e 2 multiplicações matriciais com blocos de ordem b . Assim, o custo computacional de uma operação de complemento de Schur é $O(3b^3)$.

Teorema 1. *Seja $M \in \mathbb{R}^{m \times m}$ uma matriz em blocos invertível $k \times k$, com blocos de ordem b . Então, a complexidade $C_{BRI}(k, b)$ do algoritmo BRI para a inversão de M é:*

$$C_{BRI}(k, b) = O(k^2 b^3 4^{k-1}). \quad (4.30)$$

Demonstração. De acordo com a ideia básica do algoritmo BRI, no Passo i ($i = 1, \dots, k-1$) do Procedimento Recursivo Retroativo, apresentado na Subseção 4.1.2, existem (4^{k-1-i}) matrizes em blocos 2×2 , com blocos de ordem b , para calcular o complemento de Schur por meio de (3.15), (3.14), (3.13) e (3.11). Assim, considerando a propriedade (c), no Passo i ($i = 1, \dots, k-1$) do Procedimento Recursivo Retroativo, são necessárias $3b^3 4^{k-1-i}$ operações. Além disso, no último passo, $k-1$, é preciso inverter a última matriz em blocos para obter N_{11} , que é um bloco de M^{-1} . Feito isso, são necessárias mais $k^2 - 1$ execuções do algoritmo BRI para obter os demais blocos de M^{-1} . Portanto, o algoritmo proposto possui uma complexidade de:

$$C_{BRI}(k, b) = \left(\sum_{i=1}^{k-1} 3b^3 4^{k-1-i} + b^3 \right) k^2 = O(k^2 b^3 4^{k-1}). \quad (4.31)$$

□

4.4.1 Análise do Custo de Memória

De acordo com a implementação proposta do BRI na Seção 4.3, é possível armazenar apenas alguns blocos $b \times b$ para executar o cálculo por inteiro, devido à estrutura recursiva do algoritmo.

Teorema 2. *Seja $M \in \mathbb{R}^{m \times m}$ uma matriz em blocos invertível $k \times k$, com blocos de ordem b . Então, o custo de memória MC_{BRI} do algoritmo BRI para a inversão de M é*

$$MC_{BRI} = O(b^2). \quad (4.32)$$

Demonstração. Suponha que não seja necessário ou que não seja possível armazenar M na memória principal. Além disso, cada elemento de M é acessado apenas para aplicar as operações do complemento de Schur. Então, considere o espaço de memória necessário para armazenar um bloco de M de ordem b como sendo de b^2 unidades.

Suponha que o Procedimento Recursivo Retroativo, definido na Subseção 4.1.2 seja computado sequencialmente, ramo por ramo. Assim, para calcular a operação de complemento de Schur da matriz em blocos 2×2 , com blocos de ordem b , envolvidos em cada ramo, é necessário apenas $3b^2$ unidades de memória para armazenar os 2 operandos envolvidos e a resultado respectivo para cada operação de multiplicação, de inversão e de subtração de blocos. Portanto, precisa-se de $3(b)^2$ unidades para calcular a operação de complemento de Schur em cada ramo por nível i ($i = 1, \dots, k-1$) da recursividade retroativa. Além disso, mais b^2 unidades de memória são necessárias para armazenar um

bloco no nível i enquanto o cálculo do complemento de Schur em um ramo do nível $i - 1$ termina. Então, o algoritmo proposto tem um custo de memória de:

$$MC_{BRI} = 3(b)^2 + (b)^2 = O(b^2). \quad (4.33)$$

□

Vale ressaltar que a complexidade $C_{LU}(k, b)$ e o custo de memória MC_{LU} da decomposição LU e inversão numérica da matriz M são de ordem:

$$C_{LU}(k, b) = O(k^3 b^3) \quad (4.34)$$

e

$$MC_{LU}(k, b) = O(k^2 b^2). \quad (4.35)$$

Conseqüentemente, o método LU é mais rápido que o algoritmo BRI. Por outro lado, o custo da memória do algoritmo proposto é muito menor do que a inversão LU. Este fato é verificado pelos resultados numéricos apresentados na Seção 4.5.

4.5 Resultados Experimentais

Para fins de comparação, foram realizados experimentos com algoritmo BRI e a inversa pela chamada de função do Armadillo `inv(·)` medindo tempo de execução e uso de memória. O `inv(·)` obtém a inversa de uma matriz quadrada geral por fatoração LU usando pivoteamento parcial com permutações de linhas através da integração com a biblioteca matemática LAPACK (do inglês, *Linear Algebra PACKage*) [Anderson et al. 1999].

Para obter a fatoração LU, LAPACK faz uso das rotinas `xGETRF` e `xGETRI`. Especificamente, a função `DGETRF` calcula a fatoração LU de uma matriz $m \times n$ densa usando pivoteamento parcial com permutações de linhas. Uma vez que os dois fatores L e U são conhecidos (onde $A = LU$), através de `DGETRF`, a função `DGETRI` consiste em inverter U , resolvendo então o sistema de matriz triangular $XL = U^{-1}$ (isto é, $L^T X^T = (U^{-1})^T$, portanto $X = A^{-1}$).

Nos experimentos, foram consideradas matrizes quadradas de ordem m , composta de entradas aleatórias, escolhidas de uma distribuição normal com média 0,0 e desvio padrão 1,0 gerada por chamada de função do Armadillo `randn()`. Os experimentos foram realizados com matrizes com diferentes quantidades de blocos ($k \times k$), para o BRI,

comparando-os com matrizes não particionadas do mesmo tamanho, para inversão LU.

As medições do uso de memória foram realizadas com uma ferramenta de perfilamento de memória do *framework* Valgrind, chamado *Massif* [Nethercote & Seward 2007].

Os experimentos foram executados em um computador com um processador AMD AthlonTM II X2 B28 com sistema operacional Linux, com 4GB de RAM.

Esta seção descreve os experimentos, bem como os resultados obtidos. Na Subseção 4.5.1, são apresentadas as medidas de uso da memória física durante o processamento da inversão BRI e LU. E, na Subseção 4.5.2, os resultados em tempo de execução são analisados.

4.5.1 Uso de Memória

Para fins de testes, tanto o algoritmo BRI quanto o método de inversão LU foram utilizados para calcular a matriz A_γ^{-1} usada no método de validação cruzada de LS-SVM proposto por [An et al. 2007], sendo

$$A_{\gamma_{m \times m}} = \begin{bmatrix} 0 & \mathbf{1}_n^T \\ \mathbf{1}_n & K_\gamma \end{bmatrix} \quad (4.36)$$

com $\mathbf{1}_n = [1, 1, \dots, 1]^T$, $K_\gamma = K + \frac{1}{\gamma}I_n$ e $K_{i,j} = K(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$.

A função $K(\cdot, \cdot)$ é o *kernel* do tipo Gaussiano ou Função de Base Radial (RBF, do inglês *Radial Basis Function*) e $\{x_i\}_{i=1}^n$ é um conjunto de dados de entrada. Para os testes, este conjunto foi composto de entradas aleatórias, escolhidas de uma distribuição normal com média 0,0 e desvio padrão 1,0, gerada pelo comando `randn()` da biblioteca Armadillo.

O uso de memória para inversões em relação à quantidade de blocos ($k \times k$), para o BRI, e em relação às matrizes não particionadas, para a inversão LU, é mostrado na Figura 4.7.

O BRI consome, claramente, menos memória física do que a inversão LU. Considerando uma matriz de entrada da mesma dimensão $m \times m$, à medida que o número de blocos aumenta, $k \times k$, o uso da memória diminui. Isso ocorre porque a dimensão $b \times b$ dos blocos diminui, onde $b = \frac{m}{k}$, em relação a (3.5). Consequentemente, menos dados são mantidos na memória durante todo o processo de inversão. Assim, a aplicação da inversão recursiva nos permite considerar matrizes A_γ , de ordem m , muito maiores do que aquelas que a inversão LU permite.

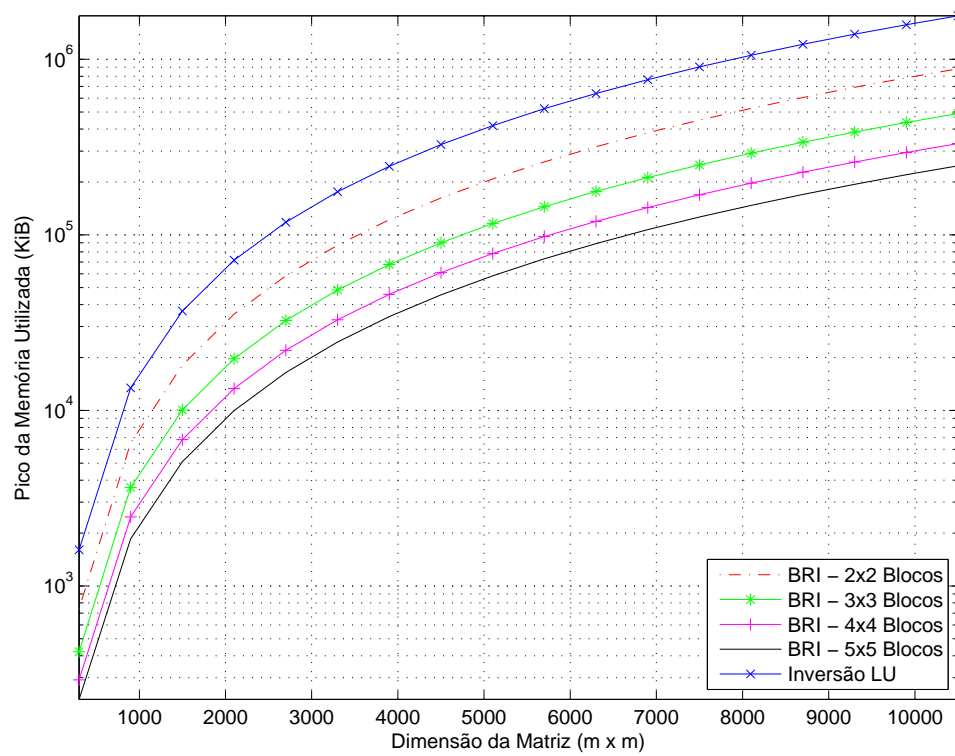


Figura 4.7: Uso de memória para a computação da inversa de matrizes $m \times m$ usando decomposição LU e o algoritmo BRI com diferentes números de blocos.

4.5.2 Tempo de Execução

A Figura 4.8 mostra os tempos de processamento obtidos na inversão de matrizes $m \times m$ para diferentes quantidades de blocos ($k \times k$), usando BRI, e para matrizes não particionadas, usando LU. A plotagem dos gráficos mostra que o tempo de processamento do BRI é maior do que a inversão LU, aumentando à medida que aumentamos o número de blocos que A_γ é particionada.

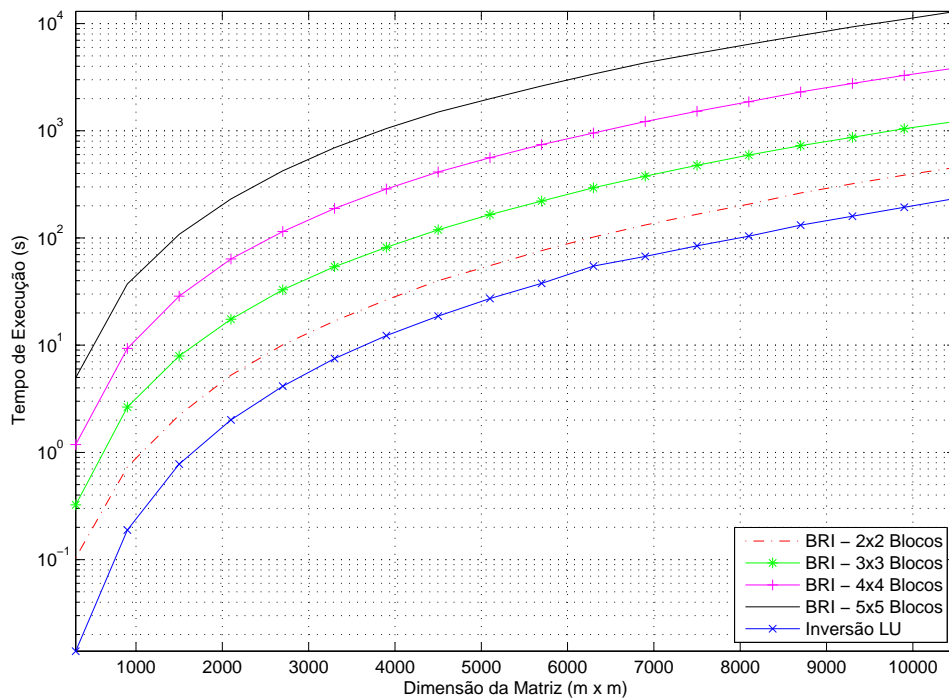


Figura 4.8: Tempo de execução da computação da inversa de matrizes $m \times m$ usando decomposição LU e o algoritmo BRI com diferentes números de blocos.

Comparando as Figuras 4.7 e 4.8, observe que o uso do BRI para calcular o inverso de uma matriz $m \times m$ produz um *trade-off* entre uso da memória e tempo de processamento. É possível trocar um aumento no tempo de processamento pelo consumo de memória ao variar a quantidade de blocos, durante o processo de inversão matricial e, consequentemente, permitir a inversão de matrizes muito grandes que, de outra forma, não se encaixariam na memória. Além disso, para evitar o uso de memória mais lenta, o valor de k pode ser escolhido de modo que os níveis mais baixos da hierarquia da memória, citados na Seção 3.3, sejam evitados.

Capítulo 5

BRI Aplicado à Validação Cruzada de LS-SVM

A Aprendizagem de Máquina (em inglês, *machine learning*), sub-campo da Ciência da Computação, consiste no desenvolvimento de algoritmos e técnicas que possibilitem, a uma máquina, a capacidade de adquirir conhecimentos a partir de exemplos. Segundo Mitchell (1997), é dito que um programa de computador aprende, quando seu desempenho em relação a alguma tarefa é aperfeiçoado através da experiência. Essa capacidade de aprendizagem deve ser mensurada através de alguma medida de desempenho P .

Sob esta perspectiva, uma máquina de aprendizagem deve ter a propriedade de, após a observação de vários pares $\{x_i, y_i\}_{i=1}^n$ — onde x_i representa um exemplo e y_i denota o seu rótulo — imitar o comportamento do sistema, gerando saídas próximas de y_i a partir de entradas próximas de x_i [Vapnik 1995]. Caso o número de padrões (saídas ou classes) seja finito, normalmente números naturais, a tarefa é denominada classificação de padrões. Se houver apenas duas classes possíveis, denominam-se classificação binária. E, finalmente, quando existe um número infinito de padrões possíveis (valores reais), são conhecidos como problemas de regressão.

Para que uma máquina de aprendizagem seja capaz de produzir um classificador (ou modelo) capaz de prever precisamente o rótulo de novos dados, ela deve ser submetida a um processo de aprendizagem, chamado de treinamento. Assim sendo, o objetivo do treinamento consiste em ajustar os parâmetros livres da máquina de aprendizagem de forma a encontrar uma ligação entre os pares entrada e saída [Braga et al. 2000]. O classificador obtido também pode, então, ser visto como uma função f , a qual recebe um dado x e fornece uma predição y .

Na estratégia de aprendizado denominado de aprendizagem supervisionada, os dados de treinamento juntamente com suas saídas correspondentes são apresentadas à máquina objetivando o ajuste de seus parâmetros e a obtenção de uma função inferida, que possa ser

utilizada no mapeamento de novos exemplos. Isso exige que o algoritmo de aprendizado tenha a capacidade de generalizar, a partir dos dados de treinamento, para novas entradas de uma maneira razoável. Um exemplo de uma técnica que lida com o aprendizado de máquina supervisionado corresponde às Máquinas de Vetor de Suporte por Mínimos Quadrados (LS-SVM) [Carvalho et al. 2002]. Na formulação da LS-SVM, são os chamados hiperparâmetros, a saber, os parâmetros do *kernel* e o parâmetro de regularização, que regem o desempenho da generalização dos seus classificadores.

Por sua vez, um método trivial utilizado para estimar a capacidade de generalização de um modelo, bem como calcular a taxa de predições corretas ou incorretas (também denominadas taxa de acerto e taxa de erro, respectivamente), obtidas pelo classificador, sobre novos dados é a Validação Cruzada (CV).

Conforme será apresentado na Seção 5.3, o método de validação cruzada do tipo *l*-partições (do inglês, *l-folds*) para LS-SVM, proposto por [An et al. 2007], é centrado na obtenção da inversa da matriz de *kernel* (K_Y^{-1}) e na resolução de *l* sistemas de equações, onde *l* é o número de partições, para estimar os rótulos previstos nos subconjuntos de testes omitidos. Segundo Edwards et al. (2013), os *l* sistemas de equações baseiam-se no uso dos blocos diagonais da matriz em blocos **C**. Esses blocos diagonais são designados por C_{kk} e têm dimensão de $n_v \times n_v$, onde *k* é a *k*-ésima partição; $n_v \simeq \frac{n}{l}$ e *n* é quantidade total de amostras de entrada.

An et al. (2007) percebeu que a resolução de $\beta = C_{kk}^{-1} \alpha_k$ fornece o erro estimado pelas amostras omitidas na *k*-ésima partição. Contudo, esta técnica lida com a inversão da matriz de *kernel* K_Y inteira para obter os blocos diagonais, C_{kk} , da matriz **C**. Esta matriz de *kernel* se apresenta como sendo de alta ordem para grandes conjuntos de dados de entrada. Sendo assim, a escalabilidade deste método fica restrita à memória disponível pelo sistema computacional. Por conseguinte, a fim de possibilitar um melhor ajuste dos dados na memória, o algoritmo BRI, apresentado no Capítulo 4, foi aplicado para calcular tanto os blocos diagonais, C_{kk} , quanto os blocos da matriz K_Y^{-1} , de acordo com a necessidade, durante a execução do algoritmo CV para grandes conjuntos de amostras de entrada em LS-SVMs. O consumo de memória foi, desta forma, reduzido de $O(n^2)$ para $O(n_v^2)$.

Neste capítulo, são expostos conceitos básicos de LS-SVM e validação cruzada. Em seguida, é apresentado o algoritmo eficiente para validação cruzada, definido em [An et al. 2007], e exposto o algoritmo resultante ao aplicar o BRI à validação cruzada das LS-SVMs. Por fim, faz-se uma explanação dos resultados experimentais obtidos.

5.1 Máquinas de Vetor de Suporte por Mínimos Quadrados

A Máquina de Vetor de Suporte por Mínimos Quadrados (do inglês, *Least Squares Support Vector Machine* – LS-SVM), proposta por Suykens & Vandewalle (1999), é uma máquina de aprendizagem que corresponde a uma versão modificada da Máquina de Vetor de Suporte (do inglês, *Support Vector Machine* – SVM) [Vapnik 1998] e que tem sido adotada com sucesso em muitas aplicações [Suykens et al. 2002].

A LS-SVM busca uma menor complexidade computacional em relação à SVM, sem perder qualidade nas soluções. Assim, consiste na utilização de uma função de custo de mínimos quadrados e no uso de restrições de igualdade na função de custo primal a ser minimizada. Essas modificações permitem que seu treinamento seja realizado através da resolução de um conjunto de equações lineares ao invés da programação quadrática, usada na SVM.

5.1.1 Classificação de Padrões em LS-SVM

Dado o conjunto de treinamento $\{x_i, y_i\}_{i=1}^n$, com entradas $x_i \in \mathbb{R}^n$ e saída binária correspondente $y_i \in \{-1, +1\}$, a LS-SVM objetiva construir um hiperplano ótimo (com a maior margem de separação possível) para separar os vetores da classe -1 dos da classe +1. Desta forma, a superfície de decisão $f(x) = 0$ é definida como:

$$w^T \phi(x) + b = 0, \quad (5.1)$$

onde $w \in \mathbb{R}^n$ é o vetor de pesos, b é o termo de polarização e $\phi(\cdot)$ é o mapeamento do espaço de entrada em um espaço de alta dimensionalidade, denominado espaço de características. Este mapeamento permite à LS-SVM atuar em problemas de classificação não-linear, tendo em vista que, neste espaço de características, os padrões de entrada têm uma alta probabilidade de serem linearmente separáveis [Carvalho 2005a]. Ele é usualmente fornecido por uma função de *kernel*, conforme será aprofundado na Subseção 5.1.2.

Conforme An et al. (2007), um classificador assume a seguinte forma:

$$y(x) = \text{sign}[w^T \phi(x) + b]. \quad (5.2)$$

Portanto, o processo de treinamento de uma LS-SVM consiste na obtenção dos valores para os pesos w e para o termo de polarização b de maneira a minimizar a função de custo

$J_P(w, b, e)$, sendo o problema de classificação formulado como:

$$\min_{w,b,e} J_P(w, b, e) = \frac{1}{2} w^T w + \gamma \frac{1}{2} \sum_{i=1}^n e_i^2, \quad (5.3)$$

sujeito a

$$y_i [w^T \phi(x_i) + b] = 1 - e_i, \text{ para } i = 1, \dots, n.$$

na qual γ é um parâmetro, dito de regularização, usado para ponderar os dois termos de $J_P(w, b, e)$; e e_i é a variável de erro, permitida de forma que erros de classificação possam ser tolerados em caso de sobreposição de distribuições.

É válido observar que o método de aprendizado das LS-SVMs baseia-se em duas modificações na formulação da SVM. Em primeiro lugar, a função de perda é definida como o erro quadrado em todas as amostras. E, em segundo lugar, as restrições de desigualdade são substituídas por restrições de igualdade [Suykens & Vandewalle 1999].

O objetivo das LS-SVMs é, enfim, encontrar um hiperplano ótimo que maximize a margem de separação Δ entre as classes. A Figura 5.1 ilustra um conjunto de classificadores lineares que separam duas classes, mas apenas um (em destaque) maximiza a margem de separação (distância entre o hiperplano e a amostra mais próxima de cada classe).

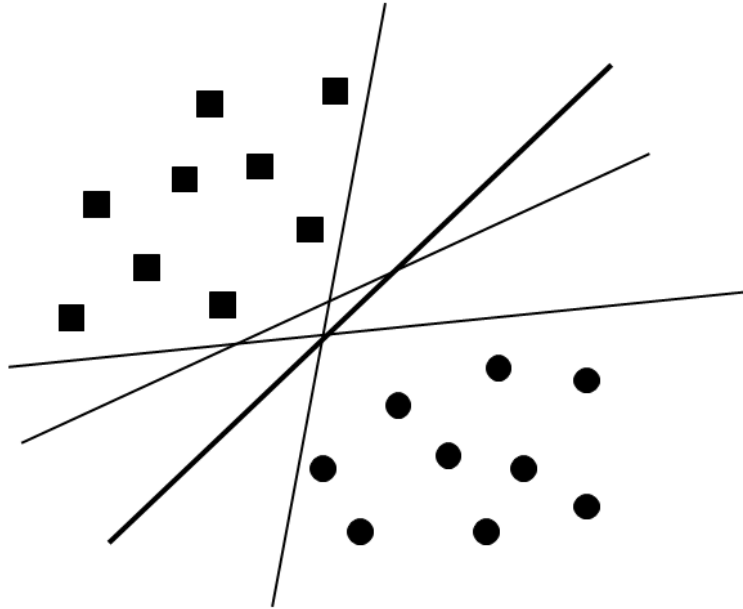


Figura 5.1: Hiperplano de separação ótimo.

Por sua vez, a Figura 5.2 apresenta um hiperplano ótimo para um espaço de entrada bidimensional. É possível observar também que os vetores de suporte são os pontos

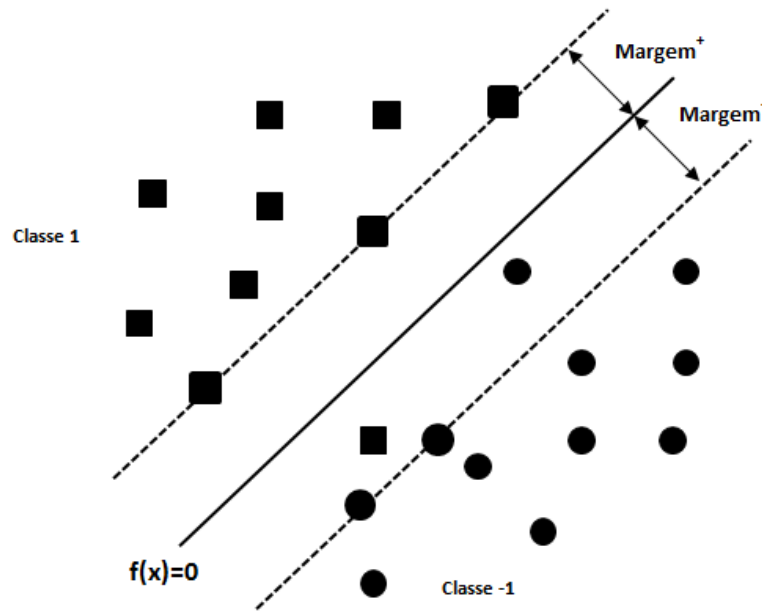


Figura 5.2: Maximização da margem de separação. Os vetores em destaque correspondem aos vetores de suporte.

(amostras de treinamento) mais próximas da margem de separação que serviram de base para a definição do hiperplano ótimo.

Conforme apresentado em Suykens et al. (2002), aplicando-se o método de otimização dos multiplicadores de Lagrange [Fletcher 1987], a solução do problema de minimização no espaço primal pode ser obtida pela resolução do seguinte sistema linear:

$$\begin{bmatrix} 0 & 1_n^T \\ 1_n & K + \frac{1}{\gamma} I_n \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix}, \quad (5.4)$$

com $y = [y_1, y_2, \dots, y_n]$, $1_n = [1, 1, \dots, 1]^T$, $\alpha = [\alpha_1, \alpha_2, \dots, \alpha_n]^T$ e K representa a matriz de kernel, definida na Subseção 5.1.2. A resolução deste sistema linear produz a seguinte função de classificação das LS-SVMs:

$$y(x) = \text{sign} \left[\sum_{i=1}^n \alpha_i K(x_i, x) + b \right], \quad (5.5)$$

onde α e b são as soluções de (5.4).

Sendo assim, a solução do sistema de equações lineares em (5.4) é a mesma do problema primal apresentado em (5.3), tendo o termo de polarização b como sendo o primeiro elemento do vetor solução e os demais elementos correspondendo aos multiplicadores de

Lagrange α_i , associados aos vetores de treinamento x_i .

Diferentemente da SVM, na LS-SVM praticamente todos os vetores de treinamento contribuem para a construção do hiperplano. Isto é resultante do fato de que esses vetores possuem multiplicadores de Lagrange não nulos, sendo considerados vetores de suporte.

5.1.2 Funções de *Kernel*

Na funções definidas em (5.1)-(5.3), aparece um produto interno de funções mapeadoras, representado por $\varphi(\cdot)$, que pode ser substituído por uma única função, denominada função de *kernel*, representado por K , conforme apresentado em (5.4) e (5.5). Um *kernel* K é uma função simétrica definida semi-positiva que recebe dois pontos x_i e x_j , do espaço de entrada, e retorna o produto escalar desses dados no espaço de características [Hebrich 2002]. Tem-se que:

$$K(x_i, x_j) = \varphi(x_i)^T \varphi(x_j). \quad (5.6)$$

É bastante comum aplicar a função *kernel* sem conhecer o mapeamento φ , que é gerado implicitamente. Segundo Lorena & de Carvalho (2007), a utilidade dos *kernels* está na simplicidade de seu cálculo e em sua capacidade de representar espaços abstratos.

Esta função deve satisfazer ao Teorema de Mercer [Mercer 1909] para garantir a convexidade do problema de otimização e para que o *kernel* apresente o mapeamento, no qual seja possível o cálculo de produtos escalares. De maneira simplificada, um *kernel* que satisfaz as condições de Mercer é caracterizado por dar origem a matrizes positivas semidefinidas \mathbf{K} , em que cada elemento K_{ij} é definido por $K_{ij} = K(x_i, x_j)$, para todo $i, j = 1, \dots, n$ [Hebrich 2002].

Na Tabela 5.1, encontram-se algumas funções comumente utilizadas como funções de *kernel*.

Kernel	Expressão	Parâmetro
Linear	$(x_i^T x_j)$	
RBF	$e^{-\ x_i - x_j\ ^2 / 2\sigma^2}$	σ^2
Polinomial	$(x_i^T x_j + a)^b$	a, b
Sigmóide	$\tanh(\beta_0 x_i^T x_j + \beta_1)$	β_0, β_1

Tabela 5.1: Algumas funções usadas como kernel

Tendo em vista que a superfície de decisão é sempre linear no espaço das características e, normalmente, não linear no espaço de entrada, problemas não-linearmente separáveis podem ser resolvidos, pelas LS-SVMs, graças às funções de *kernel*.

Neste estudo de caso, optou-se por usar a função radial de base (RBF), também denominado *kernel* Gaussiano, por ser a função mais comumente utilizada em LS-SVM.

5.2 Validação Cruzada (*Cross-Validation*)

A validação cruzada é uma técnica usada para avaliar a taxa de erros de um modelo a fim de se conseguir sistemas com boa capacidade de generalização. Ela permite que todo o conjunto de dados seja usado para treinamento [Stone 1978]. Existem várias formas de se implementar tal estratégia, mas para este trabalho é suficiente a explanação das principais técnicas abordadas a seguir.

No primeiro método, denominado de *l*-partições (do inglês, *l-Fold Cross Validation – l-fold CV*), o conjunto total de dados (amostras) disponíveis é dividido aleatoriamente em *l* subconjuntos de tamanhos iguais (aproximadamente) sendo que $(l - 1)$ delas comporão o subconjunto de treinamento, enquanto que a partição restante constituirá o subconjunto de teste [Silva et al. 2010].

Desta forma, sendo *n* o número total de padrões do conjunto de dados, cada subconjunto de treinamento será constituído de $\frac{n}{l}(l - 1)$ padrões e cada subconjunto de teste conterá $\frac{n}{l}$ padrões, que serão utilizados para cálculo do erro de generalização.

Durante o processo de aprendizagem, o classificador é treinado *l* vezes. A cada ciclo, um subconjunto é deixado de fora do conjunto de treinamento, que servirá para calcular os erros de classificação. No final do processo, todas as partições terão sido utilizadas como subconjunto de teste. A Figura 5.3 ilustra o processo desta estratégia para um total de 20 amostras, assumindo um valor *l* igual a 5, que implica necessariamente na realização de um número igual de experimentos.

O valor do parâmetro *l* está atrelado à quantidade total de amostras disponíveis, sendo usualmente atribuído um número compreendido entre 5 e 10.

Em um caso particular do método de *l*-partições, denominado de validação cruzada por unidade (do inglês, *Leave-One-Out Cross Validation – LOO-CV*), o parâmetro *l* corresponde ao número total de amostras disponíveis. Neste caso, apenas uma única amostra é considerada para o subconjunto de teste, sendo as demais alocadas para o subconjunto de treinamento [Silva et al. 2010]. O processo de aprendizagem é então repetido até que todas as amostras sejam individualmente utilizadas como subconjunto de teste. Contudo, tem-se aqui um elevado esforço computacional, uma vez que o processo de aprendizagem será repetido um número de vezes que será igual ao tamanho do conjunto total de amostras.

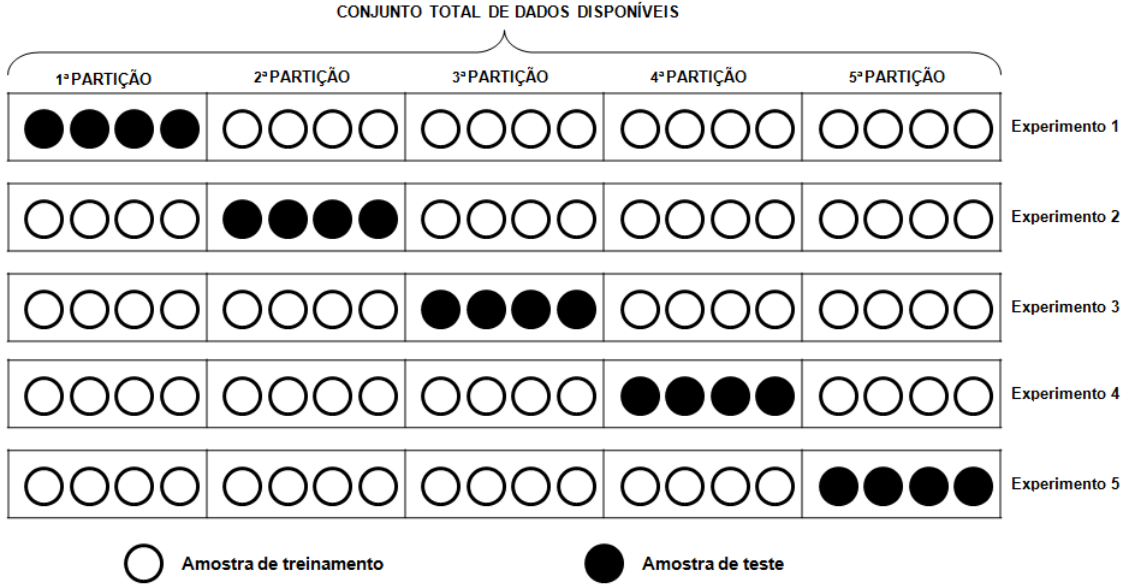


Figura 5.3: Método de validação cruzada utilizando l -partições.

Por sua vez, o LMO-CV (do inglês, *Leave-More-Out Cross Validation*), também conhecido por LNO-CV (do inglês, *Leave-N-Out Cross Validation*), é altamente recomendável para testar a robustez de um modelo, contudo é uma versão bastante custosa de validação cruzada que envolve deixar de fora todos os possíveis subconjuntos de m exemplos de treinamento [An et al. 2007]. No LMO-CV, o classificador é treinado e validado C_m^n vezes (onde n é número total de padrões do conjunto de dados original). Assim, sendo n muito grande, torna-se impossível de calcular.

Já o Monte-Carlo LMO-CV (do inglês, *Monte-Carlo Leave-More-Out Cross Validation*), sugerido por Shao (1993), define cada subconjunto de validação selecionando aleatoriamente m padrões para um número de vezes suficiente, dito L . A principal diferença entre o Monte-Carlo LMO-CV e o l -fold CV é que, no Monte-Carlo LMO-CV, os diferentes subconjuntos de teste são escolhidos aleatoriamente e não é necessário serem disjuntos.

5.3 Trabalhos Relacionados

Conforme apresentado na Subseção 5.1.1, para as LS-SVM, praticamente quase todos os vetores de treinamento possuem multiplicadores de Lagrange não nulos, o que caracteriza ausência da propriedade denominada esparsidade da solução. Para análise de dados em grande escala, resolver (5.3) no dual não é vantajoso, pois o tamanho da matriz da

solução é igual ao tamanho dos dados originais. Vários trabalhos na literatura, incluindo Geebelen et al. (2012), Burges (1996), Downs et al. (2002), Suykens et al. (2000) e Li et al. (2006) abordam o problema da esparsidade no modelo LS-SVM. Entretanto, essas técnicas não podem garantir uma grande redução no número de vetores de suporte.

A abordagem, proposta inicialmente em Suykens et al. (2002), denominada de Máquinas de Vetor de Suporte por Mínimos Quadrados de Tamanho Fixo (do inglês, *Fixed-size Least Squares Support Vector Machines* – FS-LSSVM) fixa o número de vetores de suporte, referidos como vetores protótipos (do inglês, *prototype vectors* – PVs), com antecedência. Ele fornece uma solução para o problema LS-SVM no espaço primal resultando em um modelo paramétrico e uma representação esparsa. Este método usa uma expressão explícita para o mapa de características, utilizando o método Nyström [Williams & Seeger 2001], e conjunto de PVs de cardinalidade $M \ll N$, onde N é o conjunto dos dados de entrada. Todavia, não é uma solução esparsa e a escolha de um valor ótimo de M é um problema em aberto.

Nos últimos anos, a norma l_0 (do inglês, *L_0 -norm*) tem recebido cada vez mais atenção. A norma l_0 conta o número de elementos diferentes de zero de um vetor. Sendo assim, quando a norma l_0 de um vetor é minimizada, resulta-se em modelos esparsos. Entretanto, este problema é NP-completo. Várias aproximações são discutidas em [Weston et al. 2003] e abordada em FS-LSSVM em [Mall & Suykens 2002] e [Huang et al. 2010]. As principais desvantagens dos métodos descritos em [Huang et al. 2010] e [López et al. 2011] são que estas abordagens não podem ser aplicadas para conjuntos de dados muito grandes devido às restrições de memória (matriz de *kernel* de comprimento $N \times N$) e computacionais ($O(N^3)$ de tempo).

Por sua vez, An et al. (2007) propôs um eficiente método de validação cruzada para o cálculo do erro de generalização em classificação LS-SVM e para implementar dois algoritmos para avaliações l -fold CV e LMO-CV. Este método é centrado em torno do cálculo de uma matriz inversa e resolução de l sistemas de equações, onde l é o número de partições. Conforme exposto no Algoritmo 6, o cálculo da inversa envolve computar K_Y^{-1} e os l sistemas de equações são baseados no uso das matrizes de bloco diagonais encontradas em **C** (linha 2) para estimar os rótulos esperados pelas partições reservadas para teste. Essas matrizes de bloco diagonais são denotadas por C_{kk} e têm dimensão $n_v \simeq \frac{n}{l}$, onde k é a k -ésima partição e n é o número de pontos de dados. [An et al. 2007] percebeu que a solução $C_{kk}\beta_{(k)} = \alpha_{(k)}$ fornece erros estimados pelos exemplos omitidos na k -ésima partição.

Este algoritmo pode ser facilmente modificado para suportar regressão LS-SVM e KRR (do inglês, *Kernel Ridge Regression*). Para ser aplicado em problemas de regressão

Algorithm 6 Validação Cruzada Eficiente**Entrada:** K (matriz do *kernel*), l (número de partições), y (resposta)

1. $K_\gamma^{-1} \leftarrow \text{inv}(K + \frac{1}{\gamma}I)$, $d \leftarrow 1_n^T K_\gamma^{-1} 1_n$
2. $C \leftarrow K_\gamma^{-1} + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1}$
3. $\alpha \leftarrow K_\gamma^{-1} y + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1} y$
4. $n_k \leftarrow \text{size}(y)/l$, $y^{(k)} \leftarrow \text{zeros}(l, n_k)$
5. **for** $k \leftarrow 1, k \leq l$ **do**
6. Resolva $C_{kk} \beta_{(k)} = \alpha_{(k)}$
7. $y^{(k)} \leftarrow \text{sign}[y_{(k)} - \beta_{(k)}]$
8. $k \leftarrow k + 1$
9. **end for**
10. $\text{erro} \leftarrow \frac{1}{2} \sum_{k=1}^l \sum_{i=1}^{n_k} |y_{k,i} - y^{(k,i)}|$

Saída: *erro*

basta apenas remover a função sinal (*sign*) e alterar a função de erro.

Para tratar conjunto de dados grandes, An et al. (2007) implementou algoritmos aproximados usando a decomposição de Cholesky incompleta.

5.4 Algoritmo BRI aplicado à Validação Cruzada de LS-SVM

Analisando o Algoritmo 6, observa-se que é necessário o armazenamento das matrizes K_γ^{-1} e C , mesmo fazendo uso apenas dos blocos diagonais de C , para estimar os rótulos esperados pelas partições reservadas para teste ($\beta_{(k)}$), conforme apresentado nas Linhas 5-8 do mesmo. Uma vez que a dimensão da matriz K_γ (e, consequentemente, de C) é proporcional ao tamanho do conjunto das amostras de entrada, a obtenção de K_γ^{-1} pode ser uma tarefa bastante desafiadora para conjuntos de dados muito grandes.

A fim de permitir uma economia de memória e, portanto, expandir o uso do método eficiente de validação cruzada para LS-SVM, definido em An et al. (2007), para grandes conjuntos de entrada, foi feita uma adequação no Algoritmo 6 e aplicação do algoritmo BRI nas etapas de obtenção dos blocos da matriz inversa em questão.

5.4.1 Definições

Seja $\{x_i, y_i\}_{i=1}^n$, o conjunto de dados de treinamento, com $x_i \in \mathbb{R}^n$ e $y_i \in \{-1, +1\}$. Considerando a técnica *l-fold CV*, essas amostras são divididas em l subconjuntos $\{x_{k,i}\}_{i=1}^{n_k}$

de tamanhos aproximadamente iguais (n_v), onde k representa cada um dos l subconjuntos e n_k , a quantidade de entradas por subconjunto. Portanto, $k = 1, 2, \dots, l$; $\sum_{k=1}^l n_k = n$ e $n_k \approx n_v$. Em correspondência, os rótulos y e os multiplicadores de Lagrange α , da Equação (5.4), foram também divididos em l subvetores, a saber: $y = [y_{(1)}, y_{(2)}, \dots, y_{(l)}]^T$ e $\alpha = [\alpha_{(1)}, \alpha_{(2)}, \dots, \alpha_{(l)}]^T$, nos quais $y_{(k)} = [y_{k,1}, y_{k,2}, \dots, y_{k,n_k}]$ e $\alpha_{(k)} = [\alpha_{k,1}, \alpha_{k,2}, \dots, \alpha_{k,n_k}]$.

Para simplificar a implementação do novo algoritmo de validação cruzada, tendo em vista que o BRI obtém o bloco superior esquerdo da matriz inversa, aplicamos permutações adequadas de linhas e colunas a fim de posicionar o bloco K_γ na posição superior esquerda da matriz A_γ . Então, a Equação (5.4) passa a corresponder a:

$$\begin{bmatrix} K + \frac{1}{\gamma} I_n & 1_n \\ 1_n^T & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ b \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix}, \quad (5.7)$$

com $y = [y_{(1)}, y_{(2)}, \dots, y_{(n)}]^T$, $1_n = [1, 1, \dots, 1]$, $\alpha = [\alpha_{(1)}, \alpha_{(2)}, \dots, \alpha_{(n)}]^T$ e K , uma matriz de kernel que satisfaça as condições apresentadas da Subseção 5.1.2.

Sendo assim, por analogia a [An et al. 2007] considere as seguintes definições:

$$K_\gamma \triangleq K + \frac{1}{\gamma} I_n, \quad (5.8)$$

$$A_\gamma \triangleq \begin{bmatrix} K_\gamma & 1_n \\ 1_n^T & 0 \end{bmatrix} \text{ e} \quad (5.9)$$

$$A_\gamma^{-1} \triangleq \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} & C_1 \\ C_{12}^T & C_{22} & \cdots & C_{2l} & C_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ C_{1l}^T & C_{2l}^T & \cdots & C_{ll} & C_l \\ C_1^T & C_2^T & \cdots & C_l^T & C \end{bmatrix}, \quad (5.10)$$

no qual C é um escalar, $C_i \in \mathbb{R}^{n_i}$ e $C_{ij} \in \mathbb{R}^{n_i \times n_j}$ para $i, j = 1, 2, \dots, l$. E, finalmente, seja:

$$\mathbf{C} \triangleq \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} \\ C_{12}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ C_{1l}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix}. \quad (5.11)$$

É oportuno observar que a matriz \mathbf{C} corresponde a uma submatriz da matriz A_γ^{-1} , obtida removendo-se a linha de blocos mais inferior e a coluna de blocos mais à direita

da Equação (5.10). Posto isto, formula-se:

Teorema 3. *Seja $y^{(k)}(x) = \text{sign}[g_k(x)]$, o classificador formulado deixando o k -ésimo subconjunto de fora e seja $\beta_{k,i} = y_{k,i} - g_k(x_{k,i})$. Então,*

$$\beta_{(k)} = C_{kk}^{-1} \alpha_{(k)}, \quad k = 1, 2, \dots, l, \quad (5.12)$$

onde $\beta_{(k)} = [\beta_{k,1}, \beta_{k,2}, \dots, \beta_{k,n_k}]^T$. A prova deste Teorema está disponibilizada em [An et al. 2007].

5.4.2 Algoritmo CV usando BRI

De acordo com An et al. (2007), o Teorema 3 fornece uma fórmula exata para computar as respostas previstas, pelo classificador, referentes aos exemplos excluídos no procedimento l -fold CV. As saídas são exatamente as mesmas obtidas pelo treinamento direto dos classificadores LS-SVM para cada divisão dos exemplos de treinamento. Assim, seguindo o Algoritmo 6, uma vez que K_γ está disponível (linha 1), calcula-se \mathbf{C} e α , conforme as equações apresentadas nas linhas 2 e 3, respectivamente, e, conseqüentemente, $\beta_{(k)}$ resolvendo o sistema linear $C_{kk}\beta_{(k)} = \alpha_{(k)}$.

Objetivando um uso limitado de memória, o Algoritmo 6 foi adaptado de forma que os blocos de \mathbf{C} , a saber C_{ki} e C_{kk} , passassem a ser obtidos, utilizando o algoritmo BRI, de acordo com sua necessidade, para calcular $\alpha_{(k)}$ e $\beta_{(k)}$. Essa alteração possibilita a sua aplicação em situações nas quais os dados de entrada não se ajustam à memória disponível. Para tanto, tem-se que:

Teorema 4. *Seja $\alpha_{(k)} = [\alpha_{k,1}, \alpha_{k,2}, \dots, \alpha_{k,n_k}]$, o subvetor composto pelos multiplicadores de Lagrange obtido a partir do k -ésimo subconjunto do conjunto total de dados de entrada, $\{x_i, y_i\}_{i=1}^n$, com $x_i \in \mathbb{R}^n$ e $y_i \in \{-1, +1\}$. E seja $C_{ki} \in \mathbb{R}^{n_k \times n_k}$, para $k, i = 1, 2, \dots, l$; com n_k representando a quantidade de amostras contidas no subconjunto k . Então,*

$$\alpha_{(k)} = \sum_{i=1}^l C_{ki} y_{(i)}, \quad k, i = 1, 2, \dots, l, \quad (5.13)$$

onde $\alpha = [\alpha_{(1)}, \alpha_{(2)}, \dots, \alpha_{(l)}]^T$ e $y = [y_{(1)}, y_{(2)}, \dots, y_{(l)}]^T$.

Demonstração. Conforme as Linhas 2 e 3 do Algoritmo 6, em [An et al. 2007], observa-se que:

$$\alpha = \mathbf{C}y. \quad (5.14)$$

com \mathbf{C} correspondendo à Equação (5.11). Então,

$$\begin{bmatrix} \alpha_{(1)} \\ \alpha_{(2)} \\ \vdots \\ \alpha_{(l)} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} \\ C_{21} & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ C_{l1} & C_{l2} & \cdots & C_{ll} \end{bmatrix} \begin{bmatrix} y_{(1)} \\ y_{(2)} \\ \vdots \\ y_{(l)} \end{bmatrix}, \quad (5.15)$$

com l sendo a quantidade de subconjuntos cujas n amostras de entrada foram divididas durante o processo de validação cruzada l -fold CV. E, conseqüentemente, $k = 1, 2, \dots, l$, com $\sum_{k=1}^l n_k = n$.

Portanto,

$$\alpha_{(k)} = \sum_{i=1}^l C_{ki} y_{(i)}.$$

□

É importante observar que, conforme a Equação (5.13), é possível obter subdivisões do vetor α , a saber $\alpha_{(k)}$, uma vez que a linha de blocos k da matriz \mathbf{C} esteja disponível. Desta forma, pode-se obter $\beta_{(k)}$ a partir de $\beta_{(k)} = C_{kk}^{-1} \alpha_{(k)}$, dinamicamente, sem a necessidade de manter o vetor α completamente na memória. Essas adequações foram incorporadas pelas linhas 5 e 9 do novo algoritmo de validação cruzada eficiente com o uso limitado de memória, proporcionado pelo BRI, apresentado em Algoritmo 7.

Considerando que a matriz \mathbf{C} corresponde a uma submatriz da matriz A_γ^{-1} , os blocos C_{ki} e C_{kk} são obtidos a partir do processo de inversão da matriz A_γ , definida em (5.9), utilizando o algoritmo BRI, apresentado na Seção 4.1 do Capítulo 4. Assim, no Algoritmo 7, o bloco C_{ki} e a inversa do bloco C_{kk} são calculados segundo funções do algoritmo BRI, a saber $BRI_get_C_{xy}(k, i)$ e $BRI_get_C_{kk}(k)$, nas linhas 4 e 8. Neste caso, a função $BRI_get_C_{xy}(k, i)$ retorna C_{ki} , que corresponde ao bloco localizado na linha de blocos k e na coluna de blocos i da matriz inversa A_γ^{-1} , conforme Equação (5.10). Por sua vez, a função $BRI_get_C_{kk}(k)$ retorna a inversa do bloco diagonal, C_{kk} , da matriz inversa A_γ^{-1} . Contudo, analisando a Equação (4.9), conclui-se que a obtenção do bloco localizado na posição superior esquerda da matriz inversa em questão, no caso N_{11} , ocorre a partir do cálculo da inversão do último bloco resultante do procedimento recursivo retroativo do BRI. Então, para obter a inversa do bloco C_{kk} , basta o BRI não efetuar a inversão do último bloco. Isso economiza processamento e memória.

Somado a isso, note que a dimensão do bloco C_{kk} é aproximadamente n_v , o qual é, no geral, muito menor que $(n - n_v)$. Então, resolver $\beta_{(k)} = C_{kk}^{-1} \alpha_{(k)}$, no geral, é muito mais simples do que treinar o classificador LS-SVM com base nos $(n - n_k)$ padrões do

conjunto de treinamento. E, finalmente, é válido observar que, para a obtenção da taxa de erro de classificação, o algoritmo proposto não necessita do termo de polarização b e da última linha e última coluna de A_γ^{-1} , em (5.10), conforme Teorema 3.

Todo este procedimento será válido desde que a matriz de kernel K_γ , definida na Equação (5.8), seja positiva definida, conforme condições apresentadas na Subseção 5.1.2.

Algorithm 7 Validação Cruzada Eficiente com BRI

Entrada: l (número de partições), vetor y (resposta real)

1. $n_k \leftarrow \text{size}(y)/l$, $y^{(k)} \leftarrow \text{zeros}(l, n_k)$
2. **for** $k \leftarrow 1, k \leq l$ **do**
3. **for** $i \leftarrow 1, i \leq l$ **do**
4. $C_{ki} \leftarrow \text{BRI_get_C}_{xy}(k, i)$
5. $\alpha_{(k)} \leftarrow \alpha_{(k)} + C_{ki}y_{(i)}$
6. $i \leftarrow i + 1$
7. **end for**
8. $C_{kk}^{-1} \leftarrow \text{BRI_get_C}_{kk}(k)$
9. $\beta_{(k)} \leftarrow C_{kk}^{-1} \alpha_{(k)}$
10. $y^{(k)} \leftarrow \text{sign}[y_{(k)} - \beta_{(k)}]$
11. $k \leftarrow k + 1$
12. **end for**
13. $\text{erro} \leftarrow \frac{1}{2} \sum_{k=1}^l \sum_{i=1}^{n_k} |y_{k,i} - y^{(k,i)}|$

Saída: erro

5.5 Implementação

O algoritmo de validação cruzada l -fold eficiente com o uso limitado de memória, proposto na Seção 5.4, foi implementado em linguagem C++, integrando os benefícios proporcionados pelo algoritmo CV eficiente de [An et al. 2007] com as funcionalidades do BRI. Usou-se a biblioteca de álgebra linear Armadillo [Sanderson & Curtin 2016] para a inversão dos blocos de ordem n_k , no procedimento recursivo retroativo do BRI, durante a obtenção dos blocos C_{ki} e C_{kk} , com $k, i = 1, 2, \dots, l$.

Em se tratando de aspectos técnicos, o algoritmo de validação cruzada aqui proposto foi implementado conforme definições do *toolbox* LS-SVMLab [De Brabanter et al. 2011], especificamente no tocante à função *crossvalidatelssvm*. Este *toolbox* destina-se, especialmente, para uso com o software Matlab® e contém implementações para uma série de algoritmos LS-SVMs relacionados à classificação, regressão, previsão de séries temporais e aprendizagem não-supervisionada [Pelckmans et al. 2002]. Trata-se de um

ambiente de treinamento e simulação de máquinas LS-SVMs, implementado em código C, e pode ser usado em diferentes arquiteturas de computadores, incluindo Linux e Windows. Sua função denominada *crossvalidateLssvm* estima o desempenho do modelo com o algoritmo *fast l-fold crossvalidation*, cuja implementação se baseou em [De Brabanter et al. 2010] que, por sua vez, baseou-se em [An et al. 2007].

Sendo assim, na implementação do Algoritmo 7, em consonância com a função *crossvalidateLssvm* do LS-SVMLab, os dados de entrada $\{x_i, y_i\}_{i=1}^n$ são divididos em l conjuntos disjuntos. Contudo, a estimação do desempenho do modelo que, conforme a técnica tradicional l -fold VC, normalmente seria obtido iterativamente a partir do i -ésimo conjunto deixado de fora (conjunto de validação), no algoritmo proposto é obtido através do Teorema 3.

Assim, as l diferentes estimativas do desempenho do modelo, ditos erros de classificação, são combinadas segundo o cálculo do Erro Quadrático Médio (do inglês, Mean Squared Error – MSE), gerando a taxa de erro de generalização do classificador ou custo. O MSE é definido como sendo a diferença entre o atributo que deve ser estimado e o estimador. Consequentemente, pode ser chamado de uma função de risco que corresponde ao valor esperado do quadrado dos erros. O erro, neste caso, é a quantia que o estimador difere do valor observado [Wang & Bovik 2009]. Assim, considerando $y^{(i)}$ como sendo o vetor dos valores estimados e $y_{(i)}$ o vetor dos verdadeiros valores que estão sendo esperados nas classificações das n amostras de entrada, com $i = 1, 2, \dots, n$, têm-se que:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_{(i)} - y^{(i)})^2. \quad (5.16)$$

Como no processo de validação cruzada l -fold de LS-SVMs, a matriz a ser invertida, A_γ , pode ser obtida bloco por bloco, conforme discutida na Subseção 5.4.2, não é necessário armazená-la, por inteiro, na memória. Usando o algoritmo BRI, a matriz A_γ será calculada, bloco por bloco, a partir de uma função aplicada à matriz de amostras de treinamento de entrada, $x_i \in \mathbb{R}^n$ que, embora necessite ser armazenada, é frequentemente muitas ordens de magnitude menor do que armazenar A_γ e/ou, até mesmo, K_γ . Desta maneira, e devido a estrutura recursiva do algoritmo BRI, serão armazenados apenas alguns blocos C_{ki} , cada um apresentando dimensão n_v , com $n_k \approx n_v$ e $n_v = \frac{n}{l}$, de acordo com a necessidade para obter $\alpha_{(k)}$ e $\beta_{(k)}$.

Observe que a dimensão de cada bloco diagonal principal C_{kk} é, aproximadamente, n_v , o qual é, no geral, muito menor que $(n - n_v)$. Consequentemente, estimar as classificações esperadas pelas partições reservadas para teste ($\beta_{(k)}$) através de $\beta_{(k)} = C_{kk}^{-1} \alpha_{(k)}$ é, no geral, muito mais simples do que treinar o classificador LS-SVM com base nos $(n - n_k)$ padrões

do conjunto de treinamento. Assim, $\beta_{(k)}$ será utilizado para obter $y^{(i)}$ que, por sua vez, será utilizado para calcular o Erro Quadrático Médio, conforme (5.16).

O uso particionado da matriz \mathbf{C} , proporcionado pelo Algoritmo VC eficiente 6, associado aos benefícios de consumo limitado de memória do algoritmo BRI, permite a estimativa da capacidade de generalização de um modelo a partir do treinamento com conjuntos de entrada $\{x|x \in \mathbb{R}^n\}$ de, praticamente, qualquer tamanho.

5.6 Análise de Complexidade

A análise de complexidade do algoritmo de validação cruzada eficiente com o uso limitado de memória, proporcionado pelo BRI, pode ser desenvolvida com base na análise de complexidade descrita na Seção 4.4. Para tanto, considere os seguintes fatos básicos:

- (a) O custo computacional do algoritmo BRI para obter um único bloco C_{ki} , de ordem n_k , da matriz em blocos $l \times l$, \mathbf{C} , a partir A_γ^{-1} , é $\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i} + n_k^3$, para $k = l$ e $b = n_k$, conforme analisado na Seção 4.4.
- (b) O custo computacional da inversa X^{-1} de um bloco X de ordem n_k é $O(n_k^3)$;
- (c) A inversa da inversa de uma matriz é igual à própria matriz $X = (X^{-1})^{-1}$.

Teorema 5. *Dado n conjuntos de treinamentos (x_i, y_i) , com $x_i \in \mathbb{R}^n$ e rótulos de classes $y_i \in \{-1, +1\}$. Então, a complexidade $C_{crossBRI}$, do algoritmo de validação cruzada l -fold eficiente usando BRI é:*

$$C_{crossBRI}(l, n_k) = O(4^{l-1} n_k^3 l^2). \quad (5.17)$$

Demonstração. De acordo com a ideia básica do processo validação cruzada l -fold usando BRI, apresentado no Algoritmo 7, na estrutura de repetição descrita nas linhas 3-7, são obtidos l blocos $n_k \times n_k$ de uma linha de blocos k da matriz \mathbf{C} , que corresponde a blocos da matriz inversa de A_γ , para obter $\alpha_{(k)}$. Assim, considerando a propriedade (a), neste laço, são efetuadas l operações $\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i} + n_k^3$. Soma-se a esta equação mais 2 (dois), correspondente ao desempenho das iterações das duas operações aritméticas contidas nas Linhas 5 e 6. Resultando, então, em um desempenho de $\left(\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i} + n_k^3 + 2 \right) l$.

Posteriormente, nas linhas 8-11, faz-se necessário obter a inversa do bloco C_{kk} para calcular $\beta_{(k)}$. Contudo, conforme a Equação (4.9), no último passo do algoritmo BRI, Passo $k - 1$, é preciso inverter a última matriz resultante do processo recursivo retroativo para obter o respectivo bloco da matriz inversa, M^{-1} . Posto isto, e considerando o disposto na propriedade (c), foi excluída a última inversa retornada pela função $BRI_get_C_{kk}(k)$.

Assim, conforme a propriedade (b), nesta etapa do algoritmo, são necessárias mais $\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i}$ operações. Soma-se a esta equação mais 3 (três), correspondente ao desempenho das iterações das três operações aritméticas contidas nas Linhas 9-11. Resultando, então, em um desempenho de $\left(\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i} + 3\right)$.

Feito isso, são necessárias mais l iterações, acima descritas, para obter o vetor $\beta = [\beta_{(1)}, \beta_{(2)}, \dots, \beta_{(l)}]^T$ inteiro. Portanto, o algoritmo proposto possui uma complexidade de

$$C_{crossBRI}(l, n_k) = \left[\left(\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i} + n_k^3 + 2 \right) l + \left(\sum_{i=1}^{l-1} 3n_k^3 4^{l-1-i} + 3 \right) \right] l = O(4^{l-1} n_k^3 l^2).$$

□

5.6.1 Análise do Custo de Memória

Para as análises seguintes, considere $n_k \approx n_v \approx \frac{n}{l}$, conforme apresentado na Subseção 5.4.1. De acordo com a implementação proposta do BRI na Seção 4.3, é possível armazenar apenas alguns blocos $n_k \times n_k$ para efetuar o cálculo inteiro devido à estrutura recursiva do algoritmo. Sendo assim, chega-se ao seguinte teorema:

Teorema 6. *Seja a matriz em blocos $C \in \mathbb{R}^{n \times n}$, com blocos C_{ki} de ordem $n_k \times n_k$, com $k, i = 1, 2, \dots, l$. Sendo $\alpha_{(k)} = \sum_{i=1}^l C_{ki} y_{(i)}$ e $\beta_{(k)} = C_{kk}^{-1} \alpha_{(k)}$. Então, o custo de memória $CM_{crossBRI}$ do algoritmo de validação cruzada l -fold eficiente usando BRI é:*

$$CM_{crossBRI} = O(n_k^2). \quad (5.18)$$

Demonstração. Observe que, pelos Teoremas 3 e 4, não é necessário armazenar a matriz C na memória principal. Além disso, considerando o Algoritmo 7, os blocos C_{ki} e C_{kk} são calculados iterativa e sequencialmente para obter $\alpha_{(k)}$ e $\beta_{(k)}$.

Sob esses aspectos e considerando que, no algoritmo BRI, apresentado na Seção 4.1, cada bloco de C é acessado apenas para aplicar as operações do complemento de Schur. Então, o algoritmo de validação cruzada l -fold eficiente usando BRI apresentará o mesmo custo de memória definido pelo algoritmo BRI, conforme demonstração do Teorema 2, a saber:

$$MC_{crossBRI} = 3(n_k)^2 + (n_k)^2 = O(n_k^2).$$

□

Vale ressaltar que a validação cruzada l -fold ingênua (do inglês, *naive*) apresenta com-

plexidade de [An et al. 2007]:

$$C_{crossNaive}(l, n_k) = \left(\frac{(l-1)^3}{3l^2} \right) n^3. \quad (5.19)$$

Seja $n = ln_k$. Então:

$$C_{crossNaive}(l, n_k) = O(l^4 n_k^3). \quad (5.20)$$

E a validação cruzada l -fold eficiente, apresentada no Algoritmo 6, possui complexidade de [An et al. 2007]:

$$C_{fastCross}(l, n_k) = \left(\frac{1+l}{3l^2} \right) n^3. \quad (5.21)$$

Sendo $n = ln_k$. Resulta em:

$$C_{fastCross}(l, n_k) = O(l^2 n_k^3). \quad (5.22)$$

E ambas as técnicas apresentam custo de memória $CM_{crossNaive}$ e $CM_{fastCross}$ de ordem:

$$MC_{crossNaive}(l, n_k) = MC_{fastCross}(l, n_k) = O(l^3 n_k^3) \quad (5.23)$$

tendo em vista que precisam inverter a matriz A_γ e K_γ , respectivamente.

Consequentemente, os métodos de validação cruzada l -fold *naive* e o eficiente são mais rápidos que Algoritmo 7. Inclusive, é oportuno observar que, conforme Equação (5.21), na medida em que a quantidade de partições, l , aumenta, consequentemente, diminui n_k , diminui a complexidade do algoritmo e, portanto, sua execução se torna mais rápida. Por outro lado, o custo da memória do algoritmo proposto é muito menor do que ambos algoritmos, conforme as Equações (5.18) e (5.23). Este fato é verificado pelos resultados numéricos apresentados na Seção 5.7. Todavia, as comparações entre os algoritmos de validação cruzada l -fold *naive* e o eficiente encontram-se em [An et al. 2007].

5.7 Resultados Experimentais

Para fins de comparação, foram realizados experimentos com o Algoritmo 7 e o Algoritmo 6 medindo tempo de execução e uso de memória. A implementação do algoritmo l -fold VC eficiente com BRI fez uso das chamadas de funções $BRI_get_C_{xy}(k, i)$ e $BRI_get_C_{kk}(k)$ do BRI, conforme Linhas 4 e 8 do Algoritmo 7. Já o l -fold VC eficiente tradicional foi implementado usando a chamada de função da biblioteca Armadillo $inv(\cdot)$

para obter a inversa da matriz K_γ , na linha 1, e a chamada de função `solve($C_{kk}, \alpha_{(k)}$)`, também do Armadillo, para resolver os l sistemas lineares $C_{kk}\beta_{(k)} = \alpha_{(k)}$, conforme Linha 6 do Algoritmo 6. Conforme exposto na Seção 4.5, a função `inv(\cdot)` obtém a inversa de uma matriz quadrada geral por fatoração LU usando pivoteamento parcial com permutações de linhas.

Foram considerados vetores x com n dados de entrada para treinamento, compostos de entradas aleatórias, escolhidas de uma distribuição normal com média 0,0 e desvio padrão 1,0 gerada pelo comando `randn()` da biblioteca Armadillo.

Dada o vetor x , é possível calcular a matriz A_γ^{-1} , usada nos métodos de validação cruzada l -fold para LS-SVM analisados, sendo:

$$A_{\gamma(n+1) \times (n+1)} = \begin{bmatrix} K_\gamma & \mathbf{1}_n \\ \mathbf{1}_n^T & 0 \end{bmatrix} \quad (5.24)$$

com $\mathbf{1}_n = [1, 1, \dots, 1]^T$, $K_\gamma = K + \frac{1}{\gamma}I_n$ e $K_{i,j} = K(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$.

A função $K(\cdot, \cdot)$ é o *kernel* do tipo Gaussiano ou Função de Base Radial (RBF, do inglês *Radial Basis Function*) e $\{x_i\}_{i=1}^n$, o conjunto de dados de entrada.

As simulações foram realizadas com matrizes A_γ variando as quantidades de blocos $(l \times l)$ para ambos os algoritmos analisados, gerando blocos C_{ki} e C_{kk} de ordem $n_k = \frac{n}{l}$.

Todos os testes foram realizados em nós computacionais do supercomputador do Núcleo de Processamento de Alto Desempenho (NPAD) da Universidade Federal do Rio Grande do Norte¹. Cada nó possui a seguinte configuração: 2 CPUs Intel Xeon Sixteen-Core E5-2698v3 de 2.3 GHz com 128 GB de RAM.

Esta seção descreve os experimentos, bem como os resultados obtidos. Na Subseção 5.7.1, são apresentadas as medidas do valor máximo de memória alocada durante o processamento do Algoritmo 7 e do Algoritmo 6. E, na Subseção 5.7.2, os resultados em tempo de execução são analisados.

5.7.1 Uso de Memória

As medições do uso de memória foram obtidas do programa Syrupy². O Syrupy é um script escrito em Python que captura, em intervalos regulares, instantâneos (do inglês, *snapshots*) do uso de memória e CPU de um ou mais processos em execução, de modo a criar dinamicamente um perfil de uso dos recursos do sistema. Esse monitoramento de uso de recursos do sistema é baseado em chamadas repetidas ao comando do sistema `ps`.

¹Site oficial do NPAD: <http://npad.ufrn.br/>

²Para mais informações e/ou download do Syrupy, acesse <https://github.com/jeetsukumaran/Syrupy>

Para análises comparativas do pico da memória consumida pelos algoritmos testados, foram consideradas as medições da saída do Syrupy denominada `VSIZE`, ou tamanho da memória virtual, que corresponde à quantidade total de memória que o processo está usando (em kiloBytes), no momento em que foi capturado o *snapshot*. Neste caso, é considerado tanto a quantidade de memória RAM alocada para o processo (RSS - do inglês, *Resident Set Size*), bem como o valor consumido do espaço de troca (*swap*). Assim, foram obtidos o valor máximo do `VSIZE` para cada iteração dos algoritmos, aqui denominada, pico do uso de memória.

O consumo de memória durante a execução do algoritmo l -fold VC eficiente com BRI (aqui representado pelo termo `crossBRI`) e do l -fold VC eficiente tradicional, com uso da inversa por fatoração LU (representado aqui pelo termo `crossLU`) para obter a inversa da matriz K_γ , em relação à quantidade de blocos ($l \times l$), é mostrado na Figura 5.4.

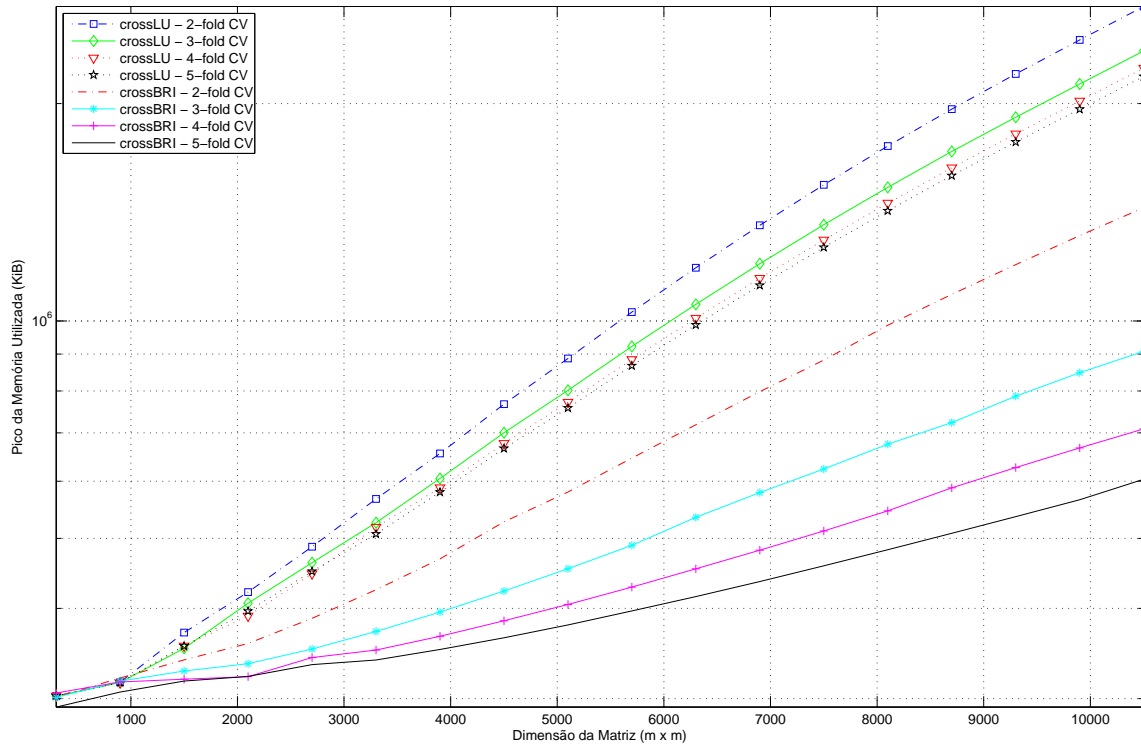


Figura 5.4: Comparação do pico do uso de memória na validação cruzada de LS-SVM com o algoritmo BRI e com a inversão LU, para diferentes l partições.

O algoritmo l -fold VC eficiente com BRI (`crossBRI`) consome, claramente, menos memória do que o l -fold VC eficiente tradicional (`crossLU`). Considerando uma matriz de entrada de dimensão $m \times m$, à medida que o número partições, l -folds, aumenta, a alocação de memória diminui. Isso acontece devido a dimensão $n_k \times n_k$ dos blocos diminuir, onde $n_k = \frac{m}{l}$, em relação a (3.5). Consequentemente, menos dados são mantidos na

memória durante todo o processo de medição da taxa de erro de generalização, através do método de validação cruzada l -folds. Assim, a aplicação da inversão recursiva nos permite considerar conjuntos de dados de treinamento x , de tamanho n , muito maiores do que aquelas que a técnica l -fold VC eficiente tradicional permite.

5.7.2 Tempo de Execução

A Figura 5.5 exibe os tempos de processamento obtidos na inversão de matrizes $m \times m$ com diferentes que o número partições, l -folds, usando os algoritmos l -fold VC eficiente com BRI (crossBRI) e l -fold VC eficiente tradicional, usando LU (crossLU). Na referida figura, a plotagem dos gráficos mostra que o tempo de processamento do crossBRI é maior do que o do crossLU, aumentando à medida que aumentamos a quantidade de blocos (l). Durante os experimentos, observou-se que os cálculos do algoritmo crossLU diminuem suavemente com o aumento de l , refletindo, assim, no tempo de execução. Com o uso do BRI para obter os blocos da matriz inversa, esta característica do algoritmo de [An et al. 2007] é contrabalanceada com o aumento da recursividade proporcionada pelo aumento das partições, l .

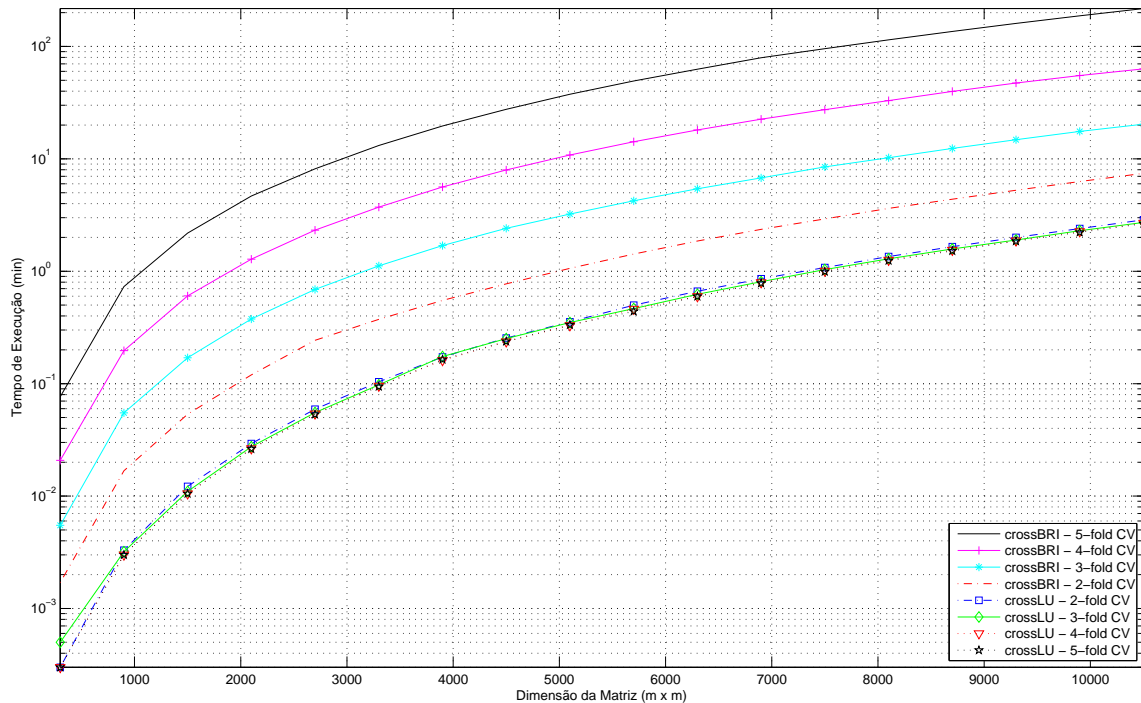


Figura 5.5: Comparação do tempo de execução da validação cruzada de LS-SVM eficiente utilizando o algoritmo BRI e a inversão LU, para diferentes l partições.

Porém, é interessante notar que é possível compensar um aumento no tempo de pro-

cessamento pelo consumo de menos memória ao variar a quantidade de partições l , durante o cálculo da taxa de erro de generalização e, consequentemente, permitir a inversão de matrizes muito grandes que, de outra forma, não se encaixariam na memória.

Capítulo 6

BRI Paralelo para Arquiteturas de Memória Compartilhada

A característica principal do algoritmo BRI consiste em, durante todo o processo de inversão matricial, trabalhar apenas com alguns poucos blocos da matriz, limitando assim o uso de memória do sistema computacional. Isso é desejável nos casos em que a matriz a ser invertida não cabe na memória disponível, mas que é possível abrir mão de tempo de processamento.

Entretanto, buscando-se ampliar sua aplicação para os casos em que há restrições de tempo, é passível cogitar a possibilidade de fazer algumas otimizações na implementação do mesmo, considerando o fato de que alguns blocos que são invertidos, no procedimento recursivo retroativo, possam ser aproveitados em outros ramos.

Além disso, a programação paralela pode ser aplicada para melhorar o seu desempenho. De acordo com García-López et al. (2002), a paralelização consiste em executar cálculos mais rápidos usando vários processadores simultaneamente, aumentando assim o desempenho do programa. Neste caso, é possível utilizar processadores individuais para efetuar os cálculos com matrizes menores (no caso, blocos) a fim de, posteriormente, combinar os resultados. Isso, além de claramente reduzir o tempo de execução do algoritmo, possibilita efetuar os cálculos aritméticos necessários trabalhando com matrizes menores na memória de alta velocidade do sistema computacional, cuja capacidade pode não ser suficientemente grande para acomodar toda a matriz de entrada [Anton & Busby 2006]. Sendo assim, o uso de técnicas de paralelização no BRI resulta na possibilidade de se alcançar velocidade adicional no processo de inversão matricial como um todo.

Neste capítulo são apresentadas as pesquisas e experimentos realizados, publicados em [Silva et al. 2017]. Desta forma, encontra-se organizado como segue. Na Seção 6.1, são introduzidos conceitos básicos sobre a paralelização de algoritmos. A Seção 6.2 apresenta alguns trabalhos relacionados. A otimização que foi realizada no algoritmo

BRI está descrita e analisada na Seção 6.3. A sua paralelização, bem como os resultados experimentais obtidos encontram-se descritos na Seção 6.4.

6.1 Programação Paralela

A programação paralela busca resolver problemas complexos e sobre grandes quantidades de dados utilizando de forma eficiente os recursos computacionais disponíveis atualmente em sistemas com múltiplos elementos de processamento [Silva 2011]. Contudo, a popularização dos sistemas com múltiplos elementos de processamento ocorreu apenas no início do século 21 motivada, principalmente, pela saturação dos recursos tecnológicos utilizados em processadores com apenas um núcleo de processamento (ditos, monoprocessadores), que se tornaram uma barreira para suprir a demanda de processamento sobre volumes de dados cada vez maiores.

Em 1966, Flynn propôs um modelo para dividir computadores em categorias bem abrangentes. Este modelo ainda se apresenta útil, pois estabelece classes de computadores de acordo com sua capacidade de paralelizar instruções e dados. Conforme este modelo, os sistemas computacionais são categorizados como se segue [Stallings 2010].

- Instrução única, único dado (SISD, do inglês, *single instruction, single data*): um único processador executa uma única sequência de instruções para operar nos dados armazenados em uma única memória.
- Instrução única, múltiplos dados (SIMD, do inglês, *single instruction, multiple data*): uma mesma instrução é executada por vários processadores utilizando diferentes fluxos de dados. Cada processador tem sua própria memória de dados.
- Múltiplas instruções, único dado (MISD, do inglês, *multiple instruction, single data*): uma sequência de dados é transmitida para um conjunto de processadores, onde cada um executa uma sequência de instruções diferente. Nenhum multiprocessador desse tipo foi implementado comercialmente.
- Múltiplas instruções, múltiplos dados (MIMD, do inglês, *multiple instruction, multiple data*): Cada processador busca sua própria instrução e opera sobre seus próprios dados. Computadores MIMD exploram paralelismo em nível de *threads* e de processos, possibilitando que múltiplas unidades de execução sejam processadas paralelamente.

A única categoria que interessa nesta pesquisa, dentre as citadas acima, é a MIMD. Processadores MIMD são divididos em dois grupos, que são determinados de acordo com

a organização da memória de trabalho e a maneira como os diferentes elementos de processamento estão interconectados. Estes grupos são classificados como Arquiteturas com Memória Compartilhada e Arquiteturas com Memória Distribuída, conforme ilustrados nas Figuras 6.1 e 6.2, respectivamente.

Na arquitetura com memória compartilhada, os núcleos são conectados à memória por meio do barramento, e cada núcleo pode acessar qualquer localização da memória. Enquanto que, na arquitetura com memória distribuída, os núcleos formam pares com sua própria memória, e cada par de núcleo-memória se comunica com outro par pelo barramento.

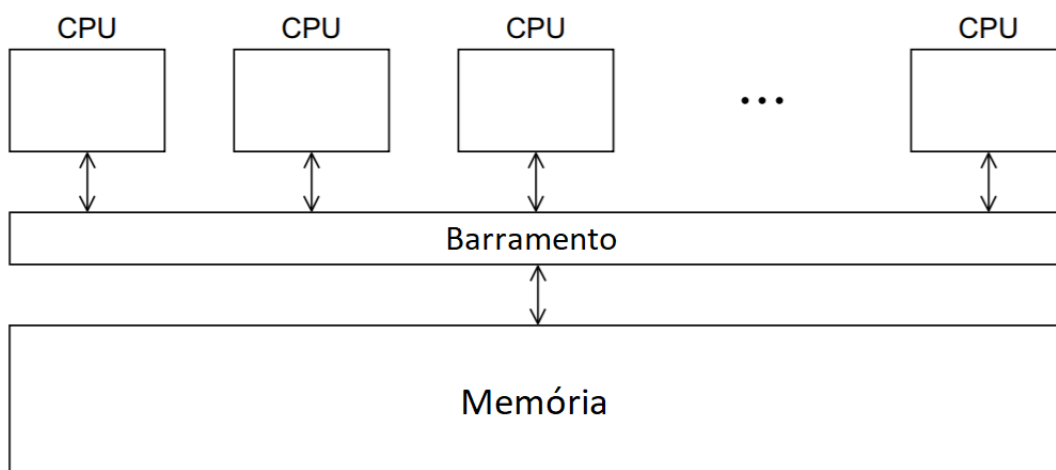


Figura 6.1: Arquitetura com memória compartilhada Fonte [Pacheco 2011].

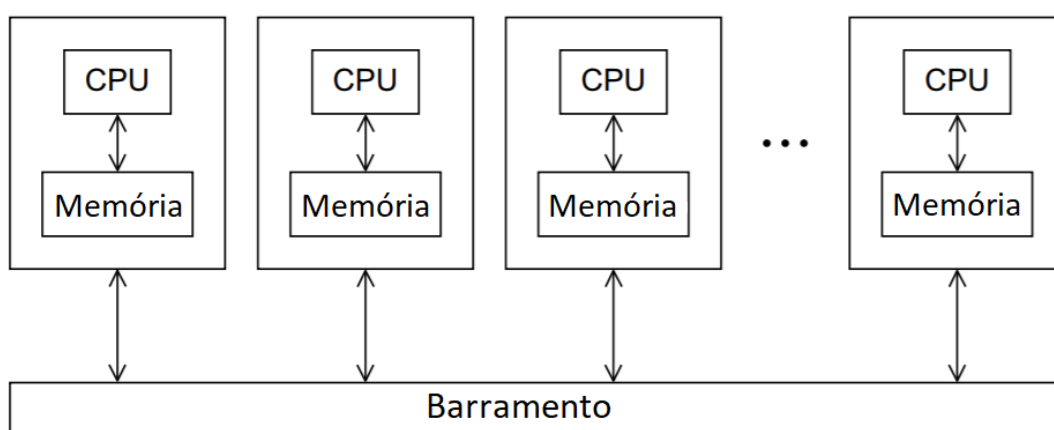


Figura 6.2: Arquitetura com memória distribuída [Pacheco 2011].

Com a evolução e popularização das arquiteturas paralelas, torna-se cada vez mais

importante o estudo e a aplicação de técnicas para implementar algoritmos concorrentes sobre tais máquinas, para que os múltiplos elementos de processamento disponíveis sejam utilizados de maneira eficiente.

A programação paralela trata da implementação de algoritmos concorrentes, utilizando recursos que permitem expressar o paralelismo das operações e incluir mecanismos de sincronização e comunicação entre diferentes unidades de execução [Silva 2011].

Programas paralelos podem apresentar diferentes formas de paralelismo, de acordo com a forma com que realizam operações simultâneas. As categorias mais abordadas são o paralelismo de dados e o paralelismo de tarefas.

O paralelismo de dados ocorre quando uma operação é aplicada sobre múltiplos elementos de uma estrutura de dados de forma paralela. Enquanto que, o paralelismo de tarefas é utilizado para expressar paralelismo de instâncias de conjuntos de instruções (tarefas) em um programa. Por ter este caráter genérico, este tipo de paralelismo pode ser utilizado para paralelizar tarefas cujo número de execuções não pode ser previsto estatisticamente, como acontece em programas recursivos. Por este motivo, buscou-se paralelizar o algoritmo BRI utilizando esta abordagem.

6.1.1 OpenMP

OpenMP é uma interface de programação (API, do inglês, Application Programming Interface) que suporta programação paralela em ambientes de memória compartilhada nas linguagens C/C++ e Fortran, desenvolvida para facilitar a implementação de programas paralelos em ambientes com memória compartilhada [Chapman et al. 2008].

Esta API especifica um conjunto de diretivas para o compilador e rotinas a serem chamadas em tempo de execução, com o objetivo de expressar paralelismo em arquiteturas com memória compartilhada. Também define variáveis de ambiente a serem utilizadas pelas rotinas. Sendo que, inicialmente, não existia suporte a paralelismo de tarefas, que foi introduzido apenas em 2008, na versão 3.0.

OpenMP utiliza o modelo de execução paralela *fork-join*. Neste caso, um programa OpenMP inicia como uma única unidade de execução, chamada de *thread* inicial. Esta *thread* executa dentro de uma região sequencial, que é chamada de região da tarefa inicial. A principal diretiva OpenMP, utilizada em todos os programas que fazem uso da API, é a diretiva *parallel*. Ela é responsável por demarcar as regiões do código a serem executadas por múltiplas *thread* (regiões paralelas).

Para o OpenMP, uma tarefa é uma instância específica de código executável e seu ambiente de dados, gerada quando uma *thread* encontra uma diretiva *task*. Quando uma

thread encontra uma diretiva *task*, uma nova tarefa é gerada, e sua execução é atribuída a uma das *threads* do grupo atual. Como a execução da tarefa depende da disponibilidade da *thread* para realizar o trabalho, a execução de uma nova tarefa pode ser imediata ou postergada.

6.1.2 Análise de Desempenho de Algoritmos Paralelos

Nesta tese, a avaliação de desempenho da versão paralela do algoritmo BRI levará em consideração o *speedup* e a eficiência.

Speedup

O *speedup* (S) é uma métrica que representa a razão entre o tempo de execução de um programa paralelo e o tempo de execução de sua versão sequencial. Portanto, trata-se de uma boa medida que serve para avaliar quantitativamente a melhoria de desempenho trazida pela versão paralela de um programa em relação à sua versão sequencial [Pacheco 2011]. A fórmula do *speedup* é definida pela seguinte equação:

$$S = \frac{T_{serial}}{T_{paralelo}}, \quad (6.1)$$

onde T_{serial} é o tempo de execução do código sequencial e $T_{paralelo}$ o tempo de execução da versão paralela.

No caso ideal, conhecido por *speedup* linear, S é igual a quantidade de processadores. Todavia, na prática é pouco provável alcançar o *speedup* linear, pois o uso de *threads* por processos inserem sobrecusto ao programa. O sobrecusto de *threads* é definido como sendo o custo imposto ao sistema operacional para gerenciar (criar, escalonar e efetuar trocas de contexto) o grupo de *threads* utilizados pelo programa (cujo tamanho é definido pela variável de ambiente OMP_NUM_THREADS).

Eficiência

Outra forma de medir o desempenho do programa é através da eficiência (E), que analisa a interferência do aumento de processadores no desempenho do programa [Pacheco 2011]. A eficiência é definida como segue:

$$E = \frac{T_{serial}}{T_{paralelo} \cdot p}, \quad (6.2)$$

onde p é a quantidade de núcleos de processamento (*core*) utilizados na execução do programa.

6.2 Trabalhos Relacionados

A paralelização de algoritmos recursivos tem sido amplamente discutida no meio científico. Em [Jonsson & Kågström 2002], algoritmos recursivos são propostos para resolver equações de matrizes triangulares com implementação paralela, usando chamadas de subprogramas recursivos, alocação de memória dinâmica e *threads*.

Em outra perspectiva, Heinecke & Bader (2008), Chatterjee et al. (2002) e Buluç et al. (2009) trataram da paralelização de algoritmos recursivos para a multiplicação de matrizes sob abordagens distintas. De outro modo, Drouvelis et al. (2006) propuseram uma implementação paralela do método recursivo da função de Green.

Com uma visão mais ampla, Ghiya et al. (1998) fizeram uma análise sobre os principais padrões detectados na aplicação de paralelismo em estruturas de dados recursivas.

Especificamente, tratando o tema de inversão de matrizes de ordem extremamente alta, Lau et al. (1996) apresentaram dois algoritmos baseados na eliminação de Gauss para inverter matrizes definida positiva e simétrica esparsas em computadores paralelos, e os implementou em computadores SIMD e MIMD. Já Bientinesi et al. (2008) introduziram um algoritmo recursivo para inverter uma matriz definida positiva simétrica baseada na fatoração de Cholesky em arquiteturas paralelas tanto de memória compartilhada quanto distribuída.

Mahfoudhi et al. (2017) projetaram dois algoritmos baseados na fatoração LU recursiva e no método de Strassen para inverter matrizes quadradas densas em uma arquitetura de computadores com processadores *do inglês, multicore*.

De outro modo, em [Ezzatti et al. 2011], os autores implementaram um algoritmo de inversão de matrizes em blocos baseado na eliminação de Gauss-Jordan em uma arquitetura híbrida consistindo em um (ou mais) processadores *multicore* conectados a várias GPUs (Unidades de Processamento Gráfico).

Todos esses trabalhos mostram a importância e o impacto da paralelização em algoritmos recursivos.

6.3 Otimização do Algoritmo BRI

Uma vez que o algoritmo BRI efetua permutações de linhas e colunas para que o bloco M_{22} atinja a posição da segunda linha de blocos e segunda coluna de blocos dos quadros resultantes, todos os nós folhas da árvore de recursão progressiva contêm este bloco.

Especificamente, analisando o procedimento recursivo retroativo, é possível perceber que, durante a obtenção dos complementos de Schur (M/D) , (M/C) , (M/B) e (M/A) (Passo 1), a operação de inversa ($\text{inv}(\cdot)$) será justamente aplicada ao bloco M_{22} nos nós folhas.

Considerando uma matriz de entrada particionada em $k \times k$ blocos, existirão 4^{k-2} nós folhas. Portanto, será necessário processar o mesmo cálculo de inversão em M_{22} (4^{k-2}) vezes ao se atingir o último nível da recursividade progressiva. A eliminação dessas inversões matriciais economiza uma quantidade considerável de recursos computacionais e tempo.

Sendo assim, uma forma de otimizar o algoritmo BRI se constitui em calcular a inversa do referido bloco uma única vez e armazená-la em uma variável global, de modo que cada folha possa acessar a mesma variável ao invés de efetuar o cálculo da inversa desse bloco inúmeras vezes.

O Algoritmo 8 apresenta o pseudocódigo simplificado desta pequena modificação implementada no módulo principal do algoritmo BRI. Neste caso, a variável global denominada *invFolha* armazenará a inversa do bloco M_{22} que será utilizada pelas funções recursivas BRI_A , BRI_B , BRI_C e BRI_D .

O Algoritmo 9 apresenta o código modificado do Algoritmo 2. Observe que a única alteração sofrida foi que, no bloco de instruções responsável por calcular o complemento de Schur nas folhas da árvore de recursão (Linhas 8 e 9), a chamada à função $\text{inv}(\cdot)$ foi substituída pela matriz armazenada em *invFolha*. A mesma alteração foi realizada nos Algoritmos 3, 4 e 5 que implementam as funções BRI_B , BRI_C e BRI_D .

Algorithm 8 Algoritmo de Inversão Recursiva de Blocos Otimizado

Entrada: k (quantidade de blocos por linha), b (ordem do bloco), $M = (M_{\alpha\beta})_{1 \leq \alpha, \beta \leq k}$ (matriz de entrada)

1. $\text{vet} : \text{linha}[1..k]$ de inteiro
2. $\text{vet} : \text{coluna}[1..k]$ de inteiro
3. $\text{invFolha} = \text{inv}(M[2][2])$
4. $N_{11} = \text{inv}(BRI_A(b, \text{linha}, \text{coluna}))$

Saída: N_{11}

Algorithm 9 Função recursiva BRI_A modificada

Entrada: k (quantidade de blocos por linha), $coluna$ (vetor coluna de blocos), $linha$ (vetor linha de blocos)

```

1. function  $BRI_A(k, linha, coluna)$ 
2.   if  $k > 2$  then
3.      $M_A \leftarrow BRI_A(k-1, linha[1:k-1], coluna[1:k-1])$ 
4.      $M_B \leftarrow BRI_B(k-1, linha[1:k-1], coluna[2:k])$ 
5.      $M_C \leftarrow BRI_C(k-1, linha[2:k], coluna[1:k-1])$ 
6.      $M_D \leftarrow BRI_D(k-1, linha[2:k], coluna[2:k])$ 
7.      $(M/D) \leftarrow M_A - M_B * inv(M_D) * M_C$ 
8.   else
9.      $(M/D) \leftarrow A - B * invFolha * C$ 
10.  end if
11. end function

```

Saída: (M/D)

6.3.1 Resultados Experimentais

Para fins de comparação, foram realizados experimentos com o Algoritmo BRI original, apresentado no Capítulo 4 e a versão otimizada do mesmo, conforme analisado na Seção 6.3, medindo tempo de processamento.

Todos os testes foram realizados em nós computacionais do supercomputador do Núcleo de Processamento de Alto Desempenho (NPAD) da Universidade Federal do Rio Grande do Norte. Cada nó possui a seguinte configuração: 2 CPUs Intel Xeon Sixteen-Core E5-2698v3 de 2.3 GHz com 128 GB de RAM.

A Figura 6.3 exibe os tempos de processamento obtidos nos experimentos realizados entre a versão original do Algoritmo BRI (denominado, algoritmo original) e a versão otimizada (denominada, algoritmo aprimorado) para a inversão de matrizes $m \times m$ variando a quantidades de blocos ($k \times k$).

Como pode ser observado na Figura 6.3, mantendo a quantidade de blocos k constante, a diferença entre o tempo de execução do algoritmo BRI e o de sua versão otimizada tende a crescer conforme o tamanho da matriz de entrada $m \times m$ aumenta. Isso indica um melhor aproveitamento de processamento proporcionado pelo algoritmo otimizado, em comparação ao algoritmo original, nos casos em que o consumo de tempo de processamento é maior.

Além disso, observa-se que as distâncias entre as curvas dos gráficos aumenta à medida que k aumenta. Isso ocorre, pois o aumento de k significa que a matriz de entrada $m \times m$ se tornou mais particionada e, conseqüentemente, haverá mais folhas na

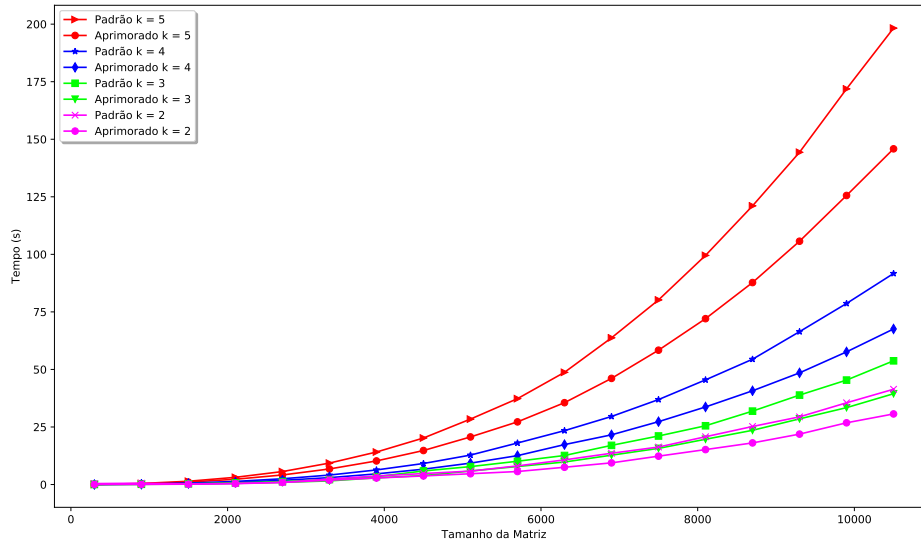


Figura 6.3: Tempo de execução da computação da inversa de matrizes $m \times m$ usando o algoritmo BRI original e uma versão otimizada.

árvore de recursão durante o procedimento de recursividade retroativa. Desta forma, serão $(4^{k-2} - 1)$ inversões de matriz em blocos 1×1 , M_{22} que o algoritmo otimizado não precisará calcular.

Considerando esses benefícios, anteriormente apontados, optou-se por utilizar a versão otimizada do algoritmo BRI durante o processo de paralelização.

6.4 Paralelização do Algoritmo BRI

Analisando a Figura 6.4, observa-se que, no procedimento recursivo progressivo, as gerações de quadros em um mesmo nível i de recursão se constituem em atividades completamente independentes. Portanto, é possível perceber que o algoritmo BRI possui ramos que são horizontalmente independentes uns dos outros, podendo, então, serem processados por diferentes *threads*. Sendo assim, o processamento desses ramos poderá ocorrer simultaneamente e o tempo utilizado em percorrer a árvore de chamadas recursivas, consequentemente, será reduzido consideravelmente.

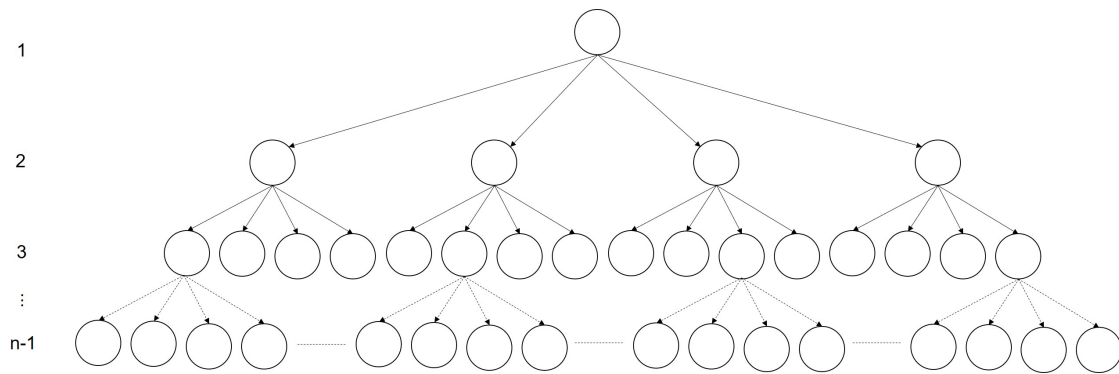


Figura 6.4: Ilustração da árvore de chamadas recursivas do algoritmo BRI.

Para realizar a paralelização do algoritmo BRI, utilizou-se o paralelismo de tarefas OpenMP. Desta forma, diretivas *task* foram implementadas para as chamadas às funções recursivas BRI_A , BRI_B , BRI_C e BRI_D . Esta diretiva permite definir tarefas cuja a execução pode ser feita de forma assíncrona. A utilização de tarefas no algoritmo é justificada por alguns fatores discutidos na sequência.

Definir blocos de código como tarefas possibilita que eles sejam executados em paralelo por uma única *thread*, cada. Neste caso, as *threads* podem ser criadas uma única vez e reaproveitadas para executarem diferentes tarefas. A criação e destruição de *threads* geram um sobrecusto maior do que a criação e destruição de tarefas.

Além disso, o uso de tarefas possibilita uma melhor distribuição de trabalho entre as *threads*. Isso ocorre porque as tarefas são colocadas em filas, aguardando serem executadas por *threads* disponíveis. Isso evita que algumas *threads* estejam trabalhando enquanto outras estejam ociosas. O Algoritmo 10 apresenta o pseudocódigo da função BRI_A paralelizada com as diretivas *task* para cada chamada às funções recursivas BRI_A , BRI_B , BRI_C e BRI_D . Assim, nesta implementação, uma nova tarefa foi associada a cada nó da árvore de chamadas recursivas a ser executada de acordo com a disponibilidade das *threads*.

Na função principal do algoritmo BRI é criada a região paralela onde a função BRI_A será executada, iniciando o processo de recursividade. Porém, como é desejável que apenas uma *thread* faça a chamada inicial da função, devemos utilizar a diretiva *single* antes da primeira chamada. A cláusula *nowait* faz com que as demais *threads* do grupo não aguardem o fim da região delimitada pela diretiva *single*, permitindo que as mesmas participem da execução de BRI_A , possibilitando o paralelismo das tarefas.

Quando uma *thread* encontra uma diretiva *task*, uma nova tarefa é gerada, e sua execução é atribuída a uma das *threads* do grupo atual. Desta forma, como podemos observar no Algoritmo 10, cada tarefa irá gerar quatro tarefas filhas a fim de obter os 4 quadros (a saber, M_A , M_B , M_D e M_C) que, posteriormente, serão utilizados no

Algorithm 10 Função recursiva BRI_A paralelizada

Entrada: k (quantidade de blocos por linha), $coluna$ (vetor coluna de blocos), $linha$ (vetor linha de blocos)

```

1. function  $BRI_A(k, linha, coluna)$ 
2.   if  $k > 2$  then
3.     # pragma omp task shared ( $M_A$ )
4.      $M_A \leftarrow BRI_A(k-1, linha[1:k-1], coluna[1:k-1])$ 
5.     # pragma omp task shared ( $M_B$ )
6.      $M_B \leftarrow BRI_B(k-1, linha[1:k-1], coluna[2:k])$ 
7.     # pragma omp task shared ( $M_C$ )
8.      $M_C \leftarrow BRI_C(k-1, linha[2:k], coluna[1:k-1])$ 
9.     # pragma omp task shared ( $M_D$ )
10.     $M_D \leftarrow BRI_D(k-1, linha[2:k], coluna[2:k])$ 
11.    # pragma omp taskwait
12.     $(M/D) \leftarrow M_A - M_B * inv(M_D) * M_C$ 
13.  else
14.     $(M/D) \leftarrow A - B * invFolha * C$ 
15.  end if
16. end function

```

Saída: (M/D)

cálculo do complemento de Schur, no procedimento recursivo retroativo. A diretiva *taskwait* se trata de um ponto de sincronização das tarefas criadas, assim, a instrução $(M/D) \leftarrow M_A - M_B * inv(M_D) * M_C$ será atingida somente após o término das tarefas filhas. Assim, esta diretiva de sincronização bloqueia a execução da função até que a tarefa tenha sido concluída.

Uma vez que o OpenMP considera como privadas as variáveis declaradas anteriormente à diretiva *task* por omissão, faz-se necessário tornar as variáveis M_A , M_B , M_D e M_C compartilhadas. Tal é feito através das opções *shared*(M_A), *shared*(M_B), *shared*(M_C) e *shared*(M_D), como apresentado no Algoritmo 10.

Já os parâmetros de entrada da referida função, a saber, k e os vetores *linha* e *coluna* são classificados como *private*, pois cada nó é constituído por quadros com configurações diferentes de linha de blocos e coluna de blocos. Desta maneira, é fundamental que cada *thread* possa trocar as suas variáveis sem afetar os demais nós.

As funções BRI_B , BRI_C e BRI_D têm um pseudocódigo semelhante ao Algoritmo 10, a única diferença é o tipo de operação de complemento de Schur aplicado sobre os dados, conforme apresentado nos Algoritmos 3, 4 e 5.

6.4.1 Resultados Experimentais

Para fins de comparação, foram realizados experimentos com o algoritmo BRI original (versão sequencial), apresentado no Capítulo 4 e com a versão paralela do mesmo, conforme analisado na Seção 6.4, medindo tempo de execução.

Todos os testes foram executados em nós computacionais do supercomputador do Núcleo de Processamento de Alto Desempenho (NPAD) da Universidade Federal do Rio Grande do Norte. Cada nó possui a seguinte configuração: 2 CPUs Intel *Xeon Sixteen-Core* E5-2698v3 de 2,3 GHz, com 40M de *cache*, e 128 GB de RAM. Sendo assim, cada nó apresenta um desempenho de 32 núcleos¹.

Tempo de execução

A medição dos tempos de processamento da versão sequencial e da versão paralela do algoritmo BRI foi realizada a partir da função *gettimeofday*² da biblioteca *sys/time.h*, cujos valores obtidos encontram-se na Tabela 6.1.

Os experimentos com a implementação paralela do BRI foram realizados com grupos de *threads* de tamanho 32.

Analisando a Tabela 6.1, observa-se que os algoritmos apresentam padrões opostos em relação ao tempo de processamento. Considerando a versão sequencial do BRI, o tempo de execução cresce com o aumento de k . Por outro lado, na versão paralela, o tempo de execução decresce para maiores valores de k .

¹Núcleo (ou *core*) é um termo de hardware que descreve o número de unidades de processamento central independentes em um único componente de computação (matriz ou chip).

²A função *gettimeofday* retorna a hora atual em segundos ou microsegundos de acordo com o parâmetro passado.

Tam. da Matriz	$k = 2$		$k = 3$		$k = 4$		$k = 5$	
	Serial (s)	Paralelo (s)	Serial (s)	Paralelo (s)	Serial (s)	Paralelo (s)	Serial (s)	Paralelo (s)
900	0.054	0.076	0.073	0,045	0.143	0,051	0.334	0,083
2100	0.362	0.609	0.503	0,294	0.996	0,206	2.306	0,203
3300	1.876	1.745	1.573	0,705	2.991	0,485	6.765	0,584
4500	3.729	3.189	3.888	2,144	6.661	1,062	14.775	1,061
5700	5.654	5.516	7.824	3,526	12.553	2,428	27.221	1,992
6900	9.414	9.951	12.748	5,667	21.590	3,746	46.108	3,461
8100	15.179	14.869	19.671	9,567	33.704	5,609	72.0555	5,058
9300	21.902	22.062	28.532	13,492	48.513	8,759	105.698	7,37
10500	30.670	30.435	39.447	18,302	67.586	11,698	145.839	10,332

Tabela 6.1: Tempos de processamento do BRI sequencial e do BRI paralelo

O crescimento do tempo de execução no programa serial acompanha o aumento de k porque esse parâmetro influencia na quantidade de níveis da árvore de chamadas. Assim, quanto mais complexa a estrutura da árvore maior o tempo para percorre-la e, por conseguinte, isso interfere na duração do programa.

O resultado obtido no programa paralelo ocorre devido à diferença de velocidade no acesso de cada tipo de memória, conforme apresentado na Seção 3.3. Na computação paralela, o aumento do número de processadores implica tanto na diminuição da quantidade de tarefas por processador quanto no aumento de memória *cache* com dados do programa, devido ao acúmulo de memória associada a cada processador. Dessa forma, o aumento do tamanho da *cache* permite que mais blocos da matriz possam ser armazenados na mesma e, consequentemente, isso gera um aumento de *cache hits* e uma redução considerável no tempo de processamento do programa.

Análise de threads

Programas com paralelismo de tarefas necessitam de processamento adicional (sobrecusto resultante do gerenciamento de *threads* e tarefas) em relação as suas versões sequenciais. Este sobrecusto é decorrente do custo imposto ao sistema operacional para gerenciar o grupo de *threads* utilizadas pelo programa, bem como o custo de criação,

manutenção e escalonamento das tarefas OpenMP [Silva 2011]. Portanto, isso influencia diretamente no desempenho de um programa paralelo.

Em se tratando do BRI, outros parâmetros, como a quantidade de blocos (k) e o tamanho da matriz, também apresentam ligação direta com o seu desempenho, conforme será analisado a seguir. Os experimentos foram realizados com grupos de *threads* de tamanhos 2, 4, 8, 16 e 32. Nos gráficos apresentados posteriormente, p indica o tamanho do grupo de *threads* (definido através da variável de ambiente OMP_NUM_THREADS), sendo $p = 1$ a representação do algoritmo sequencial.

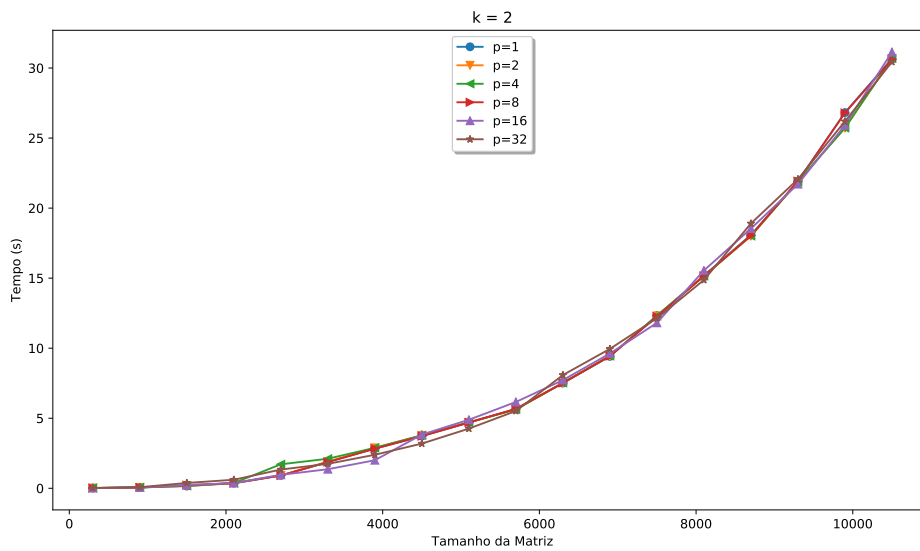


Figura 6.5: Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 2$, utilizando diferentes grupos de *threads* (p).

Observa-se na Figura 6.5 que, para a quantidade de blocos $k = 2$, o tempo de execução do algoritmo BRI paralelo para inverter uma determinada matriz $m \times m$, praticamente, se mantém para diferentes tamanhos de grupos de *threads*, p . Por isso, os gráficos estão praticamente sobrepostos. Isso acontece porque, como apresentado no Algoritmo 9, para $k \leq 2$, a função não entrará em recursividade. Assim, as Linhas 13 e 15 serão executadas de forma sequencial. Consequentemente, esses testes não entrarão nas análises de desempenho de paralelismo abordadas na Subseção 6.4.1.

Para $k = 3$, a árvore de chamadas recursivas do algoritmo BRI possui apenas um nível. Desta forma, serão criadas apenas 4 (quatro) tarefas que serão executadas por 4 *threads* (uma para cada).

Quanto mais *threads* disponíveis para processar as tarefas, menor o tempo em que mesmas ficarão suspensas, esperando uma *thread* atendê-las. Assim, maior será a quantidade de tarefas executadas por unidade de tempo [Silva 2011]. Sendo assim, para uma mesma matriz de entrada $m \times m$ com $k = 3$, o tempo de execução do algoritmo BRI paralelo será maior para tamanhos de grupo de *threads* $p < 4$. Já para as execuções com $p > 4$, o desempenho do algoritmo será semelhante ao mesmo obtido com $p = 4$, conforme ilustrado na Figura 6.6.

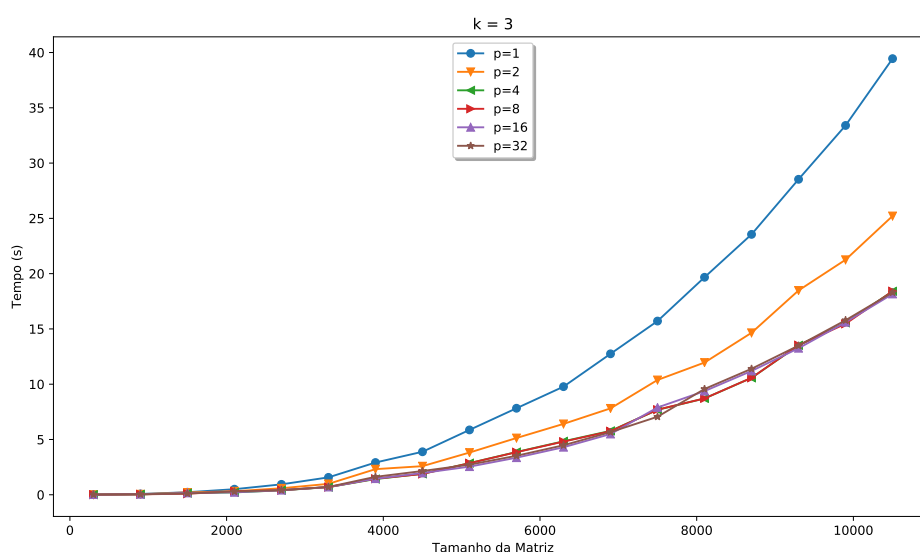


Figura 6.6: Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 3$, utilizando diferentes grupos de *threads* (p).

Sendo $k = 4$, a árvore de chamadas recursivas do algoritmo BRI possui 2 (dois) níveis. Portanto, serão criadas, no máximo, 20 (vinte) tarefas.

Conforme o gráfico ilustrado na Figura 6.7, as curvas começam a se distanciar. Além disso, observa-se que quando maior o tamanho da matriz, maior a separação entre as curvas. Isso ocorre, porque a granularidade começa a se tornar mais fina. Desta forma, quanto menos *threads* disponíveis para processar as tarefas, maior o tempo em que mesmas ficarão suspensas, esperando algum *thread* ficar livre.

Neste caso, para uma mesma matriz de entrada $m \times m$, com $k = 4$, o tempo de execução do algoritmo BRI paralelo será maior para grupos de *threads* com tamanhos $p < 20$, conforme ilustrado na Figura 6.6.

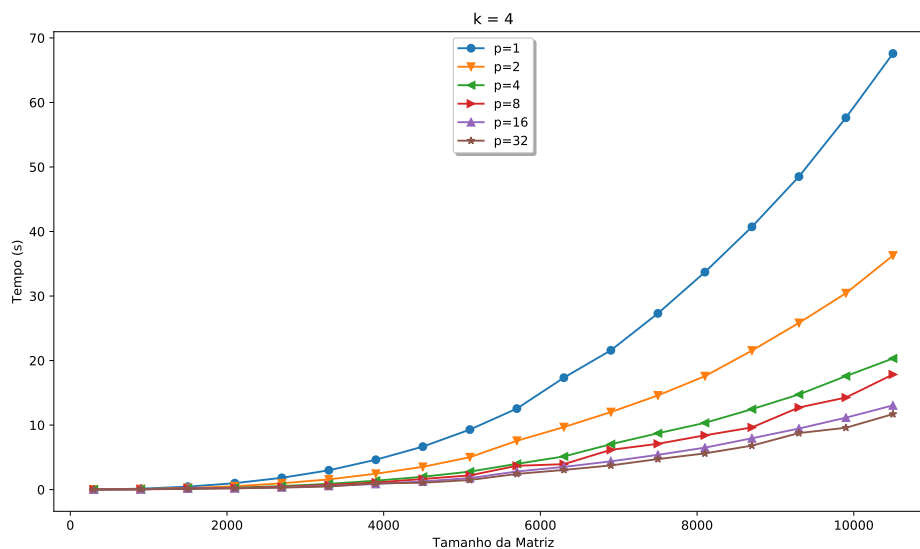


Figura 6.7: Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 4$, utilizando diferentes grupos de *threads* (p).

E, finalmente, para $k = 5$ o algoritmo necessita do uso de até 84 *threads*, a saber, 4 *threads* no primeiro nível, 16 no segundo nível e mais 64 no terceiro nível. Assim, a granularidade se torna mais fina e, por consequência, ocorre um maior aproveitamento da paralelização. Conforme a Figura 6.8, percebe-se uma distinção mais bem definida entre as curvas do gráfico.

Para $p = 32$, é possível perceber que o aumento do tamanho da matriz de entrada não resulta em um impacto tão considerável no tempo de processamento como no algoritmo serial.

Dessa forma, pode-se concluir que, quanto maior o valor de k , maior o nível de paralelismo e, consequentemente, maior será o impacto no tempo de execução do programa. As curvas do gráfico, portanto, tenderão a se afastem com o aumento do valor de k .

Speedup e eficiência

A seguir estão os gráficos com as curvas de *speedup* obtidas em função do tamanho do grupo de *threads* utilizado durante os experimentos.

Na Figura 6.9, as curvas de *speedup* foram medidos para uma matriz de entrada com dimensão igual a 6000 e grupos de *threads* com tamanhos iguais a 2, 4, 8, 16 e 32. O algoritmo paralelo do BRI apresenta um comportamento no qual o aumento da quantidade

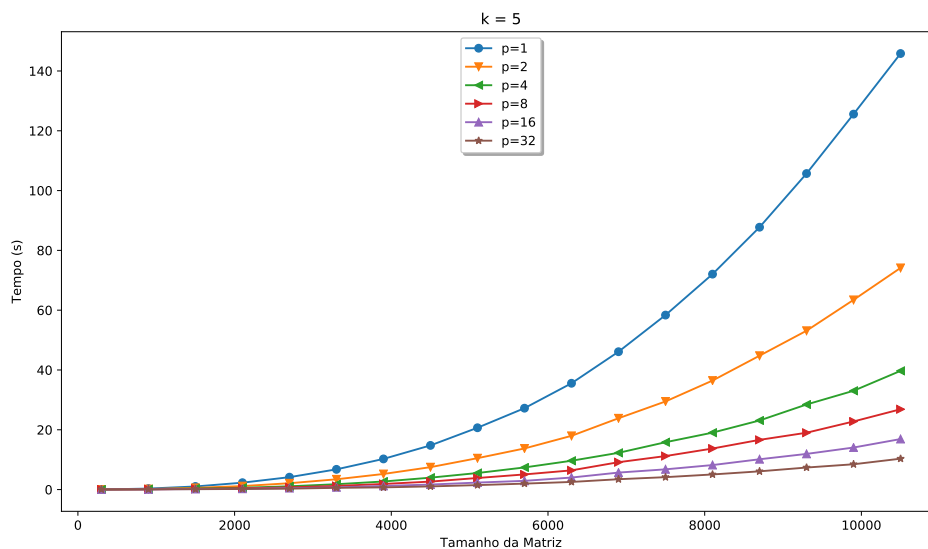


Figura 6.8: Tempo de execução do algoritmo BRI paralelo na inversão de matrizes $m \times m$, com $k = 5$, utilizando diferentes grupos de *threads* (p).

de *threads* acompanha o crescimento do *speedup*. Tal comportamento é explicado pelo fato de que quanto mais *threads* disponíveis para processar as tarefas, mais rapidamente a fila de tarefas será esvaziada.

Em conformidade ao que foi discutido na Seção 6.4.1, é válido observar que, para $k = 3$, o *speedup* se mantém constante para grupos de *threads* de tamanho $p > 4$. Isto ocorre porque serão criadas apenas 4 tarefas ao mesmo tempo durante a inversão da matriz de entrada. Para $k = 4$, praticamente, não existe grande diferença entre os dois últimos pontos analisados, pois as 20 tarefas criadas serão processadas quase que simultaneamente pelos *threads* disponibilizados. E, finalmente, para $k = 5$, o *speedup* cresce consideravelmente com o aumento de *threads* devido ao maior aproveitamento da paralelização.

Sob outra perspectiva, observa-se que o aumento do k afeta positivamente o *speedup* do algoritmo, pois proporciona que a matriz original sofra mais divisões e, consequentemente, apresente blocos com ordem menor. Portanto, a diminuição do tamanho bloco faz com que as tarefas executadas por cada *thread* sejam menores e o processamento dos blocos da matriz sejam distribuídos de forma mais igualitária entre as *threads*. Assim sendo, a granularidade diminui e o algoritmo apresenta um maior nível de paralelização. Dessa forma, essa métrica induz que o algoritmo BRI paralelo apresenta melhor desempenho para maiores valores de k .

A Figura 6.10 apresenta as curvas da eficiência geradas para as mesmas entradas con-

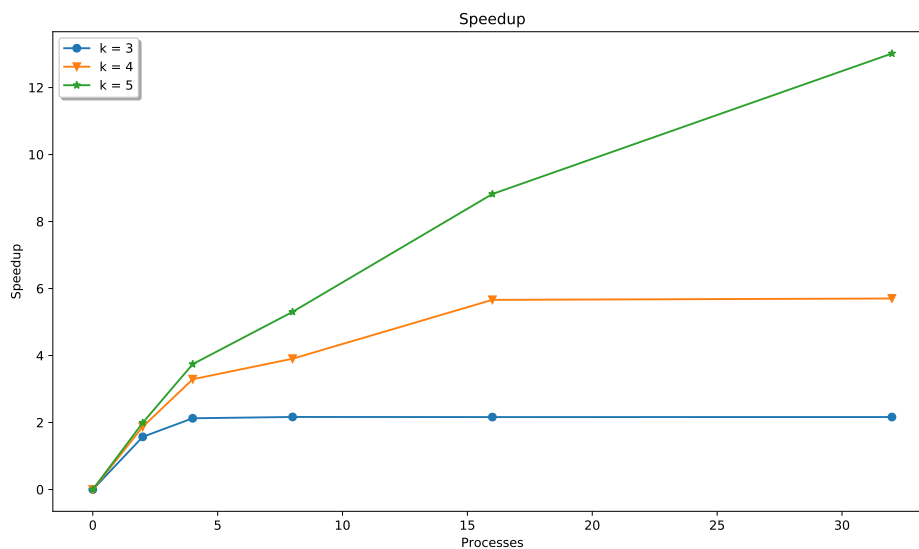


Figura 6.9: Curvas do *speedup* em função do tamanho de grupo de *threads*.

sideradas, anteriormente, no cálculo do *speedup*. Esse gráfico mostra que, para um determinado k , a eficiência decresce com o aumento do número de *threads*. Esse resultado é previsível, pois a taxa de crescimento do *speedup* decresce com o aumento do tamanho do grupo de *threads* utilizados pelo algoritmo. Isso é explicado pelo fato de que quanto maior o tamanho do grupo de *threads* utilizado pelo algoritmo, maior o sobrecusto resultante do gerenciamento de *threads*.

De outra forma, é possível perceber que a eficiência e, consequentemente, o desempenho do algoritmo aumentam para valores de k mais elevados. Este fato induz, então, que a paralelização proporciona melhores resultados justamente nos casos em que a versão sequencial do BRI apresenta maiores tempos de processamento.

As Figuras 6.11, 6.12 e 6.13 mostram a eficiência do algoritmo BRI paralelo com diferentes tamanhos da matriz de entrada, a saber, 3000, 6000, 12000, 24000 e 48000, e diferentes valores de p e k . Com gráficos, é possível perceber valores similares de eficiência para diferentes tamanhos da matriz de entrada e grupos de *threads* de mesmo tamanho.

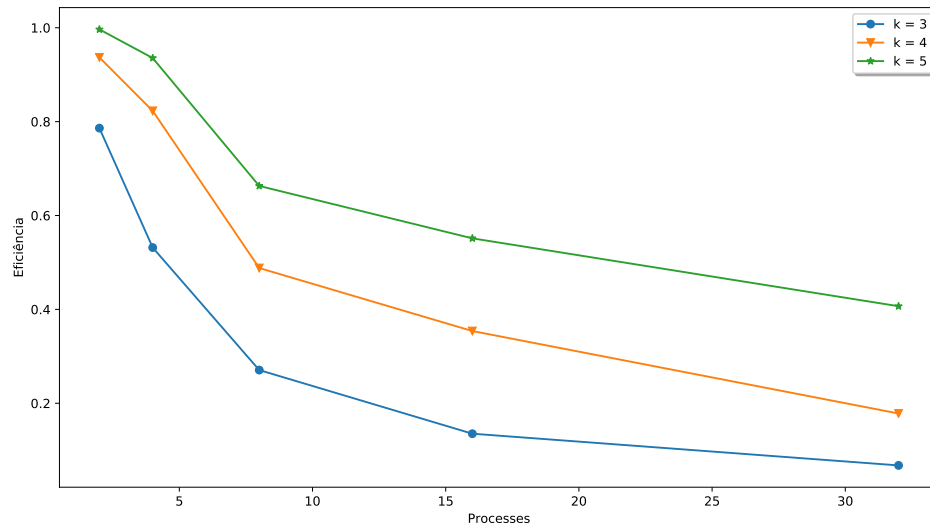


Figura 6.10: Curvas da eficiência em função do tamanho de grupo de *threads*.

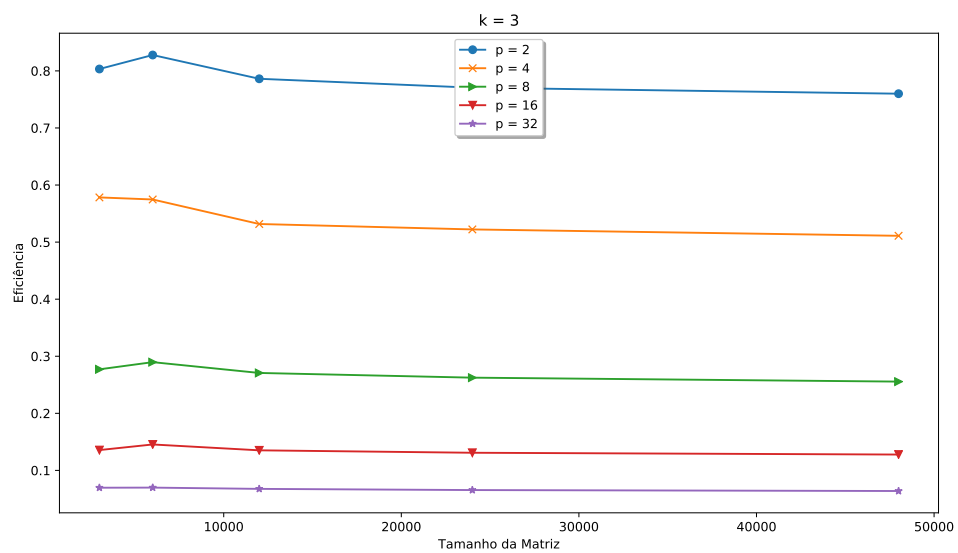
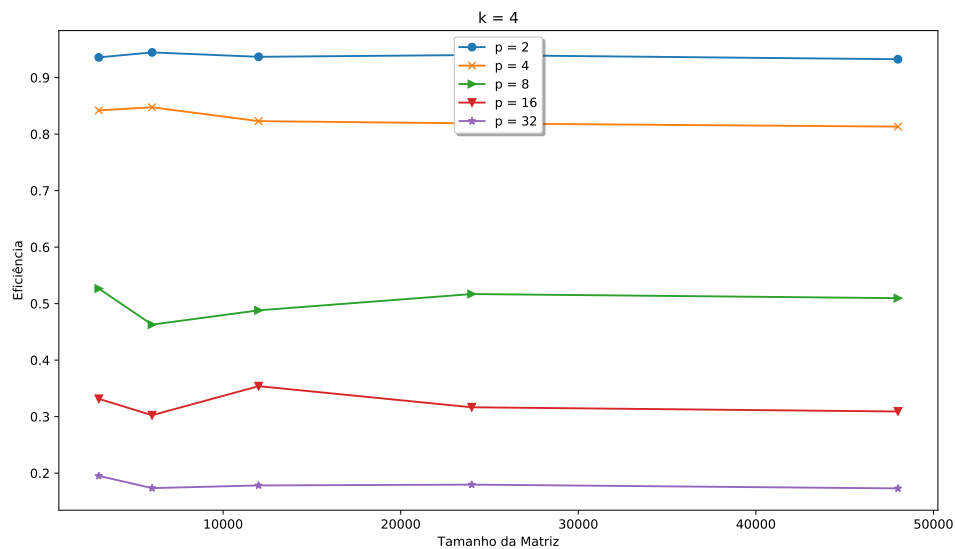
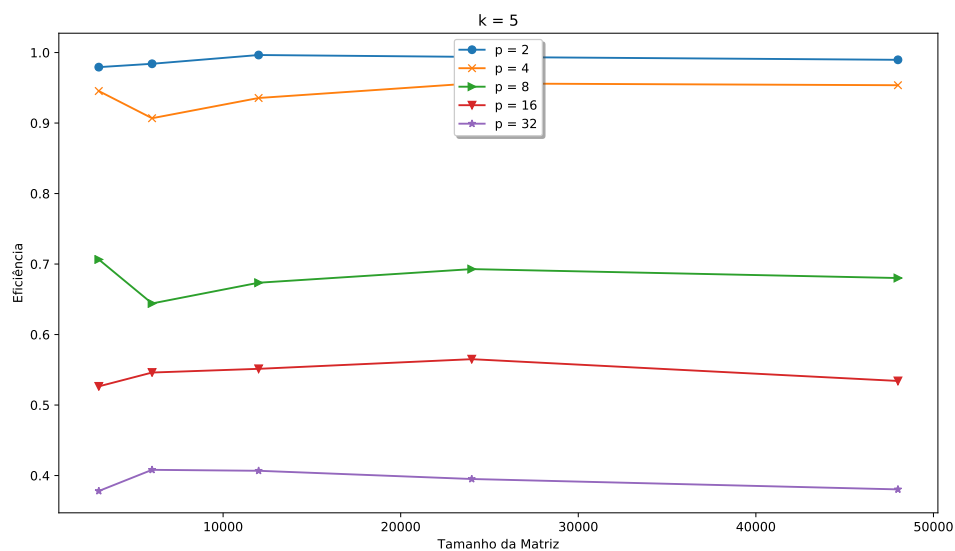


Figura 6.11: Eficiência para diferentes matrizes de entrada com $k = 3$.

Figura 6.12: Eficiência para diferentes matrizes de entrada com $k = 4$.Figura 6.13: Eficiência para diferentes matrizes de entrada com $k = 5$.

Capítulo 7

Considerações Finais

Esta tese propõe um novo algoritmo recursivo para a inversão de matrizes $m \times m$ particionadas em $k \times k$ blocos, com $k \geq 2$, denominado de Inversão Recursiva de Blocos (BRI). Este algoritmo obtém um bloco da matriz de cada vez, resultante da divisão recursiva da matriz original em 4 matrizes quadradas de uma ordem inferior. O algoritmo proposto permite uma redução do uso de memória que, por sua vez, permite a inversão de matrizes de alta ordem que, de outra forma, excederiam a capacidade de memória do computador.

Os resultados experimentais mostraram que o algoritmo proposto consome muito menos memória do que a inversão utilizando decomposição LU. Esse uso de memória diminui à medida que o número de blocos aumenta, para uma mesma matriz de entrada. Aumentar o número de blocos também resulta em um aumento do tempo de processamento, o que caracteriza uma troca entre o tempo de processamento e uso da memória. Contudo, devido ao alto grau de concorrência do BRI, esse maior tempo de processamento pode ser amenizado usando processamento paralelo para calcular todos os blocos ao mesmo tempo de forma embarçosamente paralela.

Uma versão paralela do algoritmo BRI, utilizando OpenMP, foi apresentada e discutida no Capítulo 6. Também foi abordado uma otimização do algoritmo sequencial do BRI ao propor a reutilização de alguns cálculos dentre algumas ramificações no procedimento de recursão. A análise realizada mostrou uma redução considerável no tempo de processamento em comparação à versão original do BRI.

Vale ressaltar que o BRI é ainda mais útil nos casos em que é necessário apenas usar partes da matriz inversa, como, por exemplo, nos cálculos para estimar os rótulos previstos dos algoritmos de validação cruzada em LS-SVMs [An et al. 2007], conforme analisado no Capítulo 5.

Como trabalhos futuros, é possível citar a paralelização do algoritmo utilizando outras APIs para programação paralela, como por exemplo, a MPI, que baseia-se em passagem de mensagem para sistemas de memória distribuída. Além disso, a análise dos efeitos

da abordagem do BRI em ganhos de desempenho devido a um possível melhor uso da hierarquia de memória apresenta-se como um tópico relevante para futuros trabalhos.

Outras extensões do trabalho são o desenvolvimento da prova matemática desta técnica, uma reformulação do algoritmo com respeito ao complemento de Schur dado por (3.11) e possíveis aplicações em engenharia e inteligência computacional.

Nesta tese, não foi implementada nenhuma técnica específica para otimizar o produto entre matrizes. Sendo assim, o BRI faz uso apenas do algoritmo padrão de multiplicação matricial que possui complexidade $O(b^3)$. Existem outros algoritmos para multiplicação de matrizes que possuem complexidade menor e podem ser implementados para se obter um melhor desempenho do sistema computacional, como, por exemplo, o algoritmo de Strassen, com $O(b^{2,807})$, e o algoritmo de Coppersmith-Winograd, com $O(b^{2,376})$.

Também é possível trocar o método de inversão da matriz de bloco 2×2 , no Procedimento Recursivo Retroativo, apresentado na Subseção 4.1.2, e implementar outro algoritmo a fim de comparar o desempenho do sistema.

Como fruto desta pesquisa, o algoritmo BRI foi publicado em [Cosme et al. 2018].

Referências Bibliográficas

- Althoen, S. C. & R. Mclaughlin (1987), ‘Gauss-Jordan reduction: a brief history’, *American Mathematical Monthly* **94**(2), 130–142.
- An, S., W. Liu & S. Venkatesh (2007), ‘Fast cross-validation algorithms for least squares support vector machine and kernel ridge regression’, *Pattern Recognition* **40**(8), 2154–2162.
- Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney & D. Sorensen (1999), *LAPACK Users’ guide*, Vol. 9, Society for Industrial and Applied Mathematics (SIAM).
- Anderson, E., Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof & D. Sorensen (1990), Lapack: a portable linear algebra library for high-performance computers, *em* ‘Proceedings of the 1990 ACM/IEEE conference on Supercomputing’, IEEE Computer Society Press, pp. 2–11.
- Anton, H. & R. C. Busby (2006), *Álgebra linear contemporânea*, Editora Bookman.
- Baker, J., F. Hiergeist & G. Trapp (1988), ‘A recursive algorithm to invert multiblock circulant matrices’, *Kyungpook Math. J* **28**, 45–50.
- Banachiewicz, T. (1937), ‘Zur berechnung der determinanten, wie auch der inversen und zur darauf basierten auflosung der systeme linearer gleichungen’, *Acta Astronom. Ser. C* **3**, 41–67.
- Betts, J. T. (2001), *Practical methods for optimal control using nonlinear programming*, Advances in Design and Control, Society for Industrial and Applied Mathematics (SIAM).
- Bientinesi, P., B. Gunter & R. A. Geijn (2008), ‘Families of algorithms related to the inversion of a symmetric positive definite matrix’, *ACM Transactions on Mathematical Software (TOMS)* **35**(1), 3.

- Björck, A. (1996), *Numerical methods for least squares problems*, Society for Industrial and Applied Mathematics (SIAM).
- Braga, A. de P., A. C. P. L. F. Carvalho & T. B. Ludermir (2000), *Redes neurais artificiais: teoria e aplicações*, Livros Técnicos e Científicos.
- Brezinski, C. (1988), ‘Other manifestations of the schur complement’, *Linear Algebra and its Applications* **111**, 231–247.
- Brezzi, F. & M. Fortin (1991), *Mixed and hybrid finite element methods*, Springer-Verlag, New York.
- Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert & C. E. Leiserson (2009), Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks, *em* ‘Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures’, ACM, pp. 233–244.
- Burges, C. J. C. (1996), Simplified support vector decision rules, *em* ‘Proceedings of 13th International Conference on Machine Learning’, Vol. 96, pp. 71–77.
- Byers, R. (1987), ‘Solving the algebraic riccati equation with the matrix sign function’, *Linear Algebra and its Applications* **85**, 267–279.
- Carvalho, B. P. R. (2005a), ‘O estado da arte em métodos para reconhecimento de padrões: Support Vector Machine’, *Congresso Nacional de Tecnologia da Informação e Comunicação (Sucesu 2005)*.
- Carvalho, B. P. R., M. B. Almeida & A. P. Braga (2002), ‘Support Vector Machines: um estudo sobre técnicas de treinamento’, *Monografia, Universidade Federal de Minas Gerais*, http://www.litc.cpdee.ufmg.br/upload/periodicos/Internal_Monograph58.zip.
- Chandorkar, M., R. Mall, O. Lauwers, J. A. K. Suykens & B. De Moor (2015), Fixed-size least squares support vector machines: scala implementation for large scale classification, *em* ‘Computational Intelligence, 2015 IEEE Symposium Series on’, IEEE, pp. 522–528.
- Chapman, B., G. Jost & R. Van Der Pas (2008), *Using OpenMP: portable shared memory parallel programming*, MIT Press.

- Chatterjee, S., A. R. Lebeck, P. K. Patnala & M. Thottethodi (2002), ‘Recursive array layouts and fast matrix multiplication’, *IEEE Transactions on Parallel and Distributed Systems* **13**(11), 1105–1123.
- Chen, M., S. Mao & Y. Liu (2014), ‘Big data: a survey’, *Mobile networks and applications* **19**(2), 171–209.
- Claudio, D. M. & J. M. Marins (2000), *Cálculo numérico computacional: teoria e prática*, 3^a edição, Editora Atlas.
- Cosme, I. C. S., I. F. Fernandes, J. L. de Carvalho & S. Xavier-de-Souza (2018), ‘Memory-usage advantageous block recursive matrix inverse’, *Applied Mathematics and Computation* **328**, 125–136.
- De Brabanter, K., J. De Brabanter, J. A. K. Suykens & B. De Moor (2010), ‘Optimized fixed-size kernel models for large data sets’, *Computational Statistics & Data Analysis* **54**(6), 1484–1504.
- De Brabanter, K., P. Karsmakers, F. Ojeda, C. Alzate, J. De Brabanter, K. Pelckmans, B. De Moor, J. Vandewalle & J. A. K. Suykens (2011), *LS-SVMLab Toolbox User’s Guide: version 1.8*, Katholieke Universiteit Leuven.
- Dean, J. & S. Ghemawat (2008), ‘MapReduce: simplified data processing on large clusters’, *Communications of the ACM* **51**(1), 107–113.
- Dongarra, J. & P. Luszczek (2011), Linpack benchmark, *em* ‘Encyclopedia of Parallel Computing’, Springer, pp. 1033–1036.
- Downs, T., K. E. Gates & A. Masters (2002), ‘Exact simplification of support vector solutions’, *Journal of Machine Learning Research* **2**, 293–297.
- Drouvelis, P. S., P. Schmelcher & P. Bastian (2006), ‘Parallel implementation of the recursive greens function method’, *Journal of Computational Physics* **215**(2), 741–756.
- Edwards, R. E., H. Zhang, L. E. Parker & J. R. New (2013), Approximate l-fold cross-validation with least squares svm and kernel ridge regression, *em* ‘Machine Learning and Applications (ICMLA), 2013 12th International Conference on’, Vol. 1, IEEE, pp. 58–64.
- Elman, H., V. E. Howle, J. Shadid, R. Shuttleworth & R. Tuminaro (2008), ‘A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible navier–stokes equations’, *Journal of Computational Physics* **227**(3), 1790–1808.

- Ezzatti, P., E. S. Quintana-Orti & A. Remon (2011), High performance matrix inversion on a multi-core platform with several GPUs, *em* 'Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on', IEEE, pp. 87–93.
- Fletcher, R. (1987), *Practical methods of optimization*, John Wiley & Sons.
- García-López, F., B. Melián-Batista, J. A. Moreno-Pérez & J. M. Moreno-Vega (2002), 'The parallel variable neighborhood search for the p-median problem', *Journal of Heuristics* **8**(3), 375–388.
- Geebelen, D., J. A. K. Suykens & J. Vandewalle (2012), 'Reducing the number of support vectors of SVM classifiers using the smoothed separable case approximation', *IEEE Transactions on Neural Networks and Learning Systems* **23**(4), 682–688.
- Ghiya, R., L. J. Hendren & Y. Zhu (1998), Detecting parallelism in C programs with recursive data structures, *em* 'International Conference on Compiler Construction', Springer, pp. 159–173.
- Golub, G. H. & C. F. Van Loan (2013), *Matrix computations*, Vol. 4, Johns Hopkins University Press.
- Haynsworth, E. V. (1968a), 'Determination of the inertia of a partitioned hermitian matrix', *Linear algebra and its applications* **1**(1), 73–81.
- Haynsworth, E. V. (1968b), On the schur complement, Relatório técnico, BASEL UNIV (SWITZERLAND) MATHEMATICS INST.
- Hebrich, R. (2002), 'Learning kernel classifiers: theory and algorithms'.
- Heinecke, A. & M. Bader (2008), Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms, *em* 'Proceedings of the 2008 workshop on Memory access on future processors: a solved problem', ACM, pp. 385–392.
- Huang, K., D. Zheng, J. Sun, Y. Hotta, K. Fujimoto & S. Naoi (2010), 'Sparse learning for support vector classification', *Pattern Recognition Letters* **31**(13), 1944–1951.
- Jonsson, I. & B. Kågström (2002), 'Recursive blocked algorithms for solving triangular systems – part II: two-sided and generalized sylvester and lyapunov matrix equations', *ACM Transactions on Mathematical Software (TOMS)* **28**(4), 416–435.

- Kågström, B., P. Ling & C. Van Loan (1998), ‘Gemm-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark’, *ACM Transactions on Mathematical Software (TOMS)* **24**(3), 268–302.
- Karimi, S. & B. Zali (2014), ‘The block preconditioned LSQR and GL-LSQR algorithms for the block partitioned matrices’, *Applied Mathematics and Computation* **227**, 811–820.
- Lau, K. K., M. J. Kumar & R. Venkatesh (1996), Parallel matrix inversion techniques, *em* ‘Algorithms & Architectures for Parallel Processing, 1996. ICAPP 96. 1996 IEEE Second International Conference on’, IEEE, pp. 515–521.
- Li, Y., C. Lin & W. Zhang (2006), ‘Improved sparse least-squares support vector machine classifiers’, *Neurocomputing* **69**(13), 1655–1658.
- Liu, J., Y. Liang & N. Ansari (2016), ‘Spark-based large-scale matrix inversion for big data processing’, *IEEE Access* **4**, 2166–2176.
- López, J., K. De Brabanter, J. R. Dorronsoro & J. A. K. Suykens (2011), Sparse LS-SVMs with L0-norm minimization, *em* ‘Proceedings of the 19th European Symposium on Artificial Neural Networks’, pp. 189–194.
- Lorena, A. C. & A. C. P. L. F. de Carvalho (2007), ‘Uma introdução às support vector machines’, *Revista de Informática Teórica e Aplicada* **14**(2), 43–67.
- Lu, T.-T. & S.-H. Shiou (2002), ‘Inverses of 2×2 block matrices’, *Computers & Mathematics with Applications* **43**(1), 119–129.
- Madsen, N. K. (1983), ‘Cyclic odd-even reduction for symmetric circulant matrices’, *Linear algebra and its applications* **51**, 17–35.
- Mahfoudhi, R., S. Achour, O. Hamdl-Larbl & Z. Mahjoub (2017), High performance recursive matrix inversion for multicore architectures, *em* ‘High Performance Computing & Simulation (HPCS), 2017 International Conference on’, IEEE, pp. 675–682.
- Mall, R. & J. A. K. Suykens (2002), ‘Reduced fixed-size LSSVM for large scale data’, *Neurocomputing* **48**(1–4), 1025–1031.
- Maziero, C. A. (2014), ‘Sistemas operacionais: conceitos e mecanismos’, *Livro aberto. Acessível em: <http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php>*.

- McCullagh, P. & J. A. Nelder (1989), *Generalized linear models*, 2ª edição, Chapman & Hal Londonl.
- Mercer, J. (1909), ‘Functions of positive and negative type, and their connection with the theory of integral equations’, *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character* pp. 415–446.
- Mitchell, T. M. (1997), *Machine learning*, McGraw-Hill Inc.
- Nethercote, N. & J. Seward (2007), Valgrind: a framework for heavyweight dynamic binary instrumentation, *em* ‘ACM Sigplan notices’, Vol. 42, ACM, pp. 89–100.
- Noble, B. & J. W. Daniel (1988), *Applied linear algebra*, Prentice-Hall New Jersey.
- Pacheco, P. (2011), *An introduction to parallel programming*, Elsevier.
- Patterson, D. A. & J. L. Hennessy (2013), *Computer organization and design: the hardware/software interface*, 5ª edição, Elsevier.
- Pelckmans, K., J. A. K. Suykens, T. Van Gestel, J. De Brabanter, L. Lukas, B. Hamers, B. De Moor & J. Vandewalle (2002), ‘LS-SVMLab: a Matlab/C toolbox for Least Squares Support Vector Machines’, *Tutorial. KULeuven-ESAT. Leuven, Belgium* **142**, 1–2.
- Press, W. H., B. P. Flannery, S. A. Teukolsky & W. T. Vetterling (2007), *Numerical recipes: the art of scientific computing*, 3ª edição, Cambridge university press.
- Sadashiv, N. & S. M. D. Kumar (2011), Cluster, grid and cloud computing: a detailed comparison, *em* ‘6th International Conference on Computer Science & Education (ICCSE 2011)’, IEEE, pp. 477–482.
- Sanderson, C. & R. Curtin (2016), ‘Armadillo: a template-based C++ library for linear algebra’, *Journal of Open Source Software* **1**(2), 26–32.
- Schur, J. (1917), ‘Über potenzreihen, die im innern des einheitskreises beschränkt sind’, *Journal für die reine und angewandte Mathematik* **147**, 205–232.
- Shao, J. (1993), ‘Linear model selection by cross-validation’, *Journal of the American statistical Association* **88**(422), 486–494.

- Silva, I. N. da, D. H. Spatti & R. A. Flauzino (2010), *Redes neurais artificiais: para engenharia e ciências aplicadas*, Artliber.
- Silva, M. C. S., S. Xavier-de-Souza & I. C. S. Cosme (2017), ‘Paralelização do algoritmo de inversão recursiva de blocos’. Trabalho de Conclusão de Curso (Engenharia da Computação) – Universidade Federal do Rio Grande do Norte.
- Silva, M. de O. da (2011), ‘Controle de granularidade de tarefas em OpenMP’. Trabalho de Conclusão de Curso (Ciência da Computação) – Universidade Federal do Rio Grande do Sul.
- Stallings, W. (2010), *Arquitetura e Organização de Computadores*, 8ª edição, Prentice-Hall, Pearson.
- Stone, M. (1978), ‘Cross-validation: a review’, *Statistics: A Journal of Theoretical and Applied Statistics* **9**(1), 127–139.
- Strassen, V. (1969), ‘Gaussian elimination is not optimal’, *Numerische Mathematik* **13**(4), 354–356.
- Suykens, J. A. K. & J. Vandewalle (1999), ‘Least Squares Support Vector Machine classifiers’, *Neural processing letters* **9**(3), 293–300.
- Suykens, J. A. K., L. Lukas & J. Vandewalle (2000), Sparse approximation using Least Squares Support Vector Machines, *em* ‘Proceedings of IEEE International Symposium on Circuits and Systems’, Vol. 2, pp. 757–760.
- Suykens, J. A. K., T. Van Gestel, J. De Brabanter, B. De Moor & J. Vandewalle (2002), *Least Squares Support Vector Machines*, Vol. 4, World Scientific.
- Tanenbaum, A. S. (2003), *Sistemas operacionais modernos*, 2ª edição, Prentice Hall.
- Tsitsas, N. L. (2010), ‘On block matrices associated with discrete trigonometric transforms and their use in the theory of wave propagation’, *Journal of Computational Mathematics* **28**(6), 864–878.
- Tsitsas, N. L., E. G. Alivizatos & G. H. Kalogeropoulos (2007), ‘A recursive algorithm for the inversion of matrices with circulant blocks’, *Applied mathematics and computation* **188**(1), 877–894.
- Vapnik, V. (1995), *The nature of statistical learning theory*, Springer-Verlag.

- Vapnik, V. N. (1998), *Statistical learning theory*, Vol. 3, Wiley, New York.
- Veloso, P., C. dos Santos, P. Azeredo & A. Furtado (1986), *Estrutura de dados*, Editora Campus.
- Vescovo, R. (1997), ‘Inversion of block-circulant matrices and circular array approach’, *IEEE Transactions on Antennas and Propagation* **45**(10), 1565–1567.
- Wang, Z. & A. C. Bovik (2009), ‘Mean squared error: Love it or leave it? A new look at signal fidelity measures’, *IEEE signal processing magazine* **26**(1), 98–117.
- Weston, J., A. Elisseeff, B. Schölkopf & M. Tipping (2003), ‘Use of the zero norm with linear models and kernel methods’, *Journal of Machine Learning Research* **3**, 1439–1461.
- Williams, C. & M. Seeger (2001), Using the Nyström method to speed up kernel machines, *em* ‘Proceedings of the 14th Annual Conference on Neural Information Processing Systems’, Vol. 13, pp. 682–688.
- Xiang, J., H. Meng & A. Aboulmaga (2014), Scalable matrix inversion using MapReduce, *em* ‘Proceedings of the 23rd international symposium on High-performance parallel and distributed computing’, ACM, pp. 177–190.
- Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker & I. Stoica (2010), ‘Spark: cluster computing with working sets.’, *HotCloud* **10**(10-10), 95.
- Zhang, F. (2005), *The Schur complement and its applications*, Vol. 4, Springer Science & Business Media.