

UNIVERSIDADE FEDERAL DA BAHIA
Curso de Pós-graduação em Geofísica

TESE DE DOUTORADO



**SOLUÇÃO DE SISTEMAS
LINEARES DE GRANDE
PORTE E COMPUTAÇÃO DE
ALTO DESEMPENHO**

CRISTIAN DAVID ARIZA ARIZA

SALVADOR – BAHIA
JANEIRO – 2024

Solução de sistemas lineares de grande porte e computação de alto desempenho

por

CRISTIAN DAVID ARIZA ARIZA

Físico (Universidade Pedagógica e Tecnológica da Colômbia, 2003)

Mestre em Geofísica (Universidade Federal Da Bahia, 2016)

Orientador: Prof. Dr. Milton J. Porsani

TESE DE DOUTORADO

Submetida em satisfação parcial dos requisitos ao grau de

DOUTOR EM CIÊNCIAS

EM

GEOFÍSICA

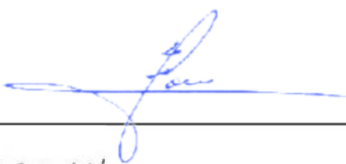

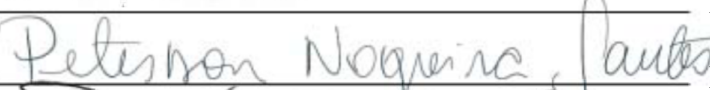
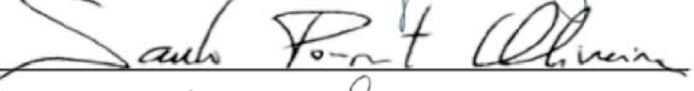
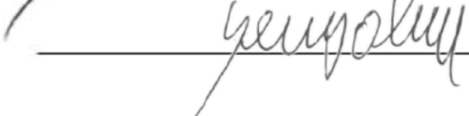
ao

Conselho Acadêmico de Ensino

da

Universidade Federal da Bahia

Comissão Examinadora

	Dr. Milton José Porsani (Orientador)
	Dr. Amin Bassrei
	Dr. Peterson Nogueira Santos
	Dr. Saulo Pomponet Oliveira
	Dr. Sérgio Adriano Moura Oliveira

Aprovada em 15 de Janeiro de 2024

Ficha catalográfica elaborada pela Biblioteca Universitária de
Ciências e Tecnologias Prof. Omar Catunda, SIBI - UFBA.

A719 Ariza Ariza, Cristian David

Solução de sistemas lineares de grande porte e computação de alto
desempenho / Cristian David Ariza Ariza. — Salvador, 2024.

167 f.: il.

Orientador: Prof. Dr. Milton J. Porsani

Tese (Doutorado - Geofísica Aplicada) - Pós-Graduação em Geofísica.
Instituto de Geociências, Universidade Federal da Bahia, 2024.

1. Sistemas lineares. 2. Computação. 3. Modelagem. I. Porsani,
Milton J. II. Título.

CDU 004

Dedico este trabalho a meus pais,
meus irmãos e ao meu amor, Rocio.
Vocês iluminam minha vida. Amo
vocês.

Resumo

Este trabalho descreve um método de solução de sistemas lineares densos de grande porte, positivo definido e bloco-estruturado, com múltiplos lados direitos, que utiliza computação paralela de alto desempenho. A solução do sistema é obtida através da recursão de Levinson generalizada que utiliza a combinação linear de soluções menores, direta e reversa, associadas aos subsistemas de menor ordem. A nova implementação é descrita para computação paralela e baseada em um algoritmo de matriz particionada. O algoritmo foi separado em duas sub-rotinas, a primeira que calcula a solução reversa e a matriz da energia dos erros para as ordens menores, e a segunda que calcula a solução recursivamente. O algoritmo foi implementado para três tipos de sistemas: sistemas de memória compartilhada, memória distribuída e para sistemas com GPU. Em cada caso os sistemas de menor ordem foram calculados usando bibliotecas apropriadas. No primeiro, foi utilizada a biblioteca OpenBLAS ou MKL, no segundo SCALAPACK e finalmente para sistemas com GPU implementamos um algoritmo OUT-OF-CORE, no qual os sistemas de menor ordem foram calculados utilizando MAGMA. Nos três casos, a solução final é comparada com a solução completa do sistema utilizando LAPACK, SCALAPACK e MAGMA, respectivamente. Nos três casos, a primeira parte do algoritmo mostrou-se mais dispendiosa computacionalmente, comparada à decomposição de Cholesky. Porém a segunda parte que calcula a solução, mostrou-se mais eficiente que a solução sucessiva de dois sistemas triangulares, quando o lado direito do sistema possui um tamanho significativo, geralmente algumas vezes o valor de N . O erro no modelo estimado não apresenta variações significativas comparado com a solução de referência. Finalmente apresentamos a utilização do algoritmo na modelagem de ondas sísmicas no domínio da frequência, que envolve a solução de grandes sistemas lineares esparsos. Estes resultados mostram uma desvantagem do algoritmo em sistemas esparsos não Toeplitz, já que aumenta o custo computacional e o consumo de memória.

Palavras Chaves: Sistema linear denso, múltiplos lados direitos, GPU, Modelagem sísmica no domínio da frequência.

Abstract

This work describes a method for solving large, positive-defined, block-structured, dense linear systems with multiple right-hand sides that uses high-performance parallel computing. The system solution is obtained through a generalized Levinson recursion that uses the linear combination of smaller forward and backward solutions associated with lower order subsystems. The new implementation is described for parallel computing and is based on a partitioned matrix algorithm. The algorithm was separated into two subroutines, the first that computes the backward solution and the error energy matrix for smaller orders, and the second that computes the solution recursively. The algorithm was implemented for three types of systems: shared memory systems, distributed memory systems, and GPU systems. In each case, the lowest order systems were calculated using appropriate libraries. In the first, the OpenBLAS or MKL library was used; in the second, SCALAPACK; and finally, for systems with GPUs, we implemented an OUT-OF-CORE algorithm, in which the lowest order systems were calculated using MAGMA. In all three cases, the final solution is compared with the complete system solution using LAPACK, SCALAPACK, and MAGMA, respectively. In all three cases, the first part of the algorithm proved to be more computationally expensive compared to the Cholesky decomposition. However, the second part that computes the solution proved to be more efficient than the successive solution of two triangular systems when the right side of the system has a significant size, generally a few times the value of N . The error in the estimated model does not present significant variations compared to the reference solution. Finally, we present the use of the algorithm in frequency-domain seismic wave modeling, which involves the solution of large, sparse linear systems. These results show a disadvantage of the algorithm in sparse non-Toeplitz systems, as it increases the computational cost and memory consumption.

Key words: Dense linear system, Multiples right-hand sides, GPU, frequency-domain seismic wave modeling.

Índice

Resumo	4
Abstract	5
Índice	6
Introdução	22
1 O problema inverso e direto	24
1.1 Formulação do problema	25
1.1.1 Problema bem e mal condicionados	27
1.1.1.1 Postulados de Hadamard	27
1.2 Sistemas lineares	28
1.2.1 Solução de sistemas lineares	30
1.2.1.1 Método dos mínimos quadrados	30
1.2.1.2 Mínimos quadrados, problema subdeterminado	32
1.2.1.3 Mínimos quadrados amortecidos MQA	33
1.2.1.4 Mínimos quadrados ponderados	34
1.2.2 Métodos de solução de sistemas lineares	35
1.3 Ambiente de Desenvolvimento	38
1.3.1 Computadores paralelos	39
1.3.1.1 Computador paralelo com multiprocessador de memória com- partilhada (SMP)	41
1.3.1.2 Computador paralelo de memória Distribuída, <i>Clusters</i>	42
1.3.1.3 Coprocessador: GPU (<i>Graphics Processing Unit</i>)	43
1.3.1.3.1 Abordagens de baixo nível, CUDA e OpenCL.	44
1.3.1.3.2 Abordagens por diretivas de compilador.	45
1.3.1.3.3 Abordagens por bibliotecas.	45
1.3.1.4 Computadores Paralelos Híbridos	46
1.3.2 Computadores Utilizados	46

1.3.2.1	Marreca	46
1.3.2.2	<i>Cluster</i> Aguia	47
1.3.2.3	<i>Cluster</i> OGUN	48
1.3.2.3.1	Nós sem GPU, partição <i>standard</i>	48
1.3.2.3.2	Nós com GPU, partição <i>gpu</i>	48
1.3.3	Bibliotecas de programação	50
1.3.3.1	BLAS	50
1.3.3.2	OpenBLAS	50
1.3.3.3	LAPACK	51
1.3.3.4	OpenMP	51
1.3.3.5	MPI	52
1.3.3.6	SCALAPACK	53
1.3.3.7	MKL	54
1.3.3.8	MAGMA	55
2	Solução De Sistemas Lineares Hermitianos, Positivos Definidos (PD)	
	E Densos De Grande Porte Com Múltiplos Lados Direitos: Contexto	
	Teórico	56
2.1	O princípio básico da recursão de Levinson para sistemas simétricos	57
2.2	Extensão do princípio de Levinson para resolver sistemas bloco-particionado de EN com N lados direitos	58
2.2.1	Recursão de Levinson para solução de um sistema bloco-Hermitiano de EN	59
2.2.1.1	Solução de EN de ordem $j = 2$	59
2.2.1.2	Solução das EN para um ordem $j + 1$	60
2.2.2	Procedimento para obter a solução reversa e a matriz de energia do erro correspondente à solução reversa	62
2.2.2.1	Solução dos subsistemas de ordem j	62
2.2.3	Algoritmo para implementação serial	65
2.2.4	Algoritmo para implementação paralela	67
2.2.5	A nova implementação proposta	69
2.2.5.1	Complexidade	73
2.2.6	Escalabilidade	75
2.2.6.1	Usando Memória compartilhada e distribuída	77
2.2.6.2	Implementação Out-of-Core, usando GPU	78

3	Solução De Sistemas Lineares Hermitianos, Positivos Definidos (PD) E Densos De Grande Porte Com Múltiplos Lados Direitos: Resultados Numéricos	82
3.1	Introdução	82
3.2	Construção sintética do problema e validação do resultado	83
3.2.1	Simulação do problema	83
3.2.2	Estabilidade do algoritmo	83
3.3	Resultados da Versão original	84
3.3.1	Resultados com a máquina MARRECA	85
3.3.1.1	Para diferentes tamanhos de matriz ($\mathbf{G}_{m=n}$), tempo vs $L=Np$	85
3.3.1.2	<i>SpeedUp</i> e eficiência	86
3.3.2	Resultados com a máquina Aguia	87
3.3.2.1	Para diferentes tamanhos de matriz ($\mathbf{G}_{m=n}$), tempo vs $L=Np$	87
3.3.2.2	<i>SpeedUp</i> e eficiência	88
3.4	Versão memória compartilhada	89
3.5	Versão memória distribuída	93
3.6	Versão OUT-OF-CORE usando GPU	96
3.6.1	Desempenho	96
3.6.1.1	GPU Tesla k40c	96
3.6.1.2	GPU Tesla P100	98
3.6.1.3	Incrementando o <i>NRHS</i> para Tesla k40c	99
3.6.2	Resíduo e erro da solução	100
3.6.3	Simulando uma GPU de 2 GB, para testar o efeito de um particionamento maior	101
3.7	Por que tem uma melhora para vários RHS?	101
4	Modelagem acústica usando diferenças finitas no domínio da frequência	105
4.1	Modelagem Acústica usando FDFD	106
4.1.1	Modelagem FDFD: Discretização da Equação de Helmholtz	106
4.2	Parametrização	109
4.2.1	Fonte sísmica: Ricker	109
4.2.2	Fronteira absorvente	109
4.2.3	Esquemas Explícitos de Diferenças Finitas	110
4.2.3.1	Operador Laplaciano de segunda ordem	110
4.2.3.2	Operador Laplaciano de quarta ordem	112
4.2.3.3	Operador Laplaciano de 9 pontos	113

4.2.4	Modelo de vp : três camadas	115
4.3	Resultados	115
5	Conclusões	121
	Agradecimentos	123
	Apêndice A Fatoração de Cholesky por blocos	125
	Apêndice B Cholesky vs Levinson	129
B.1	Sistema linear por blocos 2×2	129
B.1.1	Fatoração de Cholesky	129
B.1.2	Recursão de Levinson	130
B.1.3	Relação entre o procedimento Cholesky e Levinson	131
	Apêndice C Complexidade	132
	Apêndice D Algoritmo para a solução por blocos Cholesky usando MAGMA, Out-Of-Core	137
D.1	Sistema linear por blocos $j \times j$	137
D.2	Algoritmo Out-of-Core com multiples RHS	139
	Apêndice E Resultados adicionais: Diferentes tamanhos	140
E.1	Versão memória compartilhada	140
E.2	Versão memória distribuída	145
E.3	Versão OUT-OF-CORE usando GPU	148
E.3.1	GPU Tesla k40c: Tempo	148
E.3.2	GPU Tesla k40c: Desempenho	150
E.3.3	GPU Tesla P100: Desempenho	151
E.3.4	Incrementando o $NRHS$ para GPU Tesla k40c: Desempenho	153
E.3.5	Limitando a memória máxima a ser usada na GPU Tesla k40c a 2GB	155
	Referências Bibliográficas	159

Índice de Tabelas

1.1	Especificações técnicas principais do processador do computador MARRECA	47
1.2	Especificações técnicas principais do coprocessador Tesla K40c do computador MARRECA	47
1.3	Especificações técnicas principais dos nós da Aguia	48
1.4	Especificações técnicas principais dos nós com e sem GPU (Partições <i>standar</i> e <i>gpu</i>) do super computador OGUM	49
1.5	Especificações técnicas principais do coprocessador NVIDIA P100 NVLINK dos nós GPU do Supercomputador OGUN (SENAI SIMATEC)	49
2.1	Custo computacional detalhado do algoritmo para sistemas Hermitianos, sub-rotina 1 (Algoritmo 3) tipo Levinson. S_m representa o custo para resolver uma família de N_c sistemas associados à mesma matriz de coeficientes.	73
2.2	Custo computacional detalhado do algoritmo, sub-rotina 2 (Algoritmo 4) tipo de Levinson para solução de sistemas Hermitianos para apenas um RHS, para múltiplos RHS, basta multiplicar por NRHS.	75
C.1	Custo computacional das operações matriciais básicas, onde $\mathbf{C}^{m \times n}$, $\mathbf{G}^{m \times k}$, $\mathbf{B}^{k \times n}$, $\mathbf{S}^{n \times n}$ (simétrica), $\mathbf{E}^{n \times k}$, $\mathbf{F}^{k \times n}$ (onde o produto \mathbf{EF} é simétrico), $\mathbf{M}_{n \times NRHS}$ e $\mathbf{B}_{n \times NRHS}$, todas elas $\in \mathbf{R}$	132
C.2	Custo computacional detalhado do algoritmo para números reais, sub-rotina 1 (Algoritmo 3) tipo Levinson para solução de sistemas NE. S_m representa o custo para resolver uma família de N_c sistemas associados à mesma matriz de coeficientes.	133
C.3	Custo computacional detalhado do algoritmo, sub-rotina 2 (Algoritmo 4) tipo de Levinson para solução de sistemas NE para apenas um RHS, para múltiplos RHS multiplique por NRHS.	134
C.4	Custo computacional das operações matriciais básicas, onde $\mathbf{C}^{m \times n}$, $\mathbf{G}^{m \times k}$, $\mathbf{B}^{k \times n}$, $\mathbf{S}^{n \times n}$ (Hermitiana), $\mathbf{E}^{n \times k}$, $\mathbf{F}^{k \times n}$ (onde o produto \mathbf{EF} é hermitiano), $\mathbf{M}_{n \times NRHS}$ e $\mathbf{B}_{n \times NRHS}$, todas elas $\in \mathbf{C}$	136

E.1	Valores do particionamento (L), tamanho do bloco (Nc) e Numero máximo de RHS usango a GPU Tesla k40c (12GB) para diferentes tamanhos do sistema linear	150
E.2	Valores do particionamento (L), tamanho do bloco (Nc) e Numero máximo de RHS usango a GPU Tesla P100 (16GB) para diferentes tamanhos do sistema linear	153
E.3	Valores do particionamento (L), tamanho do bloco (Nc) e dimensão máxima do RHS quando é simulado una GPU de 2GB para diferentes tamanho do sistema linear	158

Índice de Figuras

1.1	(a) Paralelismo no chip. (b) Um coprocessador. (c) Um multiprocessador. (d) Um multicomputador. (e) Uma grade	39
2.1	Representação esquemática do algoritmo de tipo Levinson para solução de um sistema bloco-hermitiano de EN de ordem 5. A figura apresenta a distribuição espacial e a relação entre os coeficientes.	65
2.2	Representação esquemática mostrando o encadeamento das soluções menores que pode ser explorada por multiprocessadores. Ao final de L passos (L = número de partições do sistema original) a solução final é obtida.	67
2.3	Esquema mostrando como as informações por blocos da matriz \mathbf{A} são incorporadas ao algoritmo 3 em cada etapa. De cima para baixo e da esquerda para a direita. O quadrado com a linha preta grossa representa a ordem do sistema a ser resolvido e os pequenos quadrados sombreados são as informações de \mathbf{A} incorporadas no algoritmo.	71
2.4	esquema de como a informação por blocos da matriz \mathbf{A} , RHS (\mathbf{B}), fatoração de Cholesky da matriz energética do erro ($LL^*({}^1\mathbf{E}_{F_j=0,\dots,L-1})$) e solução reversa (${}^1\mathcal{F}_{j=1,\dots,L-1}$) é incorporado ao algoritmo. Observe que a matriz solução \mathcal{M}_L , vai progressivamente do branco ao preto, representando a progressão que é necessária para obter a solução final.	72
2.5	Operações de ponto flutuante (FLOP) normalizado para decompor (Algoritmo 3) um sistema $N \times N$ ($N = 50000$) em função do particionamento L	74
2.6	Operações de ponto flutuante (FLOP) normalizado para resolver (Algoritmo 4) um sistema de N ($N = 50000$) parâmetros e $NRHS$ lados direitos em função do particionamento L	76
3.1	Tempo total (segundos) em função do tamanhos de matriz \mathbf{A} de $N \times N$. <i>Th</i> faz referência aos <i>Thread(s)</i> . O particionamento L é igual ao número de <i>Thread(s)</i>	85

3.2	Tempo total em função do tamanhos de matriz \mathbf{A} de $N \times N$ em escala logarítmica base 10. Th faz referência aos $Thread(s)$. O particionamento L é igual ao número de $Thread(s)$	85
3.3	Aumento da velocidade total em função do tamanho da matriz, para diferentes níveis de particionamento ($L = th$), $L =$ número de processadores = ordem de particionamento	86
3.4	Eficiência em função do tamanho da matriz, para diferentes níveis de particionamento ($L=th$), $L =$ número de processadores.	86
3.5	Tempo total em função do tamanhos de matriz \mathbf{A} de $N \times N$. O particionamento L é igual ao número de cpu usados	87
3.6	Tempo total em função do tamanhos de matriz \mathbf{A} de $N \times N$ em escala logarítmica base 10. O particionamento L é igual ao número de cpu usados	88
3.7	Aumento da velocidade total em função do tamanho da matriz, para diferentes tamanhos de particionamento (L). O particionamento L é igual ao número de processadores.	88
3.8	Eficiência em função do tamanho da matriz, para diferentes tamanhos de particionamento (L). O particionamento L é igual ao número de processadores.	89
3.9	Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória compartilhada para um sistema quando $N = NRHS = 60k$ e variando o particionamento ($L = 2, 3, 4, 5, 6$), usando um CPU de 80 $threads$ (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	90
3.10	Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 60k$ ($L = 2$ e $N_c = 30k$) e dimensões diferentes de RHS ($12k$ a $180k$), usando um CPU de 40 núcleos , 80 $threads$ (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	91
3.11	Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 2$ e $N_c = 22.5k$) e dimensões diferentes de RHS ($9k$ a $405k$), usando um CPU de 40 núcleos , 80 $threads$ (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	92

- 3.12 Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS ($9k$ a $630k$), usando um CPU de 40 núcleos, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 92
- 3.13 Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória distribuída para um sistema quando $N = NRHS = 30k$ e variando o particionamento ($L = 2, 3, 4, 5$), usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 93
- 3.14 Comparação entre o algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dpotrf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS ($3k$ a $60k$), usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 94
- 3.15 Comparação entre o algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dpotrf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS ($3k$ a $60k$), usando um CPU de 300 núcleos (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 95
- 3.16 Comparação entre o algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dpotrf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS , usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 96
- 3.17 Comparação entre magmaf (magmaf dpotrf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 90k$ ($L = 7$ e $N_c = 12864$) e dimensões diferentes de RHS ($9k$ a $270k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 97
- 3.18 Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 90K$ e diferentes valores de $NRHS$ usando a GPU tesla k40c 98

3.19	Comparação entre <i>magmaf</i> (<i>magmaf</i> DPOTRF + <i>magmaf</i> DPOTRS) e nossas implementações para um sistema quando $N = 70k$ ($L = 5$ e $N_c = 14080$) e dimensões diferentes de RHS ($14k$ a $210k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes	99
3.20	Comparação entre <i>magmaf</i> (<i>magmaf</i> dpotrf + <i>magmaf</i> dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 75k$ ($L = 6$ e $N_c = 12544$) e dimensões diferentes de RHS ($15k$ a $615k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	100
3.21	Comparação do Resíduo e ou Erro entre a solução magma (MAGMA) e nossas implementações (PLS GPU) para um sistema quando $N = NRHS$ (de $40k$ a $100k$), usando um NVIDIA k40c de 12 GB.	100
3.22	Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L. Comparação entre <i>magmaf</i> (<i>magmaf</i> dpotrf + <i>magmaf</i> dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 50k$ ($L = 8$ e $N_c = 6272$) e dimensões diferentes de RHS ($10k$ a $150k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	101
3.23	Comparação entre o desempenho da fatoração de Cholesky (LU, usando DPOTRF do LAPACK), resolve um sistema de equações lineares usando fatoração de Cholesky (FB, substituição direta e reversa usando DPOTRS do LAPACK) para a matriz $N \times N$ e $NRHS = N$. Desempenho para multiplicação de matrizes, com implementação explícita, para tamanho de matriz $N \times N$	102
3.24	Comparação entre LAPACK (DPOTRS, substituição direta e reversa) e nossas implementações (usando $N_c = 500$) para dimensões diferentes de NRHS para NE $N \times N$ quando $N = 10000$	102
3.25	Comparação entre LAPACK (DPOTRS, substituição direta e reversa) e nossas implementações (usando $N_c = 1000$) para dimensões diferentes de NRHS para NE $N \times N$ quando $N = 10000$	103
3.26	Comparação entre LAPACK (DPOTRF+DPOTRS) e nossas implementações para sistemas de diferentes tamanhos N e $NRHS = N$	103
3.27	Comparação entre LAPACK (DPOTRF+DPOTRS) e nossas implementações para sistemas de diferentes tamanhos N e $NRHS = N$	104
4.1	Fluxograma para obtenção de sismogramas no domínio do tempo a partir da modelagem sísmica no domínio da frequência. Adaptado de Revelo (2015).	108

4.2	Representação de uma <i>wavelet</i> tipo Ricker com uma frequência pico igual a $15Hz$ e seu correspondente espectro de amplitude	109
4.3	Modelo da borda absorvente (ABC), onde n_{x_b} e n_{z_b} indica o número de pontos da borda em as direções x e z	110
4.4	Esquema de diferenças finitas para o operador Laplaciano de segunda ordem	111
4.5	Matriz de impedância usando um operador de segunda ordem. O modelo $n_{xe} = 10$ $n_{ze} = 10$, com $\Delta x = \Delta z = \Delta h$	112
4.6	Esquema de diferenças finitas para o operador Laplaciano de segunda ordem, com $\Delta x = \Delta z = \Delta h$	113
4.7	Matriz impedância usando um operador de segunda ordem, o modelo $n_{xe} = 10$ $n_{ze} = 10$, com $\Delta x = \Delta z = \Delta h$	113
4.8	(a) Operador de 5 pontos convencional, (b) operador de 5 pontos rotacionado 45° e (c) esquema compacto de 9 pontos. Tomada de Revelo (2015)	114
4.9	Padrão esparsa da matriz impedância do operador compacto de 9 pontos, supondo um modelo $n_{xe} = 10$ $n_{ze} = 10$	115
4.10	Modelo de velocidade 3 camadas, $n_z = 81, n_x = 81$ $dx = dz = 20m$	116
4.11	Arranjo do tipo <i>Single-ended spread</i> usado para a obter o tiro simulado	116
4.12	(a) <i>Shot</i> modelado referencia, (b) <i>shot</i> modelado usando $L=121$, PLS	118
4.13	(a) <i>Shot</i> modelado referencia, (b) <i>shot</i> modelado usando $L=121$, PLS	119
4.14	(a) <i>Shot</i> modelado referencia, (b) <i>shot</i> modelado usando $L=121$, PLS	120
C.1	Operações de ponto flutuante (FLOP) normalizado para decompor (Algoritmo 3) um sistema $N \times N$ ($N = 50000$) em função do particionamento L	134
C.2	Operações de ponto flutuante (FLOP) normalizado para resolver (Algoritmo 4) um sistema de N ($N = 50000$) parâmetros e $NRHS$ lados direitos em função do particionamento L	135
E.1	Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória compartilhada vários tamanhos de N , em todos eles $NRHS = N$ e variando o particionamento, usando um CPU de 80 <i>threads</i> (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	141
E.2	Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória compartilhada vários tamanhos de N , em todos eles $NRHS = 3N$ e variando o particionamento ($L = 2, 3, 4, 5, 6$), usando um CPU de 80 <i>threads</i> (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes	141

- E.3 Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $3N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes. 142
- E.4 Comparação de desempenho (GFLOP/s) entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $3N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes. 143
- E.5 Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $11N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes. 143
- E.6 Comparação de desempenho (GFLOP/s) entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $11N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes. 144
- E.7 Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k$ e $N = 30k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $20N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes. 144
- E.8 Comparação de desempenho (GFLOP/s) entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k$ e $N = 30k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $20N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes. 145

- E.9 Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória distribuída, e diferente dimensões ($N = NRHS = 10k$, $N = NRHS = 20k$, $N = NRHS = 25k$ e $N = NRHS = 30k$), variando o particionamento ($L = 2, 3, 4, 5$), usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 146
- E.10 Comparação entre o algoritmo de referência Scalapack (Scl dporf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para sistemas de memória distribuída, e diferente dimensões ($N = 10k$, $N = 20k$, $N = 25k$ e $N = 30k$) e números de tamanhos diferentes de RHS , usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 147
- E.11 Desempenho do algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dporf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 10k$ ($L = 2$ e $N_c = 5k$) e números de tamanhos diferentes de RHS , usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 147
- E.12 Desempenho do algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dporf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e números de tamanhos diferentes de RHS ($3k$ a $60k$), usando um CPU de 300 núcleos (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 148
- E.13 Comparação entre magmaf (magmaf dporf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 4$ and $N_c = 11264$) e números de tamanhos diferentes de RHS ($4.5k$ to $135k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 148
- E.14 Comparação entre magmaf (magmaf dporf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 60k$ ($L = 5$ and $N_c = 12k$) e números de tamanhos diferentes de RHS ($6k$ to $180k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 149

- E.15 Comparação entre magmaf (magmaf dporf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 75k$ ($L = 6$ and $N_c = 12512$) e números de tamanhos diferentes de RHS ($7.5k$ to $225k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 149
- E.16 Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 45k$ e diferentes valores de $NRHS$ usando a GPU tesla k40c 150
- E.17 Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 60k$ e diferentes valores de $NRHS$ usando a GPU tesla k40c 150
- E.18 Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 75k$ e diferentes valores de $NRHS$ usando a GPU tesla k40c 151
- E.19 Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 50k$ ($L = 3$ e $N_c = 16768$) e números de tamanhos diferentes de RHS ($10k$ a $150k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes 151
- E.20 Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 60k$ ($L = 4$ e $N_c = 15104$) e números de tamanhos diferentes de RHS ($12k$ a $180k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes 152
- E.21 Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 808$ ($L = 5$ e $N_c = 16000$) e números de tamanhos diferentes de RHS ($16k$ a $240k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes 152
- E.22 Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 909$ ($L = 6$ e $N_c = 15104$) e números de tamanhos diferentes de RHS ($18k$ a $270k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes 153

- E.23 Comparação entre magmaf (magmaf dporf + magmaf dptrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 4$ e $N_c = 11264$) e números de tamanhos diferentes de RHS ($9k$ a $513k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 154
- E.24 Comparação entre magmaf (magmaf dporf + magmaf dptrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 60k$ ($L = 5$ e $N_c = 12032$) e números de tamanhos diferentes de RHS ($12k$ a $684k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 154
- E.25 Comparação entre magmaf (magmaf dporf + magmaf dptrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 90k$ ($L = 7$ e $N_c = 12928$) e números de tamanhos diferentes de RHS ($18k$ a $450k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 155
- E.26 Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L. Comparação entre magmaf (magmaf dporf + magmaf dptrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 25k$ ($L = 4$ e $N_c = 6272$) e números de tamanhos diferentes de RHS ($5k$ a $75k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 156
- E.27 Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L. Comparação entre magmaf (magmaf dporf + magmaf dptrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 5$ e $N_c = 6016$) e números de tamanhos diferentes de RHS ($6k$ a $90k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 156
- E.28 Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L. Comparação entre magmaf (magmaf dporf + magmaf dptrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 35k$ ($L = 6$ e $N_c = 5888$) e números de tamanhos diferentes de RHS ($7k$ a $105k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 157

- E.29 Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L. Comparação entre magma_f (magma_f dporf + magma_f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 40k$ ($L = 7$ e $N_c = 5760$) e números de tamanhos diferentes de RHS ($8k$ a $120k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 157
- E.30 Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L. Comparação entre magma_f (magma_f dporf + magma_f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 8$ e $N_c = 5632$) e números de tamanhos diferentes de RHS ($9k$ a $135k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes 158

Introdução

O desafio proposto consiste no desenvolvimento de uma metodologia computacionalmente eficiente e robusta para a solução de sistemas lineares de grande porte, aplicável à modelagem numérica em geofísica. A solução de grandes sistemas lineares está normalmente associada à modelagem direta e inversa de dados geofísicos. Na modelagem direta tais sistemas resultam da discretização da equação da onda quando realizadas através de métodos de diferenças finitas ou de elementos finitos (2D e 3D). Na modelagem inversa eles ocorrem associados às estimativas de parâmetros, realizadas através dos métodos de Newton e Gauss-Newton, na busca de modelos geofísicos que expliquem melhor os dados. Tanto na modelagem direta quanto a inversa, a obtenção das soluções representam normalmente um alto custo computacional.

Outra limitação frequentemente encontrada na modelagem e inversão 2D e 3D seria que o volume de dados é tão grande que as matrizes associadas aos sistemas lineares (ou linearizados) a serem resolvidos, não cabem na memória de um único computador. Porsani et al. (2010), propuseram um método para solução de sistemas lineares de grande porte que efetua a solução em blocos de parâmetros resolvidos de forma independente em processadores e/ou máquinas distribuídas. O método consiste numa generalização do método clássico de solução recursiva de sistemas simétricos bloco-toeplitz, para sistemas bloco-particionados complexos e não necessariamente simétricos. A estrutura do algoritmo recursivo evolui partindo da solução de subsistemas menores para maiores, sendo naturalmente apropriada para tirar proveito da computação de alto desempenho através de máquinas com múltiplos processadores. Neste projeto é proposto a implementação, teste e validação da nova metodologia para solução de sistemas lineares hermitianos, positivos definidos (PD) e densos de grande porte com múltiplos lados direitos.

No capítulo 1 a forma de introdução, mostraremos um breve resumo sobre a solução de sistemas lineares além de uma introdução à computação paralela e ao ambiente de desenvolvimento utilizado neste trabalho com informações básicas sobre equipamentos e bibliotecas utilizadas.

No capítulo 2 são descritos os fundamentos teóricos e é apresentado um novo algoritmo paralelo por blocos para a solução de sistemas Hermitianos positivos definidos e densos com vários lados direitos, onde a solução do sistema é obtida recursivamente - usando a recursão de Levinson - combinando linearmente as soluções de menor ordem que estão relacionadas aos subsistemas *forward* e *backward* das equações normais de menor ordem. Neste capítulo são apresentados os algoritmos originais serial e paralelo, assim como suas vantagens e desvantagens são discutidas. Abordando precisamente a sua principal desvantagem (liberação ou desuso da capacidade de processamento à medida que o algoritmo avança), é apresentada a nova implementação que permite a utilização de toda a capacidade de processamento em todos os momentos. Destina-se ao uso em três sistemas de computação paralelos diferentes: memória compartilhada, memória distribuída e com o uso de unidades de processamento gráfico (GPU) adicionais.

No capítulo 3 e no apêndice E são apresentados os resultados numéricos das diferentes versões descritas no capítulo 2. Estes resultados são comparados com um algoritmo de referência, que permite validar a proposta.

No capítulo 4 apresentamos a utilização do algoritmo em um problema específico de geofísica, neste caso modelagem sísmica na frequência. Este problema específico envolve a solução de grandes sistemas lineares esparsos. Estes resultados mostram uma desvantagem do algoritmo em sistemas não Toeplitz, pois dada a forma recursiva como a solução é obtida, implica que as matrizes auxiliares (soluções de ordem inferior) rapidamente se tornam em matrizes densas, o que aumenta o custo computacional e o consumo de memória.

1

O problema inverso e direto

Problemas inversos surgem em muitas aplicações em ciência e engenharia. O termo “problema inverso” é geralmente entendido como o problema de encontrar uma propriedade física específica, ou propriedades, do meio em investigação usando medidas indiretas. Exemplos de problemas inversos podem ser encontrados em vários campos no processamento de imagens médicas (Bertero e Boccacci, 1998; Louis, 1992; Engl et al., 1996; Arridge e Hebden, 1997; Arridge, 1999) e várias áreas da geofísica, incluindo a exploração de minerais e petróleo (Kearey et al., 2002; Menke, 2012; Russell, 1988; Aster et al., 2011)

O estudo do interior da Terra na Geofísica consiste em fazer medições, em geral, de alguma grandeza física segundo algum método geofísico, realizadas com equipamentos modernos na superfície da terra ou próxima a ela. Estas medições são influenciadas pela distribuição interna das propriedades físicas. A análise pode revelar como é que as propriedades físicas (susceptibilidade magnética, densidade, condutividade elétrica, etc.) do interior da Terra variam vertical e lateralmente (Kearey et al., 2002). Grande parte do conhecimento terrestre, abaixo das profundidades que se podem atingir por intermédio de poços (muito custoso e que fornecem apenas informação local), é proveniente de observações geofísicas. A determinação da geometria e distribuição das propriedades físicas da subsuperfície são estimadas por meio de medição, análise e interpretação dos dados geofísicos. Isto constitui o chamado problema inverso na geofísica, ou seja, a partir de dados, (observações, medidas geofísicas), e um princípio geral, teoria ou modelo quantitativo, determinam estimativas dos parâmetros do modelo que explicam os dados geofísicos medidos. Em contraposição, a modelagem direta (problema direto) é um procedimento no qual um conjunto único de observações geofísicas sintéticas é calculado (resposta geofísica) com base em algum modelo geofísico e um conjunto de condições específicas relevantes (distribuição de propriedade física

conhecida) que representam os parâmetros da subsuperfície. Além de ser uma parte inerente do processo de inversão, a modelagem direta desempenha um papel importante na concepção de pesquisas geofísicas e no teste de cenários geológicos (Tarantola, 2005). Em resumo, no problema direto, os valores dos parâmetros (geométricos e físicos) do modelo são os dados e os valores das quantidades observadas são as incógnitas, enquanto no problema inverso, os dados são as medidas geofísicas e as incógnitas são os valores dos parâmetros do modelo. A análise e interpretação dos dados geofísicos são realizadas com o auxílio de modernos pacotes computacionais.

Observe que o papel da teoria inversa é fornecer informações quantitativas sobre os valores desconhecidos dos parâmetros que descrevem um determinado modelo geofísico, mas não fornecer o próprio modelo. No entanto, a teoria inversa pode muitas vezes fornecer um meio para avaliar um determinado modelo ou de discriminar entre vários modelos possíveis (Menke, 2012).

Os parâmetros do modelo que encontramos na teoria inversa variam de quantidades numéricas discretas, a funções contínuas, de uma ou mais variáveis. Na geofísica, geralmente os dados coletados e o número de parâmetros do modelo, são representados como um conjunto finito de valores numéricos discretos, isto é a teoria inversa discreta. Na prática, é possível o estudo de funções contínuas uma vez que geralmente podem ser adequadamente aproximadas por um número finito de parâmetros discretos. As parametrizações de funções contínuas são sempre propriedades aproximadas e, em certa medida, arbitrárias, que lançam uma certa quantidade de imprecisão na teoria. No entanto, a teoria inversa discreta é um bom ponto de partida para o estudo da teoria inversa em geral, uma vez que se baseia principalmente na teoria de vetores e matrizes, em vez da teoria um pouco mais complicada de funções e operadores contínuos. Além disso, a aplicação cuidadosa da teoria inversa discreta pode, muitas vezes, proporcionar uma compreensão considerável do problema, mesmo quando aplicado a problemas que envolvem parâmetros contínuos (Menke, 2012).

1.1 Formulação do problema

Na maioria dos problemas inversos os dados são simplesmente uma lista de valores numéricos, sendo representados por um vetor coluna. Se as M medições forem realizadas em um experimento particular, por exemplo, pode-se considerar essas medidas como os M elementos de um vetor \mathbf{d} ,

$$\mathbf{d} = [d_1, \dots, d_M]^T \quad (1.1)$$

onde T significa transposição.

O objetivo da análise de dados é obter conhecimento através do exame sistemático de dados. Analisamos os dados para inferir, da melhor forma possível, os valores das quantidades numéricas - parâmetros do modelo (Menke, 2012). Os parâmetros do modelo são escolhidos para serem significativos; ou seja, eles são escolhidos para capturar o caráter essencial dos processos que estão sendo estudados. Os parâmetros do modelo podem ser representados como os N elementos de um vetor \mathbf{m} ,

$$\mathbf{m} = [m_1, \dots, m_N]^T. \quad (1.2)$$

A premissa básica é que os parâmetros do modelo e os dados estão de alguma forma relacionados. Essa relação é chamada de modelo quantitativo (ou modelo, ou teoria). Normalmente, o modelo assume a forma de uma ou mais fórmulas que os dados e os parâmetros do modelo devem seguir. De um modo geral, os parâmetros de dados e modelos podem estar relacionados por uma ou mais equações implícitas, da forma

$$\begin{aligned} F_1(\mathbf{d}, \mathbf{m}) &= 0 \\ F_2(\mathbf{d}, \mathbf{m}) &= 0 \\ &\vdots \\ F_k(\mathbf{d}, \mathbf{m}) &= 0 \end{aligned} \quad (1.3)$$

onde k é o número de equações.

Essas equações implícitas podem ser escritas de forma compacta como a equação vetorial

$$\mathbf{F}(\mathbf{d}, \mathbf{m}) = 0, \quad (1.4)$$

que resume o que é conhecido sobre como os dados medidos, e os parâmetros do modelo desconhecido, estão relacionados.

No problema direto, dado um modelo \mathbf{m} , usando as equações implícitas é possível prever os dados \mathbf{d} por:

$$\mathbf{d} = \mathbf{f}(\mathbf{m}). \quad (1.5)$$

No problema inverso, o objetivo é resolver, ou “inverter”, essas equações para obtenção dos parâmetros do modelo, a partir dos dados observados,

$$\mathbf{m} = \mathbf{f}^{-1}(\mathbf{d}). \quad (1.6)$$

Os problemas inversos são classificados em problemas inversos lineares ou não-lineares (Menke, 2012). Problemas inversos lineares fazem referência a problemas em que a relação entre as medidas e os parâmetros do modelo obedecem a um relacionamento linear: $\mathbf{d} \propto \mathbf{m}$.

Os problemas inversos não-lineares surgem quando as medidas (dados) \mathbf{d} , e parâmetros do modelo, são descritas por um relacionamento não-linear, $\mathbf{d} = \mathbf{f}(\mathbf{m})$, onde \mathbf{f} é uma função não linear em \mathbf{m} . O operador $\mathbf{f}(\mathbf{m})$, que relaciona o modelo \mathbf{m} e os dados \mathbf{d} , contém as leis físicas conhecidas para governar o processo. Dependendo da aplicação, $\mathbf{f}(\mathbf{m})$ pode ser a discretização de uma Equação Diferencial Ordinária (ODE), uma Equação Diferencial Parcial (PDE) ou, se o problema for linear, um sistema de equações algébricas.

1.1.1 Problema bem e mal condicionados

No início do século passado, o matemático francês Jacques Hadamard (Hadamard, 1932) definiu um problema matematicamente bem condicionado (*Well-posed*), o que agora é conhecido como a definição clássica, como sendo aquele que cumpre as seguintes condições:

1.1.1.1 Postulados de Hadamard

- Existência: Para cada dado, a solução existe.
- Unicidade: Para cada dado, a solução é única.
- Estabilidade: A solução depende continuamente dos dados.

Se alguma dessas condições não for atendida, problema é classificado como mal condicionado (*Ill-posed*). Infelizmente, seguindo os anteriores postulados, todos os problemas inversos geofísicos (a maioria da matemática e a maioria dos problemas das ciências naturais) são mal condicionados (Zhdanov, 2002), porque suas soluções não são únicas (vários modelos podem explicar igualmente o mesmo dado) ou são instáveis (uma pequena perturbação dos dados corresponde a uma perturbação arbitrariamente grande da solução).

A estimação de parâmetros nos problemas inversos é geralmente um problema difícil de resolver, porque não podemos garantir a existência da solução, sua unicidade ou sua estabilidade.

Problemas de inexistência da solução podem surgir quando o funcional matemático que reflete a física está incompleto e, portanto, não podemos gerar um modelo para explicar os dados.

Problemas sem solução única (não unicidade), também conhecidos como problemas sub-determinados, aparecem comumente com problemas discretos quando se quer a inversão de matrizes de posto deficiente, revelando um espaço nulo não trivial. Assim, qualquer combinação linear de modelos no espaço nulo do modelo não altera os dados levando a uma

situação em que um grande número de modelos oferece a mesma solução, por exemplo, discretização das propriedades da terra, que são funções contínuas, pode causar diversidade de soluções. Dada uma série de soluções possíveis, é preciso decidir qual escolher. Se possível, é necessário incluir informações adicionais que podem suavizar significativamente o modelo, ou reduzir o seu contraste, ou compará-lo com um modelo a priori.

Finalmente, as instabilidades na inversão surgem muitas vezes devido ao fato de que uma pequena alteração na medição leva a grandes mudanças no modelo, ou seja, os dados reais são sempre contaminados por ruído e os erros nos dados são propagados em erros na estimativa dos parâmetros do modelo seja por uma relação não-linear entre o modelo e os dados ou instabilidade numérica quando resolvidas com precisão finita. Além disso, não queremos produzir um modelo que reproduza exatamente os dados observados, porque isso também reproduzirá o ruído. Portanto, os problemas inversos caracterizam-se por serem mal-postos.

Hadamard considerava que um problema matemático mal colocado não era física e/ou matematicamente significativo. Felizmente, posteriormente foi descoberto que a opinião de Hadamard estava errada: problemas mal colocados são física e matematicamente significativos e podem ser resolvidos (Zhdanov, 2002). Em meados do século XX, o matemático russo Andrei N. Tikhonov desenvolveu os fundamentos da teoria das soluções de problemas mal condicionados. Ele introduziu um método de regularização na solução de um problema inverso que era baseado em uma aproximação de um problema mal condicionado por uma série de problemas bem condicionados (Tikhonov, 1963; Tikhonov e Arsenin, 1977; Tikhonov et al., 2013; Menke, 2012).

As estratégias de solução para resolver problemas inversos serão, portanto, dependentes e específicas da classificação do problema: linear, não linear, mal posto, discreta ou contínua (Tarantola, 2005; Snieder e Trampert, 2000). Após fazer uma breve descrição das categorias de classificação, focaremos nas estratégias para resolução de problemas lineares que representam problemas inversos discretos.

1.2 Sistemas lineares

Os problemas inversos mais simples e melhor compreendidos são aqueles que podem ser representados com a equação linear explícita da forma:

$$\mathbf{f}(\mathbf{d}, \mathbf{m}) = 0 = \mathbf{d} - \mathbf{Gm} \quad (1.7)$$

ou seja

$$\mathbf{G}\mathbf{m} = \mathbf{d} \quad (1.8)$$

onde \mathbf{G} é um operador matricial, que mapeia funções no espaço do modelo para funções no espaço de dados, $\mathbf{G} : \mathcal{M} \rightarrow \mathcal{D}$, de dimensão $M \times N$ que relaciona os M dados observados aos N parâmetros do modelo

$$\mathbf{G} = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1N} \\ g_{21} & g_{22} & \cdots & g_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ g_{M1} & g_{M2} & \cdots & g_{MN} \end{bmatrix} \quad (1.9)$$

onde os elementos g_{ij} podem ser números reais ou complexos. A equação 1.8 é uma forma compacta do seguinte sistema de equações lineares com M equações e N incógnitas

$$\begin{aligned} g_{11}m_1 + g_{12}m_2 + \cdots + g_{1N}m_N &= d_1 \\ g_{21}m_1 + g_{22}m_2 + \cdots + g_{2N}m_N &= d_2 \\ &\vdots \\ g_{M1}m_1 + g_{M2}m_2 + \cdots + g_{MN}m_N &= d_M \end{aligned} \quad (1.10)$$

o vetor $\mathbf{m} = [m_1, \dots, m_N]^T$ que satisfaça o sistema de equações 1.8 é chamado de solução do sistema linear.

Para resolver o problema inverso, é necessário que \mathbf{d} seja alcançável, isto é equivalente a dizer que existe uma solução (modelo estimado \mathbf{m}^{est}). Se o espaço nulo de \mathbf{G} estiver vazio ($\mathcal{N}(\mathbf{A}) = 0$), a solução é única, de modo que os dados previstos $\mathbf{d}^{pre} = \mathbf{G}\mathbf{m}^{est}$, se tornaram iguais aos dados observados \mathbf{d}^{obs} . Na geofísica isto geralmente não acontece. A diferença entre os dados calculados e os dados observados é chamada de erro de predição $\mathbf{e} = \mathbf{d}^{obs} - \mathbf{d}^{pre}$.

A depender das características do operador \mathbf{G} o sistema poderá ter:

- Uma única solução, dada pelo vetor \mathbf{m} , onde $M = N$ e posto completo ($r = M = N$), ou seja todas as equações são linearmente independentes (todas as linhas ou colunas de \mathbf{G} são linearmente independentes). Este problema é chamado de determinado e se \mathbf{G} é inversível, o modelo estimado é $\mathbf{m}^{est} = \mathbf{G}^{-1}\mathbf{d}^{obs}$
- Infinitas soluções, ou seja, muitos vetores \mathbf{m} satisfazem a equação 1.8 (o erro de predição é zero), Isto acontece quando o sistema de equações não fornece informações suficientes para determinar exclusivamente todos os parâmetros do modelo, o problema é dito subdeterminado ($M < N$). Para obter uma única solução, é possível incluir restrições adicionais.

- Sem solução, ou seja, um sistema sobre-determinado ($M > N$), isto acontece quando há muita informação contida na equação $\mathbf{G}\mathbf{m} = \mathbf{d}$ para que ela possua uma solução exata, portanto, não é possível obter um vetor \mathbf{m}^{est} tal que $\mathbf{d}^{pre} = \mathbf{d}^{obs}$ ($\mathbf{e} \neq 0$). Neste caso é possível empregar mínimos quadrados para selecionar a “melhor” solução aproximada.

Para diferenciar entre os problemas subdeterminados que têm erro de predição igual a zero (referidos anteriormente), com aqueles problemas subdeterminados que têm erro de predição diferente de zero, é usado o termo misto-determinado (Menke, 2012). Um problema misto-determinado é aquele em que alguns dos componentes da solução estão sobre-determinados, enquanto outros componentes estão sub-determinados, então o problema tem erros devido a medidas inconsistentes e parâmetros do modelo que não podem ser determinados a partir dos dados. Uma vez que o problema é pelo menos parcialmente subdeterminado, geralmente haverá algum erro entre os dados calculados a partir de um modelo $\mathbf{d}^{pre} = \mathbf{G}\mathbf{m}^{pre}$ e os dados observados \mathbf{d}^{obs} .

1.2.1 Solução de sistemas lineares

A matriz \mathbf{G} só admite inversa se for quadrada e não singular. Para os outros tipos de matrizes, podemos usar outras técnicas para obter uma solução aceitável. A seguir serão descritas algumas delas.

1.2.1.1 Método dos mínimos quadrados

O método dos mínimos quadrados é usado para fornecer uma solução aproximada em problemas que não têm uma solução exata, como por exemplo um problema sobre-determinado, quando o número de equações é maior que o número de incógnitas. A solução obtida é tal que o somatório dos quadrados dos erro de predição é mínimo. A partir do modelo linear, o dado calculado é

$$\mathbf{d}^{pre} = \mathbf{G}\mathbf{m} \quad (1.11)$$

onde \mathbf{m} faz referência a ou modelo estimado (\mathbf{m}_{est}), e lembrando que o erro de predição, a diferença/resíduo entre os dados medidos e os dados calculados, é dado por:

$$\mathbf{r} = \mathbf{d}^{obs} - \mathbf{G}\mathbf{m} = \mathbf{d} - \mathbf{G}\mathbf{m}. \quad (1.12)$$

O somatório do quadrado dos error de todos os dados é calculado pela função escalar, que é chamada de função objetivo:

$$S(\mathbf{m}) = \mathbf{r}^T \mathbf{r}. \quad (1.13)$$

A solução que procuramos ($\mathbf{m} = \mathbf{m}_{est}$), deve ser tal que o valor de $S(\mathbf{m})$ seja mínimo, para isso, a equação 1.13 pode ser escrita como:

$$S(\mathbf{m}) = (\mathbf{d} - \mathbf{G}\mathbf{m})^T(\mathbf{d} - \mathbf{G}\mathbf{m}), \quad (1.14)$$

onde T denota a transposta. Neste capítulo, assume-se que todos os vetores e matrizes são de valor real. No caso complexo, usa-se a transposição conjugada ou adjunto (também é conhecida como a matriz adjunta, Hermitiana adjunta ou Hermitiana-conjugada). A equação 1.14 pode ser expandida como:

$$\begin{aligned} S(\mathbf{m}) &= (\mathbf{d}^T - \mathbf{m}^T \mathbf{G}^T)(\mathbf{d} - \mathbf{G}\mathbf{m}) \\ &= \mathbf{d}^T \mathbf{d} - \mathbf{d}^T \mathbf{G}\mathbf{m} - \mathbf{m}^T \mathbf{G}^T \mathbf{d} + \mathbf{m}^T \mathbf{G}^T \mathbf{G}\mathbf{m} \end{aligned} \quad (1.15)$$

Observe que os quatro termos na Eq. 1.15 são escalares, além disso que o escalar $\mathbf{m}^T \mathbf{G}^T \mathbf{d}$ é o transposto de $\mathbf{d}^T \mathbf{G}\mathbf{m}$, e portanto $\mathbf{m}^T \mathbf{G}^T \mathbf{d} = \mathbf{d}^T \mathbf{G}\mathbf{m}$. A equação 1.15 poder ser reescrita como:

$$S(\mathbf{m}) = \mathbf{d}^T \mathbf{d} - 2\mathbf{d}^T \mathbf{G}\mathbf{m} + \mathbf{m}^T \mathbf{G}^T \mathbf{G}\mathbf{m}. \quad (1.16)$$

Para minimizar $S(\mathbf{m})$ em relação a \mathbf{m} , podemos encontrar onde a derivada é zero para \mathbf{m} ou \mathbf{m}^T , mas \mathbf{m}^T é mais conveniente. A derivada da expressão anterior torna-se então

$$\frac{\partial S(\mathbf{m})}{\partial \mathbf{m}^T} = 2\mathbf{G}^T \mathbf{G}\mathbf{m} - 2\mathbf{G}^T \mathbf{d} = 0 \quad (1.17)$$

produzindo

$$\mathbf{G}^T \mathbf{d} = \mathbf{G}^T \mathbf{G}\mathbf{m}. \quad (1.18)$$

Observe que a quantidade $\mathbf{G}^T \mathbf{G}$ é uma matriz quadrada $N \times N$ e que multiplica um vetor \mathbf{m} de comprimento N . A quantidade $\mathbf{G}^T \mathbf{d}$ é também um vetor de comprimento N . Esta equação é, portanto, uma equação de matriz quadrada para os parâmetros do modelo desconhecido. Presumindo que $(\mathbf{G}^T \mathbf{G})^{-1}$ existe, então a estimativa para os parâmetros do modelo que minimiza $S(\mathbf{m})$ é:

$$\mathbf{m} = \mathbf{m}_{est} = (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T \mathbf{d} = \mathbf{G}^{-g} \mathbf{d} \quad (1.19)$$

\mathbf{m}_{est} é a solução de mínimos quadrados do sistema linear, $\mathbf{G}\mathbf{m} = \mathbf{d}$. A matriz \mathbf{G}^{-g} é referida como a inversa generalizada, e a forma exata do inverso generalizado depende do problema em questão. A aplicação direta da equação 1.19, geralmente é usada quando as dimensões de \mathbf{G} são pequenas, (por exemplo, N e M menor do que algumas centenas), porém para problemas de maiores dimensões, o custo computacional do cálculo de $(\mathbf{G}^T \mathbf{G})^{-1}$ pode ser proibitivo. Neste caso, normalmente são utilizados métodos iterativos, ou recursivos, que não requerem explicitamente a matriz inversa, a exemplo do algoritmo gradiente biconjugado

(Menke, 2012). No caso em que $\mathbf{G}^T\mathbf{G}$ seja uma matriz singular, não é possível obter a inversa $(\mathbf{G}^T\mathbf{G})^{-1}$. Nestes casos, pode-se usar a técnica dos mínimos quadrados amortecidos (MQA).

1.2.1.2 Mínimos quadrados, problema subdeterminado

Suponha que um problema inverso $\mathbf{G}\mathbf{m} = \mathbf{d}$ tenha sido identificado como puramente subdeterminado. Por simplicidade, suponha que existam menos equações do que os parâmetros do modelo desconhecido, isto é, $M < N$, e que não existem inconsistências nessas equações. Portanto, é possível encontrar mais de uma solução para a qual o erro de predição, a diferença/resíduo entre os dados medidos e os dados calculados, \mathbf{r} , é zero. Embora os dados forneçam informações sobre os parâmetros do modelo, eles não fornecem o suficiente para determiná-los de forma exclusiva. Para selecionar uma solução única para o problema inverso devemos adicionar ao problema algumas informações não contidas na equação $\mathbf{G}\mathbf{m} = \mathbf{d}$. Esta informação extra é chamada de informação a priori. As informações a priori podem assumir muitas formas, mas em cada caso, quantifica as expectativas (que os parâmetros do modelo possuem um determinado sinal ou se encontram em um determinado intervalo) sobre o caráter da solução que não são baseadas nos dados reais.

A origem e a qualidade das informações a priori são determinantes no resultado final. Um tipo de hipótese a priori que podemos considerar inicialmente é a expectativa de que a solução para o problema inverso é simples, isto pode ser quantificado pelo comprimento euclidiano da solução, $L = \mathbf{m}^T\mathbf{m}$. Portanto, uma solução é definida como simples se L for pequena, ou seja, sua energia é pequena. É certo que esta medida talvez não seja uma medida de simplicidade particularmente realista, mas pode ser útil, ocasionalmente. Uma instância em que o comprimento da solução pode ser realista advém quando os parâmetros do modelo descrevem a velocidade de vários pontos em um fluido em movimento. O comprimento L é então uma medida da energia cinética do fluido. Em certos casos, pode ser apropriado encontrar esse campo de velocidade no fluido que tenha a menor energia cinética possível dessas soluções que satisfaçam os dados (Menke, 2012).

Então, precisamos encontrar o \mathbf{m}_{est} que minimize L e o erro \mathbf{e} seja zero. A nova função objetivo usando o método de multiplicadores de Lagrange, seria:

$$S(\mathbf{m}) = L + \boldsymbol{\lambda}^T \mathbf{r} = \mathbf{m}^T \mathbf{m} + \boldsymbol{\lambda}^T (\mathbf{d} - \mathbf{G}\mathbf{m}) \quad (1.20)$$

derivando com respeito a \mathbf{m} e $\boldsymbol{\lambda}$, temos

$$\frac{\partial S(\mathbf{m})}{\partial \mathbf{m}} = 2\mathbf{m} - \mathbf{G}^T \boldsymbol{\lambda} \quad (1.21)$$

$$\frac{\partial S(\mathbf{m})}{\partial \lambda} = \mathbf{d} - \mathbf{Gm} \quad (1.22)$$

definindo as derivadas como zero:

$$\mathbf{m} = \frac{1}{2} \mathbf{G}^T \lambda \quad (1.23)$$

$$\mathbf{d} = \mathbf{Gm} \quad (1.24)$$

combinando as equações 1.23 e 1.24, temos

$$\mathbf{d} = \frac{1}{2} \mathbf{G} \mathbf{G}^T \lambda \quad (1.25)$$

vamos assumir que $\mathbf{G} \mathbf{G}^T$ é inversível. Então

$$\lambda = 2(\mathbf{G} \mathbf{G}^T)^{-1} \mathbf{d} \quad (1.26)$$

substituindo esta última equação em 1.23, dá a solução dos “mínimos quadrados”:

$$\mathbf{m} = \mathbf{G}^T (\mathbf{G} \mathbf{G}^T)^{-1} \mathbf{d} = \mathbf{G}^{-g} \mathbf{d} \quad (1.27)$$

1.2.1.3 Mínimos quadrados amortecidos MQA

No caso sobre-determinado, é minimizado $\mathbf{r}^T \mathbf{r}$. No caso sub-determinado, minimizamos $\mathbf{m}^T \mathbf{m}$. Outra abordagem é minimizar a soma ponderada: $c_1 \mathbf{r}^T \mathbf{r} + c_2 \mathbf{m}^T \mathbf{m}$. A solução \mathbf{m} depende da razão c_2/c_1 , não em c_1 e c_2 individualmente. Esta abordagem é usada para resolver problemas misto-determinados. Uma abordagem comum para obter uma solução inexata para um sistema linear é minimizar a função objetivo:

$$S(\mathbf{m}) = \mathbf{r}^T \mathbf{r} + \varepsilon \mathbf{m}^T \mathbf{m} \quad (1.28)$$

onde $\varepsilon > 0$, derivando $S(\mathbf{m})$ com respeito a \mathbf{m} , obtemos

$$\begin{aligned} \frac{\partial S(\mathbf{m})}{\partial \mathbf{m}} &= 2\mathbf{G}^T \mathbf{Gm} - 2\mathbf{G}^T \mathbf{d} + 2\varepsilon \mathbf{m} \\ &= 2\mathbf{G}^T (\mathbf{Gm} - \mathbf{d}) + 2\varepsilon \mathbf{m} \end{aligned} \quad (1.29)$$

igualando a derivada a zero,

$$\begin{aligned} \frac{\partial S(\mathbf{m})}{\partial \mathbf{m}} = 0 &\Rightarrow \mathbf{G}^T \mathbf{Gm} + \varepsilon \mathbf{m} = \mathbf{G}^T \mathbf{d} \\ &(\mathbf{G}^T \mathbf{G} + \varepsilon \mathbf{I}) \mathbf{m} = \mathbf{G}^T \mathbf{d} \end{aligned} \quad (1.30)$$

então, a solução é dada por

$$\mathbf{m} = (\mathbf{G}^T \mathbf{G} + \varepsilon \mathbf{I})^{-1} \mathbf{G}^T \mathbf{d} = \mathbf{G}^{-g} \mathbf{d}. \quad (1.31)$$

Isso é referido como “carregamento diagonal” porque uma constante, ε , é adicionada aos elementos diagonais de $\mathbf{G}^T \mathbf{G}$. A abordagem também evita o problema de posto incompleto

porque $\mathbf{G}^T\mathbf{G} + \varepsilon\mathbf{I}$ é inversível mesmo que $\mathbf{G}^T\mathbf{G}$ não seja. Além disso, a solução 1.31 pode ser usada em ambos casos: sub-determinado e sobre-determinado. A escolha do fator ε determina a importância relativa dada ao erro de predição e ao comprimento da solução. Se ε for grande, a solução não minimizará o erro de predição e não será uma estimativa muito boa dos parâmetros do modelo verdadeiro. Se ε for muito perto de zero, o erro de predição será minimizado, mas nenhuma informação a priori será fornecida para obter a solução dos parâmetros do modelo subdeterminados. No entanto, pode ser possível encontrar algum valor de ε que minimizará aproximadamente o erro de predição enquanto minimiza aproximadamente o comprimento da parte sub-determinada da solução (Menke, 2012).

1.2.1.4 Mínimos quadrados ponderados

Há muitos casos em que $L = \mathbf{m}^T\mathbf{m}$ não é uma medida muito boa de simplicidade de solução. Uma forma de obter uma melhor solução é incluindo informação a priori, por exemplo, do modelo $\langle \mathbf{m} \rangle$, então, uma generalização do L seria:

$$L = (\mathbf{m} - \langle \mathbf{m} \rangle)^T (\mathbf{m} - \langle \mathbf{m} \rangle) \quad (1.32)$$

incluindo uma matriz de ponderação \mathbf{W}_m , podemos quantificar uma grande variedade de medidas de simplicidade

$$L = (\mathbf{m} - \langle \mathbf{m} \rangle)^T \mathbf{W}_m (\mathbf{m} - \langle \mathbf{m} \rangle) \quad (1.33)$$

A matriz \mathbf{W}_m pode ser tal que é capaz de impor restrições ao respeito da suavidade da solução.

Medidas ponderadas do erro de predição também podem ser úteis. Muitas vezes, algumas observações são feitas com mais precisão do que outras. Neste caso, seria conveniente que o erro de previsão r_i das observações mais precisas tivesse maior peso na quantificação do erro geral \mathbf{E} do que as observações imprecisas. Para realizar esta ponderação, definimos um erro de predição generalizado

$$\mathbf{E} = \mathbf{r}^T \mathbf{W}_e \mathbf{r}, \quad (1.34)$$

onde a matriz \mathbf{W}_e define a contribuição relativa de cada erro individual para o erro de predição total. Normalmente, \mathbf{W}_e é uma matriz diagonal.

Em resumo, temos as seguintes situações:

- Mínimos quadrados ponderados: se o problema é completamente sobre-determinado, então pode-se estimar os parâmetros do modelo minimizando o erro de predição generalizado $\mathbf{E} = \mathbf{r}^T \mathbf{W}_e \mathbf{r}$. Usando o resultado da equação 1.19

$$\mathbf{m} = \mathbf{m}_{est} = (\mathbf{G}^T \mathbf{W}_e \mathbf{G})^{-1} \mathbf{G}^T \mathbf{W}_e \mathbf{d} = \mathbf{G}^{-g} \mathbf{d} \quad (1.35)$$

- Energia mínima ponderada: se o problema é completamente sub-determinado, em algumas situações, é desejável minimizar a energia ponderada, ou seja, minimizar $L = \mathbf{m}^T \mathbf{W}_m \mathbf{m}$, onde W_m é uma matriz diagonal, tendo em conta o procedimento para obter o resultado 1.27, o modelo estimado seria:

$$\mathbf{m} = \mathbf{W}_m^{-1} \mathbf{G}^T (\mathbf{G} \mathbf{W}_m^{-1} \mathbf{G}^T)^{-1} \mathbf{d} = \mathbf{G}^{-g} \mathbf{d} \quad (1.36)$$

- Comprimento mínimo ponderado: Se a equação $\mathbf{G} \mathbf{m} = \mathbf{d}$ estiver completamente sub-determinada, então pode-se estimar os parâmetros do modelo escolhendo a solução mais simples, onde a simplicidade é definida pelo comprimento generalizado $L = (\mathbf{m} - \langle \mathbf{m} \rangle)^T \mathbf{W}_m (\mathbf{m} - \langle \mathbf{m} \rangle)$. Este procedimento leva à solução (Menke, 2012):

$$\mathbf{m} = \langle \mathbf{m} \rangle + \mathbf{W}_m^{-1} \mathbf{G}^T (\mathbf{G} \mathbf{W}_m^{-1} \mathbf{G}^T)^{-1} (\mathbf{d} - \mathbf{G} \langle \mathbf{m} \rangle) \quad (1.37)$$

- Mínimos quadrados ponderados amortecidos: Se a equação $\mathbf{G} \mathbf{m} = \mathbf{d}$ for ligeiramente subdeterminada ou misto-determinada, pode ser resolvida minimizando uma combinação de erro de predição e comprimento da solução, $S(\mathbf{m}) = E + \lambda L$. O parâmetro λ é escolhido por tentativa e o erro para produzir uma solução que tenha um erro de predição razoavelmente pequeno. A equação para a solução, obtida ao minimizar $S(\mathbf{m})$ em relação a \mathbf{m} , é então

$$\mathbf{m} = (\mathbf{G}^T \mathbf{W}_e \mathbf{G} + \lambda \mathbf{W}_m)^{-1} (\mathbf{G}^T \mathbf{W}_e \mathbf{d} + \lambda \mathbf{W}_m \langle \mathbf{m} \rangle) \quad (1.38)$$

1.2.2 Métodos de solução de sistemas lineares

A solução de Sistemas lineares de grande porte estão presentes em várias áreas do conhecimento humano. Como exemplos em geofísica temos o processamento de dados sísmicos, modelagem de fluxo de águas subterrâneas, estimação de velocidades do *background* e inversão de dados geofísicos entre outros. Um problema frequentemente encontrado na inversão sísmica e em outras aplicações de grande porte é que o volume de dados é tão grande que as matrizes necessárias para a solução de mínimos quadrados (LQ) não cabem na memória de um único computador. Grande parte do esforço computacional visa avaliar o operador inverso generalizado \mathbf{G}^{-g} .

Os métodos de resolução de sistemas lineares, que geralmente podem ser grandes, atingindo milhares de incógnitas, esparsos, com diferentes tipos de simetria dependendo do modelo matemático que os origina, podem ser agrupados em duas classes (Gardan, 1985; Björck, 1996) o de métodos diretos e iterativos. Os métodos diretos caracterizam-se por encontrar a solução exata do sistema linear, exceto erros de arredondamento, através de um

número pré-definido de etapas. Os métodos iterativos buscam a solução através de aproximações sucessivas dos valores desconhecidos do sistema até que um limite de erro aceitável ou um número máximo de iterações seja atingido.

Os problemas de mínimos quadrados aparecem naturalmente quando se deseja estimar valores de parâmetros de um modelo matemático a partir de dados medidos, que estão sujeitos a erros (Björck, 1996). Houve duas contribuições fundamentais para a solução numérica de problemas de mínimos quadrados lineares no século passado: o primeiro foi o desenvolvimento da decomposição QR que faz uma decomposição de qualquer matriz quadrada real \mathbf{G} em um produto $\mathbf{G} = \mathbf{QR}$ de uma matriz ortogonal \mathbf{Q} e uma matriz triangular superior \mathbf{R} , este método tem uma complexidade de $\mathbf{O}(n^3)$. A depender das características do problema existem vários métodos baseados em decomposição QR (DQR) tais como: DQR por transformação *Householder*, Ortogonalização por *Gram-Schmidt* e seus variantes (tais como Modificado de *Gram-Schmidt*, *Gram-Schmidt* com reortogonalização), além de várias implementações híbridas (Björck, 1996).

O segundo foi a decomposição em valores singulares (SVD) onde qualquer matriz $\mathbf{G} \in \mathbf{C}^{m \times n}$ pode ser fatorada como $\mathbf{G} = \mathbf{UDV}^T$, onde as matrizes $\mathbf{U} \in \mathbf{C}^{m \times m}$ e $\mathbf{V} \in \mathbf{C}^{n \times n}$ são unitárias e $\mathbf{D} \in \mathbf{R}^{m \times n}$ é uma matriz com os valores singulares na diagonal principal e zero nas demais entradas, o número de operações de ponto flutuante depende da implementação mas a complexidade esta na ordem de $\mathbf{O}(mn^2)$.

No caso de uma matriz hermitiana e definida positiva, a decomposição de Cholesky ou fatoração de Cholesky é uma boa opção, pois é aproximadamente duas vezes mais eficiente que a decomposição LU para resolver sistemas de equações lineares, a complexidade computacional das implementações do algoritmo é da ordem $\mathbf{O}(n^3)$ em geral (Björck, 1996).

O método de Gauss é um conhecido algoritmo direto de resolução de sistemas de equações lineares, cujas matrizes de coeficientes são densas. Se um sistema de equações lineares é não singular, o método Gauss garante a resolução do problema com o erro determinado pela precisão do cálculo. O conceito principal do método é uma modificação da matriz \mathbf{G} por meio de transformações equivalentes (que não alteram a solução do sistema) para uma forma triangular. Depois disso, os valores das variáveis desconhecidas desejadas podem ser obtidos diretamente de forma explícita. Outros métodos usados geralmente são baseados na eliminação *Gaussiana*, tais como: o método de Peters-Wilkinson para sistemas não simétricos, a solução pseudo-inversa da decomposição LU, o método do sistema aumentado (para evitar o cálculo explícito de $\mathbf{G}^T\mathbf{G}$) entre outros.

Em princípio, qualquer método iterativo para sistemas lineares definidos simétricos positivos pode ser aplicado ao sistema de equações normais (EN) $\mathbf{G}^T\mathbf{G}\mathbf{m} = \mathbf{G}^T\mathbf{d}$. A formação

explícita da matriz $\mathbf{G}^T\mathbf{G}$ pode ser evitada usando a forma fatorada das equações normais $\mathbf{G}^T(\mathbf{G}\mathbf{m} - \mathbf{d}) = \mathbf{0}$. Trabalhar apenas com \mathbf{G} e \mathbf{G}^T separadamente tem duas vantagens importantes. Primeiro, assim como nos métodos diretos, uma pequena perturbação em $\mathbf{G}^T\mathbf{G}$, por exemplo, por arredondamento, pode mudar a solução muito mais do que perturbações de tamanho semelhante na própria \mathbf{G} . Em segundo lugar, evitamos o cálculo explícito de $\mathbf{G}^T\mathbf{G}$ (Björck, 1996).

A principal desvantagem dos métodos iterativos é a sua fraca robustez e muitas vezes uma gama limitada de aplicabilidade. Freqüentemente, um método iterativo específico pode ser considerado muito eficiente para uma classe específica de problemas, mas se usado para outros casos, pode ser excessivamente lento ou não convergir (Björck, 1996). Métodos iterativos pré-condicionados, baseados na fatoração aproximada, podem ser considerados como uma combinação entre solucionadores diretos e iterativos.

O método do gradiente conjugado (GC) é um dos algoritmos iterativos mais conhecidos. Este método pode ser usado para resolver um sistema linear simétrico positivo definido de equações de alta dimensionalidade. Como na aritmética exata ele converge em no máximo n passos (para um sistema $n \times n$), foi inicialmente considerado um método direto. No entanto, a terminação finita não se aplica ao arredondamento, e o método começou a ser amplamente utilizado em meados dos anos setenta, quando se percebeu que deveria ser considerado um método iterativo. (Björck, 1996).

Outra classe de algoritmos são os recursivos, onde destaca o algoritmo proposto por Levinson, 1946, e melhorado por Durbin, 1960, que resolve recursivamente o sistema linear $\mathbf{A}\mathbf{m} = \mathbf{b}$, sendo \mathbf{A} uma matriz $n \times n$ Toeplitz simétrica.

O algoritmo foi melhorado por Trench, 1964, para resolver o sistema linear em $4n^2$ operações e depois otimizado por Zohar, 1969, para $3n^2$ multiplicações. A recursão de Levinson (LR) é usada extensivamente na análise de sinais e no processamento de dados em Geofísica. A principal vantagem é sua eficiência computacional. A maioria das aplicações estão relacionadas com processamento de dados sísmicos e de-convolução preditiva (Robinson, 1957; Wiggins e Robinson, 1965; Treitel e Robinson, 1966; Peacock e Treitel, 1969; Claerbout, 1976; Robinson, 1983; Berkhout, 1977; Porsani e Ursin, 2007).

Uma extensão do algoritmo de Levinson foi proposta por Porsani e Ulrych, 1991. Eles desenvolveram uma modificação da LR no qual o sistema de equações não possuem esta simetria específica (não-Toeplitz), em particular para as EN do problema de MQ. A ideia fundamental desta extensão do algoritmo tipo-Levinson é o cálculo de uma solução de ordem j baseada em duas soluções independentes referidas a soluções direta e reversa (*F&B forward and backward*) de ordem $j - 1$ e qualquer combinação linear delas, associadas a dois sub-

sistemas de ordem menor. Esse novo algoritmo possibilita reutilizar as soluções parciais de um problema caso outro tenha colunas em comum, e evita o cálculo da inversa de matrizes. No caso em que a matriz dos coeficientes não possui alguma estrutura especial, o número de multiplicações e divisões necessária na inversão é $n^3 - 2n^2 + 4n$ ou seja $\mathbf{O}(n^3)$. Essa metodologia se reduz à LR clássica sempre que a matriz dos coeficientes é Toeplitz simétrica. Em resumo, usando o princípio de Levinson para todos os subsistemas menores, Porsani e Ulych, 1991, desenvolveram uma abordagem tipo-Levinson para a solução de sistemas de equações que não necessariamente possuem a estrutura Toeplitz. De acordo com o LR, a solução é obtida sem conhecer a inversa da matriz dos coeficientes.

Porsani et al., 2010, propuseram um método para solução MQ de sistemas lineares blocoparticionados, sobre-determinado ou subdeterminados, que apresenta grande potencial para a utilização de multiprocessadores permitindo que a solução do sistema possa ser obtida de forma rápida. O método é particularmente útil para o desenvolvimento de algoritmos para resolução de grandes sistemas, associados a problemas de geofísica direta e inversa. Esse algoritmo tem como objetivo resolver problemas MQ de grande porte, nos quais a memória de um único computador não suporta as matrizes envolvidas. Nesse caso, o princípio básico utilizado na recursão de Levinson também foi usada para possibilitar a construção da soluções de maior ordem a partir das menores, permitindo assim o uso de múltiplos processadores. Este método será abordado no capítulo 2 deste documento.

1.3 Ambiente de Desenvolvimento

Nesta seção é apresentado o ambiente computacional de desenvolvimento do trabalho. São abordados aspectos básicos relativos a arquitetura computacional e aos computadores usados no desenvolvimento desta pesquisa, além de uma breve descrição das ferramentas (API e bibliotecas) de programação usadas.

A nova implementação proposta nesse trabalho foi desenvolvida para uma exploração eficiente do paralelismo em sistemas SMP (*Shared Memory multiProcessor*), *clusters* de PCs multiprocessados e especialmente para uso de GPU. Neste seção são abordados, de forma resumida, conceitos relativos à arquitetura utilizada, bem como as ferramentas usadas para a exploração do paralelismo.

1.3.1 Computadores paralelos

Um computador paralelo é formado por um conjunto de processadores que trabalham em conjunto na solução de um determinado problema (Foster, 1995).

O paralelismo pode ser introduzido em vários níveis e classificados em função de seu nível de acoplamento, desde os mais fortemente acoplados, aqueles que usam o paralelismo no chip, até os mais fracamente acoplados (computação em grade). Na Figura 1.1 é resumido e ilustrado esse espectro (Tanenbaum e Austin, 2012).

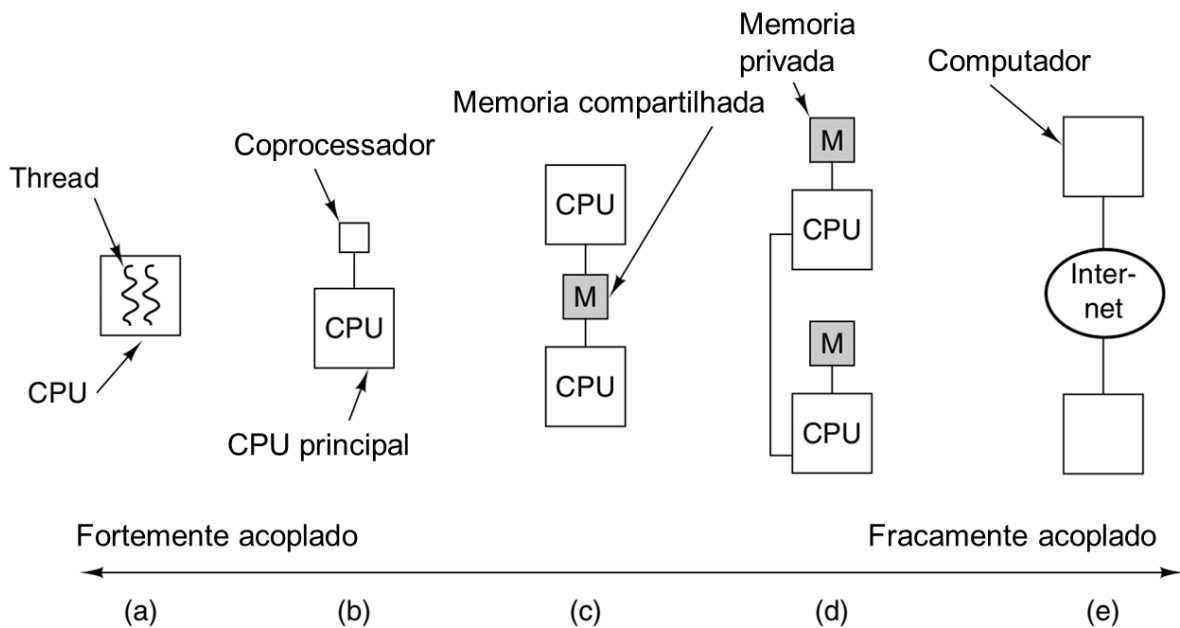


Figura 1.1: (a) Paralelismo no chip. (b) Um coprocessador. (c) Um multiprocessador. (d) Um multicomputador. (e) Uma grade

Quando duas CPUs ou elementos de processamento estão próximos, têm alta largura de banda (*bandwidth*), baixo atraso entre eles, são computacionalmente íntimos e estão fortemente acoplados. Por outro lado, quando estão distantes, têm baixa largura de banda e alto retardo e são computacionalmente remotos, são considerados fracamente acoplados (Tanenbaum e Austin, 2012).

O paralelismo no nível mais baixo (e mais fortemente acoplado), ele pode ser adicionado ao chip da CPU (Figura 1.1a), usando, por exemplo, recursos especiais que permitam que lide com vários *threads*¹ de controle ao mesmo tempo. Finalmente, várias CPUs podem ser colocadas juntas no mesmo chip. No próximo nível, placas de CPU extras com capacidade

¹Uma *thread* pode ser definido como um fluxo de instrução independente (Chapman et al., 2008)

de processamento adicional podem ser adicionadas a um sistema -Coprocessadores- (Figura 1.1b). Normalmente, essas CPUs *plug-in* têm funções especializadas, como processamento de pacotes de rede, processamento de multimídia (como as GPU) ou criptografia. O próximo nível é logrado ao replicar CPUs inteiras e fazer com que todas funcionem juntas de forma eficiente. Essa ideia leva a grandes multiprocessadores e multicomputadores (computadores em *cluster*, Figura 1.1c e d). Finalmente, agora é possível unir organizações inteiras pela Internet para formar grades de computação fracamente acopladas (Figura 1.1e) (Tanenbaum e Austin, 2012).

De acordo com a organização da memória os computadores paralelos podem ser divididos em dois grupos: máquinas com memória compartilhada (Figura 1.1c) e máquinas com memória distribuída (Figura 1.1d), nas quais cada processador possui uma memória própria e a comunicação entre os processadores é feita através de uma rede de interconexão. A organização da memória e a forma como os processadores se comunicam entre si é de fundamental importância para o projeto e desenvolvimento de aplicações paralelas, devido que cada arquitetura é mais adequada para certos tipos de problemas e não é adequada para outros tipos de problemas (Kaminsky, 2009).

Nas seções 1.3.1.1 e 1.3.1.2 são introduzidos conceitos sobre máquinas com memória compartilhada e máquinas com memória distribuída, respectivamente. Na seção 1.3.1.4 são introduzidos conceitos sobre *clusters* de PCs híbridos, um dos ambientes de desenvolvimento usado neste trabalho, uma vez que possuem memória compartilhada entre os processadores de cada nó e memória distribuída entre os nós do *clusters*. Generalidades sobre as GPU são apresentadas na seção 1.3.1.3 e na seção 1.3.2 são apresentados os computadores usados neste trabalho.

É perfeitamente possível escrever programas paralelos usando uma linguagem de programação padrão e funções genéricas do *kernel* do sistema operacional. No entanto, os programas paralelos geralmente não são escritos dessa maneira, por dois motivos. Primeiro, algumas linguagens de programação populares que se beneficiam da programação paralela (como Fortran e C para computação científica) não oferecem suporte à programação *multi-thread* e à programação em rede. Em segundo lugar, o uso de bibliotecas de baixo nível para *thread* e *socket* aumenta o esforço necessário para escrever um programa paralelo. É preciso muito esforço para escrever o código que configura e coordena os vários *threads* de um programa paralelo SMP ou para escrever um código que configura conexões de rede e envia e recebe mensagens para um programa paralelo de *cluster*. Os usuários de programas paralelos estão interessados em resolver problemas de seu interesse, então, para escrever um programa paralelo, geralmente se prefere usar uma biblioteca de programação paralela. A biblioteca

encapsula o código baixo nível e de rede que é o mesmo em qualquer programa paralelo, apresentando abstrações de programação paralela de alto nível e fáceis de usar (por exemplo OpemMP para sistemas de memória compartilhada e MPI para sistemas de memória distribuída). Então, pode-se dedicar a maior parte do esforço de programação paralela para resolver seu problema usando essas abstrações (Kaminsky, 2009). Além disso, existem bibliotecas de alto desempenho com fines específicos, como por exemplo SCALAPACK com rotinas de álgebra linear desenhadas para computadores com memória distribuída que tem encapsuladas todas as funções MPI (Dongarra, 1994). Na seção 1.3.3 é apresentado considerações básicas das bibliotecas usadas neste trabalho.

1.3.1.1 Computador paralelo com multiprocessador de memória compartilhada (SMP)

Um computador paralelo com multiprocessador de memória compartilhada (*Shared Memory multiProcessor* - SMP) é um sistema multiprocessador simétrico. Para obter ganhos de desempenho além do possível em uma única CPU, os arquitetos de computador replicaram a CPU, resultando em um multiprocessador simétrico. É chamado de “simétrico” porque todos os processadores são idênticos. Cada processador é uma CPU completa com sua própria unidade de instrução, unidades funcionais, registros e *cache*. Todos os processadores compartilham a mesma memória principal e periféricos. O computador atinge maior desempenho executando vários *threads* de execução simultaneamente, um em cada processador (Kaminsky, 2009).

O *cache* é um buffer de memória pequena e de velocidade muito alta entre a memória principal e o processador. Os dados são copiados da memória principal para o cache e retornados em blocos de dados contíguos, cujo tamanho depende da máquina. Um bloco de dados é armazenado em uma linha de cache e pode ser despejado do cache se outro bloco de dados precisar ser armazenado na mesma linha. A estratégia para substituir dados no cache é dependente do sistema. Como o cache é geralmente construído a partir de componentes de memória muito caros, ele é substancialmente menor que a memória principal. No entanto, os tamanhos de cache estão crescendo e, em algumas plataformas, pequenos conjuntos de dados podem caber inteiramente no cache. Sem o cache, um programa gastaria a maior parte do tempo de execução aguardando a chegada dos dados, já que as velocidades da CPU são significativamente mais rápidas do que os tempos de acesso à memória. Os computadores podem ter vários níveis de cache, sendo que os mais próximos da CPU são menores, mais rápidos e mais caros do que os mais próximos da memória principal. Os dados são copiados entre os níveis (Chapman et al., 2008).

A execução de um programa paralelo em um computador paralelo SMP consiste em um processo com vários *threads*, um *thread* em execução em cada processador. O programa e os dados do processo residem na memória principal compartilhada. Como todos os *threads* estão no mesmo processo, todos os *threads* fazem parte do mesmo espaço de endereço (*address space*), portanto, todos os *threads* acessam o mesmo programa e dados. Cada *thread* executa sua parte do cálculo e armazena seus resultados nas estruturas de dados compartilhadas. Se os *threads* precisam se comunicar ou coordenar uns com os outros, eles o fazem lendo e gravando valores nas estruturas de dados compartilhadas (Kaminsky, 2009).

O uso de memória compartilhada permite uma transição mais natural de ambientes monoprocessados, menor custo de comunicação e a eliminação da necessidade de tarefas como particionamento de dados e ferramentas predefinidas para o balanceamento de carga (Chapman et al., 2008).

Várias APIs foram desenvolvidas para oferecer suporte para a criação e manipulação de *threads*. Entre as quais, se destacam Pthreads e OpenMP; Pthreads (baixo nível) requer que o programador especifique explicitamente o comportamento de cada *thread*. O OpenMP (alto nível), por outro lado, às vezes permite ao programador simplesmente afirmar que um bloco de código deve ser executado em paralelo, e a determinação precisa das tarefas e qual *thread* deve executá-las é deixada para o compilador (Chapman et al., 2008).

1.3.1.2 Computador paralelo de memória Distribuída, *Clusters*

Em Computadores paralelos com memória distribuída (multicomputadores), cada processador possui uma memória própria, que não pode ser acessada de forma direta por outro processador (Tanenbaum e Austin, 2012). Cada processador está conectado aos outros por meio de uma rede dedicada de alta velocidade. Todos os processos executam o mesmo programa, uma cópia do qual reside na memória de cada processo. Os dados do programa são divididos em partes; uma parte diferente reside no espaço de endereço de cada processo. Cada processo executa sua parte do cálculo e armazena seus resultados nas estruturas de dados de sua própria memória local. Se um processo precisa de um dado que reside no espaço de endereço de outro processo, o processo que possui os dados envia uma mensagem contendo os dados por meio da rede para o processo que precisa dos dados. Os processos também podem trocar mensagens para se coordenar entre si, sem transferir dados (Kaminsky, 2009).

A principal limitação no uso do paradigma de troca de mensagens é o alto *overhead*² na comunicação e sincronização dos processos. O alto *overhead* dificulta, ao programador, o

²O *overhead* descreve o período de tempo em que um processador está ocupado enviando ou recebendo uma mensagem; durante esse tempo, nenhum outro trabalho pode ser executado por esse processador.

desenvolvimento de aplicações de alto desempenho, podendo, até mesmo, tornar inviável o uso da troca de mensagens em aplicações com grande dependência entre as operações. Para um máximo desempenho, não é suficiente equipar um *cluster* com processadores rápidos. Também é importante que a rede do *cluster* tenha três características: baixa latência,³ alta largura de banda⁴ e alta largura de banda de bisseção⁵ (Kaminsky, 2009).

Embora as aplicações que utilizam troca de mensagens sejam próprias para ambientes de memória distribuída, isso não impede que sejam executadas em computadores paralelos com memória compartilhada. Porém, não aproveitam a principal vantagem desse tipo de sistema, que é o uso de uma memória comum para a comunicação entre os processadores. Apesar do alto custo de comunicação, o uso do modelo de troca de mensagens tem crescido muito nos últimos anos. Um dos maiores motivos pela difusão desse modelo é a disponibilidade de um grande número de bibliotecas que oferecem serviços de troca de mensagens. Das quais destaca-se o MPI (*Message Passing Interface*)

Para alguns autores, um outro problema do paradigma de troca de mensagens é que esse é mais complexo e difícil de programar, quando comparado ao uso de memória compartilhada. Uma alternativa para esse problema é o uso de ferramentas que permitem simular um ambiente de memória compartilhada em ambientes com memória distribuída. Essas ferramentas são chamadas de DSM (*Distributed Shared Memory*) e adicionam custos que podem diminuir o desempenho da aplicação (Tanenbaum e Austin, 2012).

1.3.1.3 Coprocessador: GPU (*Graphics Processing Unit*)

Um computador pode ser acelerado com a adição de um segundo processador especializado. Esses coprocessadores vêm em muitas variedades, de pequeno a grande porte e atuam em diferentes áreas, das quais se destacam: processamento de rede (para lidar com o tráfego de alta velocidade), multimídia (para lidar com processamento gráfico de alta resolução, como renderização 3D fazendo uso de GPUs) e criptografia (para lidar com a computação necessária para criptografar dados com segurança, para transmissão ou armazenamento e, em seguida, descriptografá-los mais tarde) (Tanenbaum e Austin, 2012).

Atuando como coprocessador da CPU principal, a GPU é um chip especializado que lida com cálculos de renderização de gráficos em velocidades muito altas. A CPU envia co-

³Latência refere-se à quantidade de tempo necessária para iniciar uma mensagem, independentemente do tamanho; depende dos protocolos de hardware e software usados na rede, baixa Latência e o ideal.

⁴A largura de banda, medida em bit por segundo, é a taxa na qual os dados são transmitidos assim que uma mensagem é iniciada, o melhor é uma grande largura de banda

⁵A largura de banda de bisseção, também medida em bit por segundo, refere-se à taxa total na qual os dados podem ser transferidos se metade dos nós no *cluster* estiver enviando mensagens para a outra metade.

mandos de alto nível à GPU para desenhar linhas e preencher formas com sombreamento e iluminação realistas; a GPU então calcula a cor de cada pixel e direciona a exibição. Como os pixels podem ser calculados de forma independente, a GPU normalmente tem vários núcleos de processamento e calcula vários pixels em paralelo. Os programadores perceberam que as GPUs podem ser usadas para fazer cálculos paralelos, além da renderização de pixels e até criaram um acrônimo para isso: GPGPU (*General-Purpose computing on Graphics Processing Units*), ou computação de propósito geral em unidades de processamento gráfico. Em resposta, os fornecedores de GPU reempacotaram suas placas gráficas como coprocessadores massivamente paralelos de uso geral, completos com memória compartilhada *on-board* e APIs para programação paralela na GPU. (Kaminsky, 2009).

De modo geral, quando um usuário decide processar um programa em um acelerador (por exemplo em uma GPU), o programa é processado em três etapas (Kale, 2019):

- Primeiro, os dados necessários são transferidos para a memória do dispositivo (a memória do acelerador).
- Então, usando os dados na memória do dispositivo, o acelerador executa o programa.
- Após que os dados são processados, e o acelerador retorna o resultado para a CPU.

A GPU é famosa por sua capacidade de processamento paralelo. Se um algoritmo requer pouca ou nenhuma comunicação entre os *threads*, a GPU pode oferecer um aumento significativo de velocidade em comparação com a CPU. A GPU não é boa executando programas com muitas ramificações e comunicações, alguns algoritmos não otimizados podem até ser mais lentos em uma GPU do que em uma CPU. O desempenho da GPU varia para diferentes tamanhos de carga de trabalho. Se a carga de trabalho for muito pequena, a GPU não será capaz de atingir seu potencial de desempenho total (Kale, 2019).

Quando se quer programar uma GPU, podemos escolher três abordagens que diferem, pelo nível de controle que fornecem sobre o hardware e pelo número de linhas de código que você precisa escrever: abordagens de baixo nível, diretivas de compilador e bibliotecas. Abordagens de baixo nível, como CUDA, fornecem mais controle sobre o hardware. Para programar o mesmo algoritmo, as abordagens de nível superior nos permitem escrever menos código do que as abordagens de baixo nível. Um ponto importante a ser mencionado é que você não precisa se comprometer com uma abordagem em vez de outra. Você pode misturar abordagens diferentes, pois elas podem interoperar juntas.

1.3.1.3.1 Abordagens de baixo nível, CUDA e OpenCL. NVIDIA CUDA é um dos *frameworks* mais populares na categoria de baixo nível. CUDA é uma plataforma para-

lela criada e de propriedade da NVIDIA. Isso também significa que CUDA é uma estrutura proprietária voltada apenas para plataformas de computação NVIDIA. Agora, o OpenCL permite programar diferentes GPUs e aceleradores, como DSPs (*Digital Signal Processors*) e FPGAs (*Field-Programmable Gate Arrays*). O padrão, mantido por um consórcio da indústria chamado Khronos Group ⁶, permite que os programadores escrevam seus programas de aplicativos em linguagens C/C++ que são executados em uma ampla variedade de dispositivos produzidos por diferentes fornecedores, alcançando portabilidade do código do aplicativo. Ele oferece suporte a modelos de computação paralelos de dados e de tarefas paralelas. CUDA e OpenCL têm muitos conceitos comuns (Kale, 2019).

1.3.1.3.2 Abordagens por diretivas de compilador. Na abordagem da diretiva do compilador, o programador introduz anotações no código, chamadas diretivas do compilador ou *pragmas*, para informar ao compilador qual parte do código deve ser executada na GPU. A API OpenACC usa esta abordagem. OpenACC é semelhante ao OpenMP (para a aceleração em computadores com multiprocessadores e memória compartilhada), mas visa a aceleração dispositivos como GPUs.

1.3.1.3.3 Abordagens por bibliotecas. Também existem bibliotecas que podem ser usadas para programação em GPU. As duas bibliotecas mais populares são *Thrust* e *ArrayFire*. *Thrust* é uma biblioteca C++, e é basicamente uma biblioteca padronizada C++ Standard (STL) para GPU. Além disso, *ArrayFire* é uma biblioteca de funções de código aberto abrangente com interfaces para C, C++, Java, R e Fortran. Ele se integra a qualquer aplicativo CUDA e contém uma API baseada em *array* para fácil programação.

Tem outras para finalidades específicas, tais como: Bibliotecas Matemáticas (CUDA Math Library, cuBLAS, cuFFT, cuRAND, cuSOLVER, cuSPARSE, cuTENSOR, AmgX, MAGMA, IMSL Fortran Numerical Library, CHOLMOD); Bibliotecas de aprendizado profundo (cuDNN, NVIDIA TensorRT, NVIDIA Jarvis, NVIDIA DeepStream SDK, NVIDIA DeepStream SDK, NVIDIA DALI, OpenCV) entre muitas outras.

Neste trabalho, nos usamos a biblioteca MAGMA (*Matrix Algebra on GPU and Multicore Architectures*), da qual falaremos na seção 1.3.3.8.

⁶Maiores informações em: <https://www.khronos.org/opencl/>

1.3.1.4 Computadores Paralelos Híbridos

Um computador paralelo híbrido é um *cluster* em que cada processador é uma máquina SMP. Em outras palavras, é uma combinação de *cluster* e computadores paralelos SMP. Um computador paralelo híbrido tem memória compartilhada e memória distribuída. Um computador paralelo híbrido é programado usando uma combinação de técnicas de programação paralela para *cluster* e SMP. Como um computador paralelo em *cluster*, cada computador executa um processo separado com seu próprio espaço de endereço. Cada processo executa uma cópia do mesmo programa e cada processo possui uma parte dos dados. Por sua parte, como um computador paralelo SMP, cada processo, tem vários *threads*, um thread em execução em cada CPU. *Threads* no mesmo processo compartilham o mesmo espaço de endereço e podem acessar suas próprias estruturas de dados compartilhadas diretamente. *Threads* em processos diferentes devem enviar mensagens entre si para transferir dados (Kaminsky, 2009).

1.3.2 Computadores Utilizados

Nesta seção, daremos a conhecer as especificações técnicas mais relevantes dos computadores usados.

1.3.2.1 Marreca

A MARRECA é um computador do tipo SMP com um coprocessador GPU. Ela está equipada com um Processador Intel® Xeon® E5-2643 v3, e tem instalada 524 GB de memória RAM, na tabela 1.1 são apresentados os principais especificações técnicas do processador.

Item	Valor
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Thread(s) per core:	2
Core(s) per socket:	6
Socket(s):	2
NUMA node(s):	2
CPU family:	6
Model name:	Intel(R) Xeon(R) CPU E5-2643 v3 @ 3.40GHz
Stepping:	2
CPU MHz:	1200.000
BogoMIPS:	6783.80
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K

Tabela 1.1: Especificações técnicas principais do processador do computador MARRECA

A máquina está equipada com um coprocessador (Acelerador) GPU NVIDIA Tesla K40c, cujas características principais são observadas na Tabela 1.2.

Item	Valor
Peak double-precision floating point performance (board)	1.43 Tflops
flopsPeak single-precision floating point performance (board)	4.29 Tflops
GPU	1 x GK110B
CUDA cores	2,880
Memory size per board (GDDR5)	12 GB
Memory bandwidth for board (ECC off)	2,288 Gbytes/sec

Tabela 1.2: Especificações técnicas principais do coprocessador Tesla K40c do computador MARRECA

1.3.2.2 Cluster Águia

Cada máquina da ÁGUIA tem nós com processadores e memória RAM diferentes, mais nos testes só foram usados os nós com 24GB de RAM e com processadores com as seguintes características (Tabela 1.3):

Item	Valor
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Thread(s) per core:	2
Core(s) per socket:	4
CPU socket(s):	2
NUMA node(s):	2
model name :	Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
CPU family:	6
Model:	44
Stepping:	2
CPU MHz:	1600.000
BogoMIPS:	4799.88
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	12288K

Tabela 1.3: Especificações técnicas principais dos nós da Águia

1.3.2.3 Cluster OGUN

O HPC OGUN é um *cluster* híbrido e heterogêneo (que usam mais um tipo de processadores). Faremos referência só a os nós que usamos neste trabalho: para os testes de memória compartilhada e distribuída usamos a partição (fila) *standard*, que tem CPU com características descritas no paragrafo 1.3.2.3.1; para os testes usando GPU, o *cluster* tem uma partição (fila) especial chamada **gpu** que tem 2 CPU com características descritas no paragrafo 1.3.2.3.2.

1.3.2.3.1 Nós sem GPU, partição *standard* Esta partição tem 46 nós, onde cada nó está equipado com um Processador Intel(R) Xeon(R) Gold 6148 (40 cores, 80 *threads*) onde cada nodo tem instalada 196 GB de memória RAM, na tabela 1.4 são apresentados os principais especificações técnicas do processador.

1.3.2.3.2 Nós com GPU, partição *gpu* Esta partição tem 2 nós, onde cada nó esta equipado com um Processador Intel(R) Xeon(R) CPU E5-2698 v4 (40 cores, 80 *threads*) onde um dos nós tem instalada 260 GB e o outro 98GB de memória RAM, na tabela 1.4 são apresentados os principais especificações técnicas do processador.

Cada nodo esta equipado com um coprocessador (Acelerador) GPU NVIDIA P100 NVLINK, com 16GB de memória. As características principais são observadas na Tabela 1.5.

Item	Valor (P <i>standar</i>)	Valor (P <i>gpu</i>)
Architecture:	x86_64	x86_64
CPU op-mode(s):	32-bit, 64-bit	32-bit, 64-bit
Byte Order:	Little Endian	Little Endian
Thread(s) per core:	2	2
Core(s) per socket:	20	20
Socket(s):	2	2
NUMA node(s):	2	2
Vendor ID:	Intel(R)	Intel(R)
Model name:	Xeon(R) Gold 6148 CPU	Xeon(R) CPU E5-2698 v4
Stepping:	4	1
CPU MHz:	2401.0000	2473.539
CPU max MHz:	2401,0000	3600,0000
CPU min MHz:	1000,0000	1200,0000
BogoMIPS:	4800.00	4399.95
Virtualization:	VT-x	VT-x
L1d cache:	32K	32K
L1i cache:	32K	32K
L2 cache:	1024K	256K
L3 cache:	28160K	51200K

Tabela 1.4: Especificações técnicas principais dos nós com e sem GPU (Partições *standar* e *gpu*) do super computador OGUM

Item	Valor
Double-Precision Performance	5.3 teraFLOPS
Single-Precision Performance	10.6 teraFLOPS
NVIDIA CUDA® Cores	3584
NVIDIA NVLink Interconnect Bandwidth	160 GB/s
PCIe x16 Interconnect Bandwidth	32 GB/s
CoWoS HBM2 Stacked Memory Capacity	16 GB
CoWoS HBM2 Stacked Memory Bandwidth	732 GB/s

Tabela 1.5: Especificações técnicas principais do coprocessador NVIDIA P100 NVLINK dos nós GPU do Supercomputador OGUN (SENAI SIMATEC)

1.3.3 Bibliotecas de programação

Na seção 1.3.1 foram descritos sistemas de memória de compartilhada, de memória distribuída, híbridos e acelerados com um coprocessador GPU. Para cada um dos sistemas o paralelismo pode ser explorado de forma intra-nodal (SMP), inter-nodal (*Clusters*), usando a capacidade do coprocessador ou todo junto. Existem diferentes bibliotecas que permitem uma exploração eficiente do paralelismo. Nesta seção consiste em fazer uma breve apresentação das bibliotecas adotadas neste trabalho para a exploração do paralelismo.

1.3.3.1 BLAS

A biblioteca BLAS (*Basic Linear Algebra Subprograms*) são rotinas de baixo nível que fornecem blocos de construção padrão para realizar operações básicas de vetores e matrizes. O BLAS de nível 1 realiza operações escalares e vetoriais, já o BLAS de nível 2 realiza operações de matriz-vetor e por último o BLAS de nível 3 realiza operações de matriz-matriz. As rotinas têm ligações para C (“interface CBLAS”) e Fortran (“interface BLAS”). Como as rotinas BLAS são eficientes, portáteis e amplamente disponíveis, eles são comumente usados no desenvolvimento de software de álgebra linear de alta qualidade, LAPACK, por exemplo. A interface foi padronizada pelo *BLAS Technical (BLAST) Forum*, cujo último relatório BLAS pode ser encontrado no site netlib (BLAS, 2021).

A maioria das bibliotecas que oferecem rotinas de álgebra linear são congruentes com a interface BLAS, permitindo que os usuários da biblioteca desenvolvam programas que são indiferentes à biblioteca BLAS que está sendo usada. Exemplos de bibliotecas BLAS incluem: BLIS⁷, OpenBLAS, ATLAS⁸ e Intel Math Kernel Library (MKL).

1.3.3.2 OpenBLAS

OpenBLAS é uma biblioteca de código aberto (*Open source*) que explora o paralelismo usando processadores multi-core em sistemas SMP. É uma implementação da biblioteca BLAS otimizada manualmente para muitas das arquiteturas populares, incluindo Intel Sandy Bridge, Loongson, Haswell entre muitas outras. OpenBLAS é baseado no GotoBLAS2, que

⁷BLIS (*BLAS-like Library Instantiation Software*) é uma framework de software portátil para instanciar bibliotecas de álgebra linear densa semelhante a BLAS de alto desempenho. A estrutura foi projetada para isolar kernels essenciais de computação que, quando otimizados, permitem imediatamente implementações otimizadas da maioria de suas operações comumente usadas e intensivas em computação. BLIS é escrito em ISO C99. Ele está disponível em <https://github.com/flame/blis>.

⁸ATLAS (*Automatically Tuned Linear Algebra Software*) é uma biblioteca portátil que se otimiza automaticamente para uma arquitetura arbitrária. No momento, ele fornece interfaces C e Fortran77 para uma implementação de BLAS portavelmente eficiente, bem como algumas rotinas do LAPACK. Disponível em <http://math-atlas.sourceforge.net/>

foi criado por Kazushige Goto no *Texas Advanced Computing Center* (Goto e Van De Geijn, 2008) ⁹.

Os desenvolvedores do OpenBLAS reportaram que para algumas funções de álgebra linear (com matrizes densas) superam as implementações das bibliotecas Intel MKL e ATLAS, nos processadores Intel Sandy Bridge e AMD Piledriver (Wang et al., 2013).

1.3.3.3 LAPACK

LAPACK (Acrônimo para *Linear Algebra PACKage*) é uma biblioteca portátil (Originalmente escrita em FORTRAN 77 e atualizada a FORTRAN 90 em 2008) de álgebra linear para uso eficiente em uma variedade de computadores de alto desempenho, é usada para resolver equações lineares, problemas de autovalores/autovetores, problemas de mínimos quadrados lineares e problemas de valores singulares. A principal metodologia para fazer os algoritmos rodarem mais rápido é reestruturá-los para realizar operações de matriz de bloco (por exemplo, multiplicação de matriz-matriz) em seus loops internos. Essas operações de bloco podem ser otimizadas para explorar a hierarquia de memória de uma arquitetura específica. O LAPACK, foi projetado para explorar efetivamente a memória caches em arquiteturas modernas e, portanto, pode executar ordenes mais eficientemente, dada uma implementação BLAS otimizada (por exemplo OpenBlas) (Anderson et al., 1999; LAPACK, 2021). LAPACK também foi estendido para rodar em sistemas de memória distribuída em pacotes posteriores, como ScaLAPACK e PLAPACK.

1.3.3.4 OpenMP

OpenMP (*Open Multi-Processing*) é uma API ¹⁰ para desenvolvimento de aplicações paralelas que fazem uso de memória compartilhada (SMP) e fornece um conjunto de primitivas para criação, manipulação e gerenciamento de *threads*. OpenMP não é uma nova linguagem de programação. Em vez disso, é uma notação que pode ser adicionada (geralmente com modificações mínimas no código) para especificar paralelismo de alto nível em programas sequenciais Fortran, C ou C++ para descrever como o trabalho deve ser compartilhado entre *threads* que serão executados em diferentes processadores ou núcleos e para solicitar acessos a dados compartilhados conforme seja necessário (Chapman et al., 2008).

Em OpenMP o paralelismo pode ser obtido em dois diferentes níveis. Em um primeiro nível, através da paralelização de *loops* (*loop level*). Neste nível basta informar ao compilador

⁹A biblioteca OpenBLAS esta disponível em: <https://www.openblas.net/>.

¹⁰Gerenciado pelo consórcio de tecnologia sem fins lucrativos OpenMP *Architecture Review Board* ou OpenMP ARB. Maiores informações em: <https://www.openmp.org/>

o início e o fim do código (*loops*) a ser paralelizado e o número de *threads* que executarão o *loop*. Nesta forma de paralelização a geração do código paralelo é feita pelo próprio compilador. Cabe ressaltar que, um *loop* no qual os resultados de uma ou mais iterações dependem de outras iterações não pode, em geral, ser corretamente paralelizado pelo OpenMP. É responsabilidade do programador definir o escopo das variáveis (refere-se ao conjunto de *threads* que podem acessar a variável em um bloco paralelo: compartilhada ou privada) e se o *loop* precisa de alguma redução intermediária ou no final do *loop*, entre outras considerações. O segundo nível é uma paralelização de um bloco (*parallel region*) e é mais abrangente, e pode ser utilizado para atribuir peças distintas de trabalho aos *threads* individualmente. Essa abordagem é adequada quando cálculos independentes e a ordem em que são realizados é irrelevante. Nesse nível, tem-se uma programação semelhante à biblioteca Pthreads (embora facilitada pelas diretivas do OpenMP) cabendo ao programador o gerenciamento de seções críticas, cuidados com condições de corrida, etc. (Chapman et al., 2008).

1.3.3.5 MPI

MPI (*Message Passing Interface*) é um padrão que permite gerenciar processos e trocas de mensagens, ela é usada para programar sistemas de memória distribuída. É portátil e (tem múltiplas implementações, MPICH entre elas; um programa MPI correto deve ser capaz de ser executado em todas as implementações MPI sem alterações. Assim como OpenMP, MPI não é uma nova linguagem de programação. Ele define funções que podem ser chamadas a partir de programas escritos em C, C++ e Fortran. A criação de um programa paralelo baseado em MPI normalmente requer uma grande reorganização do código sequencial original. O esforço de desenvolvimento pode ser grande e complexo em comparação com OpenMP. MPI permite maior escalabilidade do que o OpenMP, no entanto MPI e OpenMP podem ser usados juntos em um programa, o qual seria útil se um programa for executado em MPPs (*Massively Parallel Computers*) que consistem em vários SMPs (possivelmente com vários núcleos cada). Os motivos para fazer isso incluem explorar uma granularidade de paralelismo mais fina do que a possível com MPI, reduzindo o uso de memória ou reduzindo a comunicação de rede (Chapman et al., 2008).

O principal objetivo da especificação MPI é oferecer a os usuários uma API eficiente, portátil e funcional. Especificamente, os usuários podem escrever programas portáteis que ainda tiram proveito do hardware e software especializados oferecidos por fornecedores individuais. A versão 3.1 inclui funções para passagem de mensagens ponto a ponto, comunicações coletivas, possibilita de criar de grupos e comunicadores, topologias de processos (define um mapeamento especial das classificações em um grupo de e para a topologia), gestão, criação

e gestão de processos, comunicações unilaterais, operações coletivas estendidas, interfaces externas, I/O, entre outras.

1.3.3.6 SCALAPACK

ScaLAPACK (*Scalable LAPACK*), é uma biblioteca de rotinas de álgebra linear de alto desempenho para computadores memória distribuída através de passagem de mensagens (por exemplo MPI). É uma continuação do projeto LAPACK (Choi et al., 1992; Choi et al., 1996a; Blackford et al., 1997). Ela oculta a maioria dos detalhes de comunicação com uma API baseada em objeto, onde cada informação do objeto da matriz é passada para as rotinas da biblioteca. Essa escolha de desenho aprimora a programabilidade da biblioteca, permitindo que os códigos sejam escritos de maneira semelhante a uma implementação LAPACK padrão. Os objetivos de ambos projetos são eficiência, escalabilidade, confiabilidade, portabilidade, flexibilidade (para que os usuários possam construir novas rotinas a partir de peças bem projetadas) e facilidade de uso (tornando a interface para LAPACK e ScaLAPACK o mais semelhante possível). Esses objetivos são atingidos já que conta com apenas duas bibliotecas externas (uma vez que PBLAS é considerado um módulo interno). O primeiro é o sequencial BLAS, fornecendo especificações para as operações mais comuns envolvendo vetores e matrizes (Seção 1.3.3.1). A segunda é a BLACS (*Basic Linear Algebra Communication Subroutines*), que, como o nome sugere, é uma especificação para tarefas comuns de comunicação matricial e vetorial (Anderson et al., 1991).

A biblioteca PBLAS (cuja interface é semelhante ao BLAS) é um módulo chave dentro do ScaLAPACK. Ele compreende a maioria das rotinas BLAS reescritas para uso em ambientes de memória distribuída. Esta biblioteca é escrita usando uma combinação da biblioteca BLAS sequencial e da biblioteca BLACS. Assim como a biblioteca BLAS contém os blocos de construção primária para rotinas LAPACK, a biblioteca PBLAS contém a base para as rotinas em ScaLAPACK. A biblioteca PBLAS tem um duplo propósito na biblioteca. Por um lado, como a biblioteca PBLAS espelha em função ao BLAS (sequencial), o nível superior de código nas rotinas ScaLAPACK principais parece basicamente o mesmo do que rotinas LAPACK correspondentes. Por outro lado, a biblioteca PBLAS adiciona uma camada de flexibilidade ao código em relação ao mapeamento de operações. Tradicionalmente, um processo é atribuído a cada núcleo durante a execução, mas com PBLAS (em paralelo), uma combinação de processos e *threads* pode ser usada. Essa capacidade de ajuste oferece mais opções ao mapear processos em núcleos antes do início da execução do programa (Mais detalhes sobre o PBLAS em e Choi et al., 1995 e Petit et al., 1999).

A estratégia de armazenamento de dados em uma computação de memória distribuída

tem um impacto significativo no custo de comunicação e no equilíbrio de carga durante a computação. Todas as rotinas ScaLAPACK assumem o chamado esquema de “distribuição cíclica em bloco” (Choi et al., 1996a; Choi et al., 1996b). Que envolve vários parâmetros definidos pelo usuário, a escolha desses parâmetros é vital para construir uma implementação eficiente. O esquema de distribuição cíclica em bloco envolve quatro parâmetros. Os dois primeiros, mb e nb , definem o tamanho do bloco, ou seja, as dimensões das submatrizes utilizadas como unidade fundamental para a comunicação entre processos. Apesar desta flexibilidade, por simplicidade geralmente se usa $mb = nb$ (o valor ideal vai depender das características do equipamento). Os dois últimos parâmetros, normalmente chamados de P e Q , determinam a forma da grade lógica do processo, Quer dizer, quantos processos no eixo x e quantos no eixo y . Para entender quais elementos de uma matriz de entrada A são armazenados em qual processo, podemos visualizar a matriz como sendo particionada em blocos. No caso simples em que mbP e nbQ dividem m e n , respectivamente, cada bloco tem tamanho uniforme. Cada bloco é composto por blocos $P \times Q$, cada um com tamanho $mb \times nb$. Finalmente, cada processo recebe uma posição na grade de blocos. O bloco nessa posição em cada bloco é armazenado no processo correspondente. Por exemplo, o bloco no local $(0, 0)$ de cada bloco pertence ao primeiro processo P_0 , o bloco no local $(0, 1)$ de cada bloco pertence ao segundo processo P_1 e assim por diante. Nos usamos uma distribuição que procure que P seja igual a Q .

1.3.3.7 MKL

A biblioteca Intel MKL (*Math Kernel Library*) (*oneAPI MKL*) é uma biblioteca de rotinas matemáticas otimizadas para aplicações científicas, de engenharia e estatística, projetada para ajudar aos programadores a aproveitar as vantagens da computação de alto desempenho (HPC) usando vários núcleos, multiprocessadores ou clusters. As funções matemáticas principais incluem BLAS, LAPACK, ScaLAPACK, solucionadores de sistemas esparsos, transformações rápidas de Fourier e matemática vetorial. Intel MKL é baseado em Intel C++ e Fortran usando OpenMP no *threading*. Esta biblioteca pode utilizar totalmente os multi-núcleos e multiprocessadores considerando o equilíbrio de carga. A versão Intel MKL do ScaLAPACK é otimizada para processadores Intel e usa a versão MPICH do MPI, bem como Intel MPI (MKL, 2021).

1.3.3.8 MAGMA

O projeto MAGMA (*Matrix Algebra on GPU and Multicore Architectures* ¹¹) tem como objetivo desenvolver uma biblioteca de alto desempenho para álgebra linear densa semelhante ao LAPACK, mas para arquiteturas heterogêneas/híbridas, começando com os atuais sistemas *Multi-core + GPU*. Isso significa que a CPU e a GPU estão envolvidas na execução do cálculo. Esta técnica é comprovada para atingir um desempenho muito alto em problemas de grande porte (Tomov et al., 2010).

A ideia por trás do desenvolvimento de MAGMA é que para enfrentar os desafios complexos dos ambientes híbridos emergentes, as soluções de software também terão que se hibridizar, combinando os pontos fortes de diferentes algoritmos em uma única *framework*. MAGMA é semelhante ao LAPACK em funcionalidade, armazenamento de dados e interface, a biblioteca MAGMA permite migrar de software desenhado para LAPACK facilmente e aproveitem as novas arquiteturas híbridas, além tem algumas rotinas com suporte para carga de trabalho em lote (*batched workloads*) (MAGMA, 2021).

Para cada função, existem 2 interfaces do estilo LAPACK. O primeiro, denominado interface da CPU, recebe a entrada e produz o resultado na memória da CPU. A segunda, conhecida como interface GPU, obtém a entrada e produz o resultado na memória da GPU. Neste trabalho, foram usadas rotinas com interface GPU.

¹¹A biblioteca esta disponível em: <http://icl.utk.edu/magma/>

2

Solução De Sistemas Lineares Hermitianos, Positivos Definidos (PD) E Densos De Grande Porte Com Múltiplos Lados Direitos: Contexto Teórico

Neste capítulo descrevemos os fundamentos teóricos e apresentamos um novo algoritmo paralelo por blocos para a solução de sistemas Hermitianos positivos definidos e densos com vários lados direitos. A solução do sistema é obtida recursivamente - usando a recursão de Levinson - combinando linearmente as soluções de menor ordem que estão relacionadas aos subsistemas *forward* e *backward* das equações normais de menor ordem. A nova implementação é descrita para computação paralela e baseada em um algoritmo de matriz por blocos. O algoritmo é separado em duas sub-rotinas, a primeira calcula a solução *backward* para as ordens $j = 1 \dots L$ e a matriz da energia dos erros, a segunda calcula a solução recursivamente. Implementamos este algoritmo para sistemas multiprocessadores e para sistemas com GPU. No primeiro caso, a solução de sistemas menores foi calculada usando SCALAPACK. Para sistemas com GPU, implementamos um algoritmo OUT-OF-CORE, onde os sistemas de ordem inferior foram calculados usando MAGMA. Em ambos os casos, a solução final é comparada com a solução do sistema completo usando SCALAPACK e MAGMA respectivamente.

2.1 O princípio básico da recursão de Levinson para sistemas simétricos

Originalmente a recursão de Levinson foi usada para resolver sistemas de equações normais (EN) simétricas Toeplitz (Levinson, 1946). A principal ideia por trás de um algoritmo tipo-Levinson é o cálculo de uma solução de ordem j baseada na combinação linear de duas soluções independentes de ordem $j - 1$, associadas a dois subsistemas de ordem menor.

A seguir, ilustramos como o princípio básico de Levinson pode ser aplicado para resolver um sistema não-Toeplitz de EN (Porsani e Ulrych, 1991).

Nós consideramos um sistema 2×2 de EN ,

$$\begin{bmatrix} v & z \\ z & w \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} = \begin{bmatrix} t \\ u \end{bmatrix}. \quad (2.1)$$

A 2.1 pode ser escrita como,

$$\begin{bmatrix} -t & v & z \\ -u & z & w \end{bmatrix} \begin{bmatrix} 1 \\ a_{2,1} \\ a_{2,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad (2.2)$$

onde os coeficientes $\{a_{2,1}, a_{2,2}\} = \{m_1, m_2\}$ representa a solução da Eq. 2.1.

A recursão começa por resolver $a_{1,1}$ e $b_{1,1}$, associadas a duas soluções independentes, a direita e a reversa (denotadas por F&B, *forward and backward*), os dois subconjuntos de equações associadas à primeira linha de equação 2.2,

$$[-t \ v \ z] \begin{bmatrix} 1 & 0 \\ a_{1,1} & b_{1,1} \\ 0 & 1 \end{bmatrix} = [0 \ 0]. \quad (2.3)$$

onde $a_{1,1} = v^{-1}t$ e $b_{1,1} = -v^{-1}z$ são as soluções F & B de ordem 1 e qualquer combinação linear e qualquer combinação linear dessas duas soluções independentes satisfaça a primeira equação,

$$[-t \ v \ z] \left\{ \begin{bmatrix} 1 & 0 \\ a_{1,1} & b_{1,1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \right\} = [0 \ 0] \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = 0. \quad (2.4)$$

Podemos definir $\alpha = 1$ e calcular β , de modo que a última equação de 2.2 esteja satisfeita:

$$[-u \ z \ w] \left\{ \begin{bmatrix} 1 & 0 \\ a_{1,1} & b_{1,1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ \beta \end{bmatrix} \right\} = [\Delta_1 \ Q_1] \begin{bmatrix} 1 \\ \beta \end{bmatrix} = 0, \quad (2.5)$$

onde $\Delta_1 = -u + za_{1,1}$ e $Q_1 = zb_{1,1} + w$.

Resolvendo para β , o resultado é:

$$\beta = \frac{\begin{bmatrix} -u & z \end{bmatrix} \begin{bmatrix} 1 \\ a_{1,1} \end{bmatrix}}{\begin{bmatrix} z & w \end{bmatrix} \begin{bmatrix} b_{1,1} \\ 1 \end{bmatrix}} = -\frac{\Delta_1}{Q_1}. \quad (2.6)$$

Ao usar as duas soluções independentes e se definimos $a_{2,2} = \beta$, podemos completar a solução da equação 2.2 como

$$\begin{bmatrix} 1 \\ a_{2,1} \\ a_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_{1,1} & b_{1,1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a_{2,2} \end{bmatrix}. \quad (2.7)$$

O mesmo procedimento pode ser usado para aumentar a solução da ordem $j - 1$ a j .

2.2 Extensão do princípio de Levinson para resolver sistemas bloco-particionado de EN com N lados direitos

Em seguida, é mostrado como a recursão de Levinson pode ser estendida para resolver o sistema Hermitiano particionado de NE (não necessariamente bloco-Toeplitz. Porsani et al., 2010). O procedimento é uma generalização do algoritmo Porsani e Ulrych, 1991.

Considerando um sistema complexo sobre-determinado e não singular de equações lineares expressado pela equação com N lados direitos, *NRHS* por suas siglas em inglês.

$$\mathbf{GM} = \mathbf{D}, \quad (2.8)$$

onde $\mathbf{G} \in \mathbb{C}^{m \times n}$ é uma matriz grande ($m > n$ sobre determinada), $\mathbf{D} \in \mathbb{C}^{m \times NRHS}$ é de posto completo e representa os dados observados e $\mathbf{M} \in \mathbb{C}^{n \times NRHS}$ é a solução do sistema de equações normais 2.8 que representa os múltiplos modelos. A estimativa da solução (modelos) de mínimos quadrados no caso sobre-determinado é dada por (Menke, 2012):

$$\mathbf{M} = \mathbf{M}_{est} = (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T \mathbf{D} = \mathbf{G}^{-g} \mathbf{D} \quad (2.9)$$

O sistema pode ser particionado como

$$\begin{bmatrix} \mathbf{G}_1 & \cdots & \mathbf{G}_j & \cdots & \mathbf{G}_L \end{bmatrix} \begin{bmatrix} \mathbf{M}_1 \\ \vdots \\ \mathbf{M}_j \\ \vdots \\ \mathbf{M}_L \end{bmatrix} = \mathbf{D}, \quad (2.10)$$

no qual as matrizes \mathbf{G}_j têm m linhas e N_c colunas (N_c é uma constante, o tamanho do bloco), e os sub-matriz \mathbf{M}_j têm N_c filas e $NRHS$ colunas. Como consequência do particionamento da matriz \mathbf{G} o sistema correspondente de equações normais (EN) terá uma matriz de coeficientes de bloco-Hermitiana como é mostrado a continuação para a ordem j ,

$$\begin{bmatrix} \emptyset \\ \vdots \\ \emptyset \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{A}_{11} & \cdots & \mathbf{A}_{1j} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_j & \mathbf{A}_{j1} & \cdots & \mathbf{A}_{jj} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j1} \\ \vdots \\ \mathbf{M}_{jj} \end{bmatrix}, \quad (2.11)$$

onde \mathbf{I}_{nr} é uma matriz identidade de tamanho $NRHS$, $\mathbf{B}_j = \mathbf{G}_j^* \mathbf{D}$ são sub-matrizes $N_c \times NRHS$, $\mathbf{A}_{i,j} = \mathbf{G}_i^* \mathbf{G}_j$ são matrizes quadradas $N_c \times N_c$. O asterisco representa o transposta conjugada e \emptyset é uma matriz nula $N_c \times NRHS$. \mathbf{A}_{jj} são matrizes quadradas hermitianas de tamanho $N_c \times N_c$.

2.2.1 Recursão de Levinson para solução de um sistema bloco-Hermitiano de EN

Aqui é apresentado o algoritmo de tipo Levinson para solução de um sistema de bloco-Hermitiano (não necessariamente Toeplitz) de EN (Porsani et al., 2010).

2.2.1.1 Solução de EN de ordem $j = 2$

Sendo:

$$\begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix} = \begin{bmatrix} -\mathbf{B}_1 & \mathbf{A}_{11} & \mathbf{A}_{12} \\ -\mathbf{B}_2 & \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{2,1} \\ \mathbf{M}_{2,2} \end{bmatrix}. \quad (2.12)$$

A recursão começa por resolver \mathbf{M}_{11} e ${}^1\mathbf{F}_{11}$, os dois subconjuntos de equações associadas à primeira linha de equação B.8:

$$[-\mathbf{B}_1 \quad \mathbf{A}_{11} \quad \mathbf{A}_{12}] \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathbf{M}_{1,1} & {}^1\mathbf{F}_{11} \\ \emptyset & \mathbf{I} \end{bmatrix} = [\emptyset \quad \oplus], \quad (2.13)$$

onde \oplus é uma matriz $N_c \times N_c$ com elementos nulos e \mathbf{I} é uma matriz identidade ($N_c \times N_c$). $\mathbf{M}_{1,1}$ e ${}^1\mathbf{F}_{11}$ são as soluções F & B (*forward* e *backward*) de ordem 1. O superindicado da direita 1 em ${}^1\mathbf{F}_{11}$ é usado para indicar a solução que tem a matriz de coeficientes \mathbf{A}_{11} . $\mathbf{M}_{1,1}$ é calculada pela resolução da equação de menor tamanho:

$$\mathbf{A}_{11} \mathbf{M}_{1,1} = \mathbf{B}_1, \quad (2.14)$$

e ${}^1\mathbf{F}_{11}$ é calculada pela resolução da equação

$$\mathbf{A}_{11} {}^1\mathbf{F}_{11} = -\mathbf{A}_{12} \quad (2.15)$$

Levando em consideração as mesmas considerações usadas para as equações 2.4, 2.5 e 2.3; a recursão de Levinson para a ordem 2 pode ser representada como:

$$\begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{2,1} \\ \mathbf{M}_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathbf{M}_{1,1} & {}^1\mathbf{F}_{11} \\ \emptyset & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{2,2} \end{bmatrix}. \quad (2.16)$$

A pré-multiplicação da equação 2.16 pela matriz associada ao sistema de ordem 2 e considerando os resultados obtidos pelas equações 2.13, ou dito de outra maneira, inserindo a equação 2.16 em a equação B.8 é obtido

$$\begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix} = \begin{bmatrix} -\mathbf{B}_1 & \mathbf{A}_{11} & \mathbf{A}_{12} \\ -\mathbf{B}_2 & \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathbf{M}_{1,1} & {}^1\mathbf{F}_{11} \\ \emptyset & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{2,2} \end{bmatrix} = \begin{bmatrix} \emptyset & \oplus \\ \Delta_{M1} & {}^1\mathbf{E}_{F1} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{2,2} \end{bmatrix}. \quad (2.17)$$

onde a solução da primeira equação da Eq. 2.17 não depende da matriz $\mathbf{M}_{2,2}$.

A matriz Δ_{M1} de tamanho $Nc \times NRHS$ e a matriz ${}^1\mathbf{E}_{F1}$ são obtidas pela multiplicação da solução de ordem 1 com a última linha da matriz (Eq. 2.17), ou seja:

$$\Delta_{M1} = -\mathbf{B}_2 + \mathbf{A}_{21}\mathbf{M}_{1,1} \quad (2.18)$$

e

$${}^1\mathbf{E}_{F1} = \mathbf{A}_{21}{}^1\mathbf{F}_{11} + \mathbf{A}_{22}, \quad (2.19)$$

onde ${}^1\mathbf{E}_{F1}$ é a matriz de energia do erro da solução direta para o primeiro sistema 2×2 de blocos.

Apenas a última parte da Eq. 2.17 deve ser resolvida para obter $\mathbf{M}_{2,2}$, portanto,

$$\mathbf{M}_{2,2} = -[{}^1\mathbf{E}_{F1}]^{-1} \Delta_{M1}. \quad (2.20)$$

Finalmente, usando este resultado (Eq 2.20), na Eq. 2.16, obtemos a solução da ordem 2 para a equação B.8.

2.2.1.2 Solução das EN para um ordem $j + 1$

O sistema a ser resolvido para a ordem $j + 1$ é representado na equação 2.21,

$$\begin{bmatrix} \emptyset \\ \vdots \\ \emptyset \\ \emptyset \end{bmatrix} = \begin{bmatrix} -\mathbf{B}_1 & \mathbf{A}_{11} & \cdots & \mathbf{A}_{1j} & \mathbf{A}_{1,j+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -\mathbf{B}_j & \mathbf{A}_{j1} & \cdots & \mathbf{A}_{jj} & \mathbf{A}_{j,j+1} \\ -\mathbf{B}_{j+1} & \mathbf{A}_{j+1,1} & \cdots & \cdots & \mathbf{A}_{j+1,j+1} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j+1,1} \\ \vdots \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}. \quad (2.21)$$

Dos primeiros j subconjuntos de equações de 2.21 é possível formar dois sistemas de equações de ordem j ,

$$\begin{bmatrix} \emptyset & \oplus \\ \vdots & \vdots \\ \emptyset & \oplus \end{bmatrix} = \begin{bmatrix} -\mathbf{B}_1 & \mathbf{A}_{11} & \cdots & \mathbf{A}_{1j} & \mathbf{A}_{1,j+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -\mathbf{B}_j & \mathbf{A}_{j1} & \cdots & \mathbf{A}_{jj} & \mathbf{A}_{j,j+1} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathcal{M}_j & {}^1\mathcal{F}_j \\ \emptyset & \mathbf{I} \end{bmatrix}, \quad (2.22)$$

onde \mathcal{M}_j , é o vetor solução para os primeiros j blocos de EN formado por vetores \mathbf{M}_{ji} com $i = 1, \dots, j$ de $N_c \times NRHS$:

$$\mathcal{M}_j = \begin{bmatrix} \mathbf{M}_{j1} \\ \vdots \\ \mathbf{M}_{jj} \end{bmatrix}, \quad (2.23)$$

e ${}^1\mathcal{F}_j$ representa a solução reversa (*backward*) de ordem j formada por matrizes quadradas ${}^1F_{ji}$ com $i = 1, \dots, j$ de $N_c \times N_c$ elementos, ou seja:

$${}^1\mathcal{F}_j = \begin{bmatrix} {}^1F_{jj} \\ \vdots \\ {}^1F_{j1} \end{bmatrix}. \quad (2.24)$$

A solução reversa de ordem j é representada na equação 2.25

$$\begin{bmatrix} \oplus \\ \vdots \\ \oplus \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1j} & \mathbf{A}_{1,j+1} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{A}_{j1} & \cdots & \mathbf{A}_{jj} & \mathbf{A}_{j,j+1} \end{bmatrix} \begin{bmatrix} {}^1\mathcal{F}_j \\ \mathbf{I} \end{bmatrix}. \quad (2.25)$$

O superindicado da direita 1 em ${}^1\mathcal{F}_j$ é usado para indicar a solução que tem \mathbf{A}_{11} como o primeiro elemento na diagonal principal da matriz de coeficientes.

A solução da equação 2.21 pode ser obtida por meio da recursão de Levinson a partir de uma combinação linear das duas soluções independentes representadas na equação 2.22:

$$\begin{bmatrix} \mathbf{I}_{nr} \\ \mathcal{M}_{j+1} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathcal{M}_j & {}^1\mathcal{F}_j \\ \emptyset & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}. \quad (2.26)$$

Substituindo a equação 2.26 na equação 2.21 e considerando os resultados fornecidos na equação 2.22, pode-se verificar que apenas o último conjunto de equações em 2.21 precisa ser resolvido para o vetor $\mathbf{m}_{j+1,j+1}$ como mostrado na equação 2.27,

$$\emptyset = \begin{bmatrix} -\mathbf{B}_{j+1} & \mathbf{A}_{j+1,1} & \cdots & \mathbf{A}_{j+1,j+1} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathcal{M}_j & {}^1\mathcal{F}_j \\ \emptyset & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j+1,j+1} \end{bmatrix} = \begin{bmatrix} \Delta_{Mj} & {}^1\mathbf{E}_{Fj} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}. \quad (2.27)$$

Analogamente à equação 2.17, as quantidades Δ_{mj} e ${}^1\mathbf{E}_{Fj}$ são obtidas multiplicando as soluções da ordem j pela última linha da matriz:

$$\Delta_{Mj} = [-\mathbf{B}_{j+1} \quad \mathbf{A}_{j+1,1} \quad \cdots \quad \mathbf{A}_{j+1,j}] \begin{bmatrix} \mathbf{I}_{nr} \\ \mathcal{M}_j \end{bmatrix}, \quad (2.28)$$

$${}^1\mathbf{E}_{Fj} = [\mathbf{A}_{j+1,1} \quad \cdots \quad \mathbf{A}_{j+1,j+1}] \begin{bmatrix} {}^1\mathcal{F}_j \\ \mathbf{I} \end{bmatrix}. \quad (2.29)$$

A solução da equação 2.27 pode ser representada como

$$\mathbf{M}_{j+1,j+1} = -[{}^1\mathbf{E}_{Fj}]^{-1} \Delta_{Mj}. \quad (2.30)$$

Usando 2.30 na equação 2.26 obtemos a solução da equação 2.21.

Na próxima seção, é apresentado o procedimento para obter $\{{}^1\mathcal{F}_j, {}^1\mathbf{E}_{Fj}\}$, $j = 1, \dots, L-1$, que são necessários para obter a solução completa.

2.2.2 Procedimento para obter a solução reversa e a matriz de energia do erro correspondente à solução reversa

Para começar é preciso resolver os subsistemas F & B (2×2) distribuídos ao longo da diagonal principal,

$$\begin{bmatrix} \mathbf{A}_{i,i} & \mathbf{A}_{i,i+1} \\ \mathbf{A}_{i+1,i} & \mathbf{A}_{i+1,i+1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathbf{F}_{11} \\ {}^i\mathbf{H}_{11} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} {}^i\mathbf{E}_{H1} & \oplus \\ \oplus & {}^i\mathbf{E}_{F1} \end{bmatrix} \quad (2.31)$$

Calculando as matrizes ${}^i\mathbf{H}_{11}$ e ${}^i\mathbf{F}_{11}$ como

$$\mathbf{A}_{i+1,i+1} {}^i\mathbf{H}_{11} = -\mathbf{A}_{i+1,i}, \quad (2.32)$$

$$\mathbf{A}_{i,i} {}^i\mathbf{F}_{11} = -\mathbf{A}_{i,i+1}, \quad (2.33)$$

é possível calcular as matrizes de energia de erro ${}^i\mathbf{E}_{H1}$ e ${}^i\mathbf{E}_{F1}$ no lado direito da equação para a solução direita e reversa da Eq. 2.31 como:

$${}^i\mathbf{E}_{H1} = \mathbf{A}_{i,i} + \mathbf{A}_{i,i+1} {}^i\mathbf{H}_{11}, \quad (2.34)$$

$${}^i\mathbf{E}_{F1} = \mathbf{A}_{i+1,i+1} + \mathbf{A}_{i+1,i} {}^i\mathbf{F}_{11}. \quad (2.35)$$

2.2.2.1 Solução dos subsistemas de ordem j

Os subsistemas F & B de ordem j são representados abaixo

$$\begin{bmatrix} \mathbf{A}_{i,i} & \cdots & \mathbf{A}_{i,i+j} \\ \mathbf{A}_{i+j,i} & \cdots & \mathbf{A}_{i+j,i+j} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathcal{F}_j \\ {}^i\mathcal{H}_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} {}^i\mathbf{E}_{Hj} & \oplus_j \\ \oplus_j & {}^i\mathbf{E}_{Fj} \end{bmatrix}, \quad (2.36)$$

onde

$$(i) {}^i\mathcal{H}_j = \begin{bmatrix} {}^i\mathbf{H}_{j1} \\ \vdots \\ {}^i\mathbf{H}_{jj} \end{bmatrix}, \quad (ii) {}^i\mathcal{F}_j = \begin{bmatrix} {}^i\mathbf{F}_{jj} \\ \vdots \\ {}^i\mathbf{F}_{j1} \end{bmatrix}, \quad (iii) \oplus_j = \begin{bmatrix} \oplus_{11} \\ \vdots \\ \oplus_{jj} \end{bmatrix} \quad (2.37)$$

a eq. 2.37(i) representa a solução direita (F) de ordem j formada por j matrizes quadradas ($N_c \times N_c$) ${}^i\mathbf{H}_{jk}$; a Eq. 2.37(ii) representa a solução reversa (B) de ordem j formada por j matrizes quadradas ($N_c \times N_c$) ${}^i\mathbf{F}_{jk}$; a Eq. 2.37(iii) é uma matriz formada por j matrizes quadradas ($N_c \times N_c$) nulas.

As soluções F & B da equação 2.36 podem ser obtidas a partir da recursão de Levinson

$$\begin{bmatrix} \mathbf{I} & {}^i\mathcal{F}_j \\ {}^i\mathcal{H}_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^i\mathcal{H}_{j-1} & {}^{i+1}\mathcal{F}_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathbf{F}_{jj} \\ {}^i\mathbf{H}_{jj} & \mathbf{I} \end{bmatrix}, \quad (2.38)$$

ou seja:

$${}^i\mathcal{H}_j = \begin{bmatrix} {}^i\mathbf{H}_{j1} \\ \vdots \\ {}^i\mathbf{H}_{jj} \end{bmatrix} = \begin{bmatrix} {}^i\mathcal{H}_{j-1} & {}^{i+1}\mathcal{F}_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} \\ {}^i\mathbf{H}_{jj} \end{bmatrix} \quad (2.39)$$

e

$${}^i\mathcal{F}_j = \begin{bmatrix} {}^i\mathbf{F}_{jj} \\ \vdots \\ {}^i\mathbf{F}_{j1} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^i\mathcal{H}_{j-1} & {}^{i+1}\mathcal{F}_{j-1} \end{bmatrix} \begin{bmatrix} {}^i\mathbf{F}_{jj} \\ \mathbf{I} \end{bmatrix}. \quad (2.40)$$

Pré-multiplicando a Eq. 2.38 pela matriz de coeficientes dada na equação 2.36 e considerando as soluções F & B já disponíveis ${}^i\mathcal{H}_{j-1}$, ${}^{i+1}\mathcal{F}_{j-1}$ o resultado é

$$\begin{bmatrix} \mathbf{A}_{i,i} & \cdots & \mathbf{A}_{i,i+j} \\ \mathbf{A}_{i+j,i} & \cdots & \mathbf{A}_{i+j,i+j} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathcal{F}_j \\ {}^i\mathcal{H}_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} {}^i\mathbf{E}_{Hj-1} & {}^{i+1}\mathbf{\Delta}_{Fj-1} \\ \oplus_{j-1} & \oplus_{j-1} \\ {}^i\mathbf{\Delta}_{Hj-1} & {}^{i+1}\mathbf{E}_{Fj-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathbf{F}_{jj} \\ {}^i\mathbf{H}_{jj} & \mathbf{I} \end{bmatrix}. \quad (2.41)$$

As matrizes ${}^i\mathbf{E}_{Hj-1}$, ${}^{i+1}\mathbf{\Delta}_{Fj-1}$, ${}^i\mathbf{\Delta}_{Hj-1}$ e ${}^{i+1}\mathbf{E}_{Fj-1}$ são calculadas combinando a Eq 2.41 com a Eq 2.38, conhecendo previamente as matrizes ${}^i\mathcal{H}_{j-1}$ e ${}^{i+1}\mathcal{F}_{j-1}$.

Como consequência da propriedade hermitiana da matriz de coeficientes, pode-se verificar que

$${}^{i+1}\mathbf{\Delta}_{Fj-1} = {}^i\mathbf{\Delta}_{Hj-1}^* \quad (2.42)$$

onde

$${}^i\mathbf{\Delta}_{Hj-1} = \mathbf{A}_{i+j,i} + \sum_{k=1}^{j-1} \mathbf{A}_{i+j,i+k} {}^i\mathbf{H}_{j-1,k} \quad (2.43)$$

Combinando as Eq 2.36 e Eq 2.41, é obtida uma forma compacta e recursiva:

$$\begin{bmatrix} {}^i\mathbf{E}_{Hj-1} & {}^i\mathbf{\Delta}_{Hj-1}^* \\ {}^i\mathbf{\Delta}_{Hj-1} & {}^{i+1}\mathbf{E}_{Fj-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathbf{F}_{jj} \\ {}^i\mathbf{H}_{jj} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} {}^i\mathbf{E}_{Hj} & \oplus \\ \oplus & {}^i\mathbf{E}_{Fj} \end{bmatrix}, \quad (2.44)$$

onde ${}^i\mathbf{E}_{Hj}$ e ${}^i\mathbf{E}_{Fj}$ correspondem às matrizes de energia do erro para as soluções ${}^i\mathcal{H}_j$ e ${}^i\mathcal{F}_j$ respectivamente. A Eq 2.44 tem que ser resolvida para ${}^i\mathbf{H}_{jj}$ e ${}^i\mathbf{F}_{jj}$:

$${}^{i+1}\mathbf{E}_{Fj-1} {}^i\mathbf{H}_{jj} = -{}^i\mathbf{\Delta}_{Hj-1}, \quad (2.45)$$

$${}^i\mathbf{E}_{Hj-1} {}^i\mathbf{F}_{jj} = -{}^i\mathbf{\Delta}_{Hj-1}^*, \quad (2.46)$$

Ou seja, a solução para ${}^i\mathbf{H}_{jj}$ e ${}^i\mathbf{F}_{jj}$ é dada por:

$${}^i\mathbf{H}_{jj} = -[{}^{i+1}\mathbf{E}_{Fj-1}]^{-1} {}^i\mathbf{\Delta}_{Hj-1}, \quad (2.47)$$

$${}^i\mathbf{F}_{jj} = -[{}^i\mathbf{E}_{Hj-1}]^{-1} {}^i\mathbf{\Delta}_{Hj-1}^*, \quad (2.48)$$

e esses resultados podem ser usados para atualizar as matrizes de energia de erro no lado direito da equação 2.44. As expressões para atualizar as matrizes de energia do erro são:

$${}^i\mathbf{E}_{Hj} = {}^i\mathbf{E}_{Hj-1} + {}^i\mathbf{\Delta}_{Hj-1}^* {}^i\mathbf{H}_{jj} \quad (2.49)$$

$${}^i\mathbf{E}_{Fj} = {}^{i+1}\mathbf{E}_{Fj-1} + {}^i\mathbf{\Delta}_{Hj-1} {}^i\mathbf{F}_{jj}. \quad (2.50)$$

Ao substituir a expressão para ${}^i\mathbf{H}_{jj}$ e ${}^i\mathbf{F}_{jj}$ dada por as Eqs, 2.47 e 2.48 em a Eq 2.49 e 2.50, respectivamente, pode-se verificar se as matrizes de energia do erro também são Hermitianas.

Ao resolver para $j = 1, \dots, L - 1$ e $i = 1, \dots, L - j - 1$, todos os subconjuntos dos sistemas formados ao longo da diagonal principal da matriz de coeficientes, pode-se obter as soluções ${}^1\mathcal{F}_j$ e suas matrizes de energia do erro correspondentes, ${}^1\mathbf{E}_{Fj}$, que são requeridas nas equações 2.26, 2.27 e 2.30 para calcular a solução do sistema de EN particionado da ordem 1 até a ordem L.

A figura 2.1 ilustra a estrutura completa do algoritmo para a ordem $L = 5$. As setas e as cruzes indicam a relação entre as duas soluções da ordem j (no lado esquerdo), necessárias para calcular a solução da ordem $j + 1$ (no lado direito).

A Seguir, apresentamos a descrição dos parâmetros da figura 2.1

- $\mathcal{M}_1, {}^1\mathcal{F}_1, \{{}^i\mathcal{H}_1, {}^{i+1}\mathcal{F}_1, \}, i = 1, 2, 3$ correspondem às soluções dos subsistemas de ordem 1 que ocorrem ao longo da diagonal principal das EN.
- $\mathcal{M}_2, {}^1\mathcal{F}_2, \{{}^i\mathcal{H}_2, {}^{i+1}\mathcal{F}_2, \}, i = 1, 2$ correspondem às soluções dos subsistemas da ordem 2.

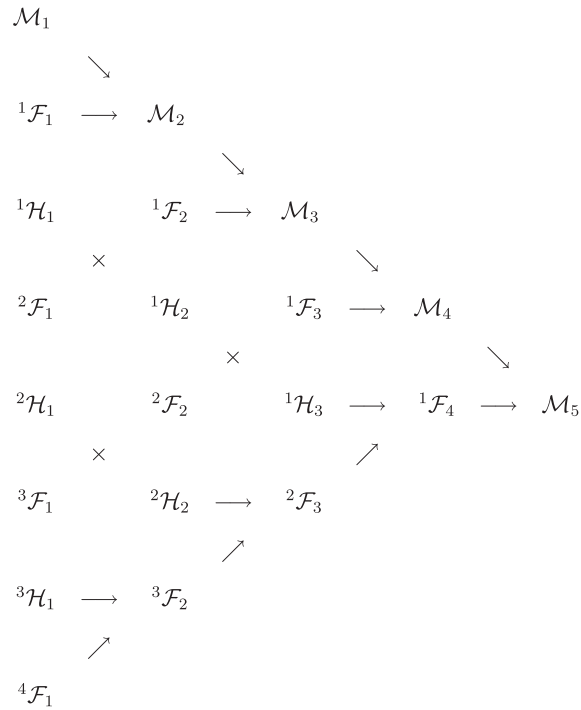


Figura 2.1: Representação esquemática do algoritmo de tipo Levinson para solução de um sistema bloco-hermitiano de EN de ordem 5. A figura apresenta a distribuição espacial e a relação entre os coeficientes.

- $\mathcal{M}_3, {}^1\mathcal{F}_3, \{ {}^i\mathcal{H}_3, {}^{i+1}\mathcal{F}_3, \}, i = 1, 2$ correspondem às soluções dos subsistemas da ordem 3.
- $\mathcal{M}_4, {}^1\mathcal{F}_4$ Corresponde às soluções dos subsistemas da ordem 4 que são usadas para obter a solução final da ordem 5 \mathcal{M}_5 .

2.2.3 Algoritmo para implementação serial

As etapas completas do algoritmo serial para solução do sistema bloco-hermitiano de EN são apresentadas no pseudocódigo 1, este algoritmo é equivalente ao mostrado por Porsani et al., 2010, possui apenas uma reorganização dos termos para evitar um *loop*.

Algoritmo 1 Algoritmo serial para solução do sistema bloco-hermitiano de EN

Require: $\mathbf{A}(L \times N_c, L \times N_c)$: sistema bloco-hermitiano de EN;

$\mathbf{B}(L \times N_c, NRHS)$: $\mathbf{G}^* \mathbf{D}$;

1: Inicializar

2: $\mathbf{M}_{11} \leftarrow \mathbf{A}_{11} \mathbf{M}_{11} = \mathbf{B}_1$

3: **for** $i = 1, L - 1$ **do**

4: ${}^i \mathbf{E}_{H0} = \mathbf{A}_{ii}$

5: ${}^{i+1} \mathbf{E}_{F0} = \mathbf{A}_{i+1,i+1}$

6: **end for**

7: **for** $j = 1, L - 1$ **do**

8: **for** $i = 1, L - j - 1$ **do**

9: ${}^i \Delta_{Hj-1} = \mathbf{A}_{i+j,i} + \sum_{k=1}^{j-1} \mathbf{A}_{i+j,i+k} {}^i \mathbf{H}_{j-1,k}$

10: ${}^{i+1} \Delta_{Fj-1} = {}^i \Delta_{Hj-1}^*$

11: ${}^i \mathbf{H}_{jj} \leftarrow {}^{i+1} \mathbf{E}_{Fj-1} {}^i \mathbf{H}_{jj} = -{}^i \Delta_{Hj-1}$

12: ${}^i \mathbf{F}_{jj} \leftarrow {}^i \mathbf{E}_{Hj-1} {}^i \mathbf{F}_{jj} = -{}^i \Delta_{Hj-1}^*$

13: ${}^i \mathbf{E}_{Hj} = {}^i \mathbf{E}_{Hj-1} + {}^i \Delta_{Hj-1}^* {}^i \mathbf{H}_{jj}$

14: ${}^i \mathbf{E}_{Fj} = {}^{i+1} \mathbf{E}_{Fj-1} + {}^i \Delta_{Hj-1} {}^i \mathbf{F}_{jj}$

15: $\{ {}^i \mathcal{H}_j, {}^i \mathcal{F}_j \} \leftarrow \begin{bmatrix} \mathbf{I} & {}^i \mathcal{F}_j \\ {}^i \mathcal{H}_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^i \mathcal{H}_{j-1} & {}^{i+1} \mathcal{F}_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i \mathbf{F}_{jj} \\ {}^i \mathbf{H}_{jj} & \mathbf{I} \end{bmatrix}$

16: **end for**

17: ${}^{L-j} \Delta_{Hj-1} = \mathbf{A}_{L,L-j} + \sum_{k=1}^{j-1} \mathbf{A}_{L,L-j+k} {}^{L-j} \mathbf{H}_{j-1,k}$

18: ${}^{L-j+1} \Delta_{Fj-1} = {}^{L-j} \Delta_{Hj-1}^*$

19: ${}^{L-j} \mathbf{F}_{jj} \leftarrow {}^{L-j} \mathbf{E}_{Hj-1} {}^{L-j} \mathbf{F}_{jj} = -{}^{L-j} \Delta_{Hj-1}^*$

20: ${}^{L-j} \mathbf{E}_{Fj} = {}^{L-j+1} \mathbf{E}_{Fj-1} + {}^{L-j} \Delta_{Hj-1} {}^{L-j} \mathbf{F}_{jj}$

21: ${}^{L-j} \mathcal{F}_j = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^{L-j} \mathcal{H}_{j-1} & {}^{L-j+1} \mathcal{F}_{j-1} \end{bmatrix} \begin{bmatrix} {}^{L-j} \mathbf{F}_{jj} \\ \mathbf{I} \end{bmatrix}$

22: $\Delta_{Mj} = \begin{bmatrix} -\mathbf{B}_{j+1} & \mathbf{A}_{j+1,1} & \cdots & \mathbf{A}_{j+1,j} \end{bmatrix} \begin{bmatrix} 1 \\ \mathcal{M}_j \end{bmatrix}$

23: $\mathbf{M}_{j+1,j+1} \leftarrow {}^1 \mathbf{E}_{Fj} \mathbf{M}_{j+1,j+1} = \Delta_{Mj}$

24: $\begin{bmatrix} 1 \\ \mathcal{M}_{j+1} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathcal{M}_j & {}^1 \mathcal{F}_j \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}$

25: **end for**

26: **return** \mathcal{M}_L

Observe-se que, para o caso mono canal ($N_c = 1$), todas as matrizes $\{ \mathbf{H}_{jj}, \mathbf{F}_{jj}, \mathbf{E}_{Hj}, \mathbf{E}_{Fj} \}$ reduzem-se a escalares e o código reduz para o algoritmo de Porsani e Ulrych, 1991, para resolver um sistema simétrico de EN.

Ao impor $i = 1$ no algoritmo anterior, ele é reduzido à recursão clássica de Levinson para resolver o sistema bloco-Toeplitz de EN, usado para calcular o filtro preditivo multicanal

Wiener-Levinson (Wiggins e Robinson, 1965; Treitel, 1970; Porsani e Ursin, 2007).

Esta implementação serial, onde são resolvidos todos os problemas de ordem 2, depois os de ordem 3 e assim sucessivamente até o sistema de ordem L; Envolve o uso de uma grande quantidade de memória de trabalho para armazenar as soluções parciais de sistemas de ordem inferior, o que torna o algoritmo pouco atraente para problemas grandes.

2.2.4 Algoritmo para implementação paralela

A Figura 2.2 ilustra a estrutura do algoritmo da implementação paralela para a solução de um sistema linear particionado de ordem L. Os elementos \mathcal{M}_j , \mathcal{D}_j e \mathcal{F}_j representam soluções de subsistemas menores. O algoritmo gera a solução final fazendo combinação linear das soluções menores, associadas às partições do sistema original. Sistemas de equações com dezenas ou centenas de milhares de equações podem ser particionados e as soluções dos subsistemas menores podem ser obtidas em processadores individuais, dividindo a carga computacional e reduzindo o tempo da inversão.

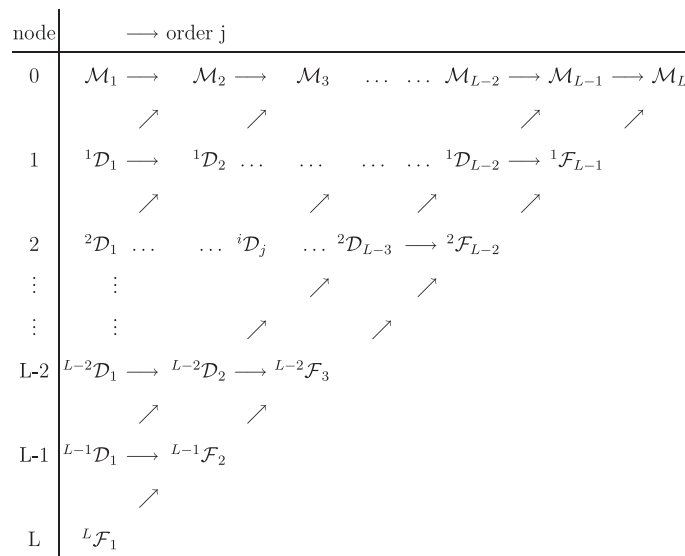


Figura 2.2: Representação esquemática mostrando o encadeamento das soluções menores que pode ser explorada por multiprocessadores. Ao final de L passos (L= número de partições do sistema original) a solução final é obtida.

Cabe salientar, que \mathcal{M}_j implica o cálculo de Δ_{M_j} (Eq. 2.28), \mathbf{M}_{jj} (Eq. 2.30) e a consequente atualização de \mathcal{M}_j (Eq. 2.26). ${}^i\mathcal{D}_j$ faz referência a resolver a Eq. 2.44, ou seja, resolver ${}^i\mathbf{H}_{jj}$ e ${}^i\mathbf{F}_{jj}$ (Eqs. 2.45 e 2.46), além de atualizar ${}^i\mathbf{E}_{Hj}$, ${}^i\mathbf{E}_{Fj}$ (Eqs. 2.49, 2.50) e $\{{}^i\mathcal{H}_j, {}^i\mathcal{F}_j\}$ usando a Eq. 2.38. ${}^i\mathcal{F}_j$ implica obter ${}^i\mathbf{F}_{jj}$ (Eq. 2.46), atualizar ${}^i\mathbf{E}_{Fj}$ (Eq. 2.50) para poder atualizar ${}^i\mathcal{F}_j$ usando a Eq. 2.40.

No algoritmo 2 abaixo, é apresentado de forma explícita, a implementação paralela

proposta por Porsani et al., 2010, para sistemas de memória distribuída sem as funções MPI (só para facilidade de entendimento). Neste algoritmo não está apresentado o procedimento para construção do sistema bloco-hermitiano de EN, apenas a solução do sistema bloco-hermitiano $\mathbf{AM} = \mathbf{B}$, ($\mathbf{A} = \mathbf{G}^T \mathbf{G}$,).

Algoritmo 2 Algoritmo Paralelo para solução do sistema bloco-hermitiano de EN

Require: $\mathbf{A}(L \times N_c, L \times N_c)$: sistema BH de EN;

$\mathbf{B}(L \times N_c, NRHS)$: $\mathbf{G}^* \mathbf{D}$; $id = i$ Rank ; $np = L \#$ total de processos.

```

1: if ( $id = i$ ) > 0 then
2:    ${}^i \mathbf{E}_{H0} = \mathbf{A}_{ii}$ 
3:    ${}^{i+1} \mathbf{E}_{F0} = \mathbf{A}_{i+1,i+1}$ 
4: end if
5: for  $j = 1, np - id$  do
6:   if ( $id = i$ ) == 0 then
7:     if  $j == 1$  then
8:        $\mathbf{M}_{11} \leftarrow \mathbf{A}_{11} \mathbf{M}_{11} = \mathbf{B}_1$ 
9:     else
10:       $\Delta_{Mj} = [-\mathbf{B}_{j+1} \quad \mathbf{A}_{j+1,1} \quad \cdots \quad \mathbf{A}_{j+1,j}] \begin{bmatrix} 1 \\ \mathcal{M}_j \end{bmatrix}$ 
11:       $\mathbf{M}_{j+1,j+1} \leftarrow {}^1 \mathbf{E}_{Fj} \mathbf{M}_{j+1,j+1} = \Delta_{Mj}$ 
12:       $\begin{bmatrix} 1 \\ \mathcal{M}_{j+1} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathcal{M}_j & {}^1 \mathcal{F}_j \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}$ 
13:    end if
14:    else if  $id \geq 1 \ \& \ id \leq L - j$  then
15:       ${}^i \Delta_{Hj-1} = \mathbf{A}_{i+j,i} + \sum_{k=1}^{j-1} \mathbf{A}_{i+j,i+k} {}^i \mathbf{H}_{j-1,k}$ 
16:       ${}^{i+1} \Delta_{Fj-1} = {}^i \Delta_{Hj-1}^*$ 
17:      if  $id \leq L - j$  then
18:         ${}^i \mathbf{H}_{jj} \leftarrow {}^{i+1} \mathbf{E}_{Fj-1} {}^i \mathbf{H}_{jj} = -{}^i \Delta_{Hj-1}$ 
19:         ${}^i \mathbf{E}_{Hj} = {}^i \mathbf{E}_{Hj-1} + {}^i \Delta_{Hj-1}^* {}^i \mathbf{H}_{jj}$ 
20:      end if
21:       ${}^i \mathbf{F}_{jj} \leftarrow {}^i \mathbf{E}_{Hj-1} {}^i \mathbf{F}_{jj} = -{}^i \Delta_{Hj-1}^*$ 
22:       ${}^i \mathbf{E}_{Fj} = {}^{i+1} \mathbf{E}_{Fj-1} + {}^i \Delta_{Hj-1} {}^i \mathbf{F}_{jj}$ 
23:      if  $id \leq L - j$  then
24:         ${}^i \mathcal{H}_j = \begin{bmatrix} {}^i \mathcal{H}_{j-1} & {}^{i+1} \mathcal{F}_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} \\ {}^i \mathbf{H}_{jj} \end{bmatrix}$ 
25:      end if
26:       ${}^i \mathcal{F}_j = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^i \mathcal{H}_{j-1} & {}^{i+1} \mathcal{F}_{j-1} \end{bmatrix} \begin{bmatrix} {}^i \mathbf{F}_{jj} \\ \mathbf{I} \end{bmatrix}$ 
27:    end if
28:  end for
29: return  $\mathcal{M}_L$ 

```

No algoritmo 2, as matrizes em vermelho são enviadas ao processo $id - 1$. As matrizes

em azul são recebidos do processador $id + 1$ para $j > 1$, e as matrizes em verde são mantidas no próprio nó para uso no passo seguinte ao ser incrementada a ordem da solução.

O método representa a aplicação do princípio de Levinson de construir a solução da equação de ordem $j + 1$ a partir de soluções menores de ordem j . Sua arquitetura é naturalmente apropriada para emprego de sistemas de computação com multiprocessadores, na redução do tempo computacional da solução. Sistemas de equações com dezenas ou centenas de milhares de equações podem ser particionados e a solução dos subsistemas menores podem ser obtidas em processadores individuais, dividindo a carga computacional e reduzindo o tempo total da solução.

O algoritmo pode ser aplicado diretamente sobre o sistema $\mathbf{AM} = \mathbf{B}$, onde a matriz \mathbf{A} é a matriz positiva definida associada ao problema de modelagem direta. Também pode ser aplicado diretamente sobre o sistema linear particionado, sem a necessidade de obtenção da matriz \mathbf{A} , sendo mais apropriado para a inversão através do método de Gauss-Newton ou de Newton.

Esta versão paralela do algoritmo, projetado para implementação em sistemas multi-CPU, usando por exemplo MPI, reduz fortemente o uso de memória de trabalho, mas tem a desvantagem de não aproveitar de forma integral os processadores disponíveis no sistema. Note que conforme ilustrado na figura 2.2, enquanto a ordem (j) do problema aumenta, o número de nós computacionais usados diminui. Portanto, a eficiência máxima seria apenas de 50 %. É possível implementar esta metodologia para que à medida que os nós de computação sejam liberados, eles sejam utilizados na próxima etapa, mas seria um grande desafio de programação redistribuir os dados a cada novo nível da solução.

2.2.5 A nova implementação proposta

Com o objetivo de tornar a metodologia apresentada competitiva com outros algoritmos, em particular com a decomposição de Cholesky, apresentamos uma nova implementação deste algoritmo que reduz significativamente o uso de memória de trabalho e adicionalmente permite a utilização de todos os recursos computacionais, além de descrever uma implementação OUT OF THE CORE para uso de GPU.

Esta nova implementação do algoritmo é bloco serial, onde cada passo, multiplicações de matrizes e resolução de pequenas equações do sistema $N_c \times N_c$ lineares, pode ser executado em paralelo, seja com múltiplos *threads*¹, múltiplos nós² ou usando GPU³. Esta implementação é

¹ex. Usando OpenMP com openBLAS ou biblioteca MKL

²ex: usando MPI com a biblioteca SCALAPACK

³ex: usando a biblioteca MAGMA

apresentada na forma de duas sub-rotinas, onde a primeira (Algoritmo 3) depende apenas da matriz \mathbf{A} e calcula a fatoração de Cholesky das matrizes de energia do erro ($LL^*({}^1\mathbf{E}_{Fj=0,\dots,L-1})$) correspondendo a ${}^1\mathcal{F}_{j=1,\dots,L-1}$ e, obviamente ${}^1\mathcal{F}_{j=1,\dots,L-1}$. A segunda sub-rotina (Algoritmo 4) usa esta informação, a matriz \mathbf{A} e o RHS para calcular a solução do sistema.

Algoritmo 3 Algoritmo para solução de sistema bloco-hermitiano de EN, sub-rotina 1, fatoração

Require: $\mathbf{A}(L \times N_c, L \times N_c)$: Sistema bloco-hermitiano de NE;

$\mathbf{B}(L \times N_c, NRHS)$: $\mathbf{G}^* \mathbf{D}$;

```

1: Initialize
2:  ${}^1\mathbf{E}_{F0} \leftarrow \mathbf{A}_{11}$ 
3: for  $i = 1, L - 1$  do
4:    ${}^i\mathbf{E}_{H0} = \mathbf{A}_{ii}$ 
5: end for
6: for  $p = 1, L - 1$  do
7:    ${}^{p+1}\mathbf{E}_{F0} = \mathbf{A}_{p+1,p+1}$ 
8:   for  $i = p, 1, -1$  do
9:      $j = p - i + 1$ 
10:     ${}^i\Delta_{Hj-1} = \mathbf{A}_{i+j,i} + \sum_{k=1}^{j-1} \mathbf{A}_{i+j,i+k} {}^i\mathbf{H}_{j-1,k}$ 
11:     ${}^{i+1}\Delta_{Fj-1} = {}^i\Delta_{Hj-1}^*$ 
12:    if  $p < L - 1$  then
13:       ${}^i\mathbf{H}_{jj} \leftarrow {}^{i+1}\mathbf{E}_{Fj-1} {}^i\mathbf{H}_{jj} = -{}^i\Delta_{Hj-1}$ 
14:       ${}^i\mathbf{E}_{Hj} = {}^i\mathbf{E}_{Hj-1} + {}^i\Delta_{Hj-1}^* {}^i\mathbf{H}_{jj}$ 
15:    end if
16:     ${}^i\mathbf{F}_{jj} \leftarrow {}^i\mathbf{E}_{Hj-1} {}^i\mathbf{F}_{jj} = -{}^i\Delta_{Hj-1}^*$ 
17:     ${}^i\mathbf{E}_{Fj} = {}^{i+1}\mathbf{E}_{Fj-1} + {}^i\Delta_{Hj-1} {}^i\mathbf{F}_{jj}$ 
18:     ${}^i\mathcal{F}_j = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^i\mathcal{H}_{j-1} & {}^{i+1}\mathcal{F}_{j-1} \end{bmatrix} \begin{bmatrix} {}^i\mathbf{F}_{jj} \\ \mathbf{I} \end{bmatrix}$ 
19:    if  $p < L - 1$  then
20:       ${}^i\mathcal{H}_j = \begin{bmatrix} {}^i\mathcal{H}_{j-1} & {}^{i+1}\mathcal{F}_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} \\ {}^i\mathbf{H}_{jj} \end{bmatrix}$ 
21:    end if
22:  end for
23: end for
24: for  $j = 0, L - 1$  do
25:    $LL^*({}^1\mathbf{E}_{Fj}) \leftarrow LL^*({}^1\mathbf{E}_{Fj})$ 
26: end for
27: return  ${}^1\mathcal{F}_{j=1,\dots,L-1}; LL^*({}^1\mathbf{E}_{Fj=0,\dots,L-1})$ ;

```

A figura 2.3 mostra como os blocos de \mathbf{A} são adicionados no código pelo algoritmo em cada estágio. Na proposta original (Porsani et al., 2010) a sequência de adição de blocos era feita da esquerda para a direita e de cima para baixo, nesta nova implementação a sequência de adição é de cima para baixo e da esquerda para a direita.

Algoritmo 4 Algoritmo para solução do sistema bloco-hermitiano de NE, sub-rotina 2, solução de múltiplos RHS

Require: $\mathbf{A}(L \times N_c, L \times N_c)$: Sistema bloco-hermitiano de NE;

$${}^1\mathcal{F}_{j=1,\dots,L-1};$$

$$LL^*({}^1\mathbf{E}_{Fj=0,\dots,L-1});$$

$\mathbf{B}(L \times N_c, NRHS)$: $\mathbf{G}^* \mathbf{D}$;

1: $\mathbf{M}_{11} \leftarrow LL^*({}^1\mathbf{E}_{F0}) \mathbf{M}_{11} = \mathbf{B}_1$

2: **for** $j = 1, L - 1$ **do**

3: $\Delta_{Mj} = [-\mathbf{B}_{j+1} \quad \mathbf{A}_{j+1,1} \quad \cdots \quad \mathbf{A}_{j+1,j}] \begin{bmatrix} \mathbf{I}_{nr} \\ \mathcal{M}_j \end{bmatrix}$

4: $\mathbf{M}_{j+1,j+1} \leftarrow LL^*({}^1\mathbf{E}_{Fj}) \mathbf{M}_{j+1,j+1} = \Delta_{Mj}$

5: $\begin{bmatrix} \mathbf{I}_{nr} \\ \mathcal{M}_{j+1} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{nr} & \mathbf{O}^T \\ \mathcal{M}_j & {}^1\mathcal{F}_j \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}$

6: **end for**

7: **return** $\mathcal{M}_L(L * N_c, NRHS)$

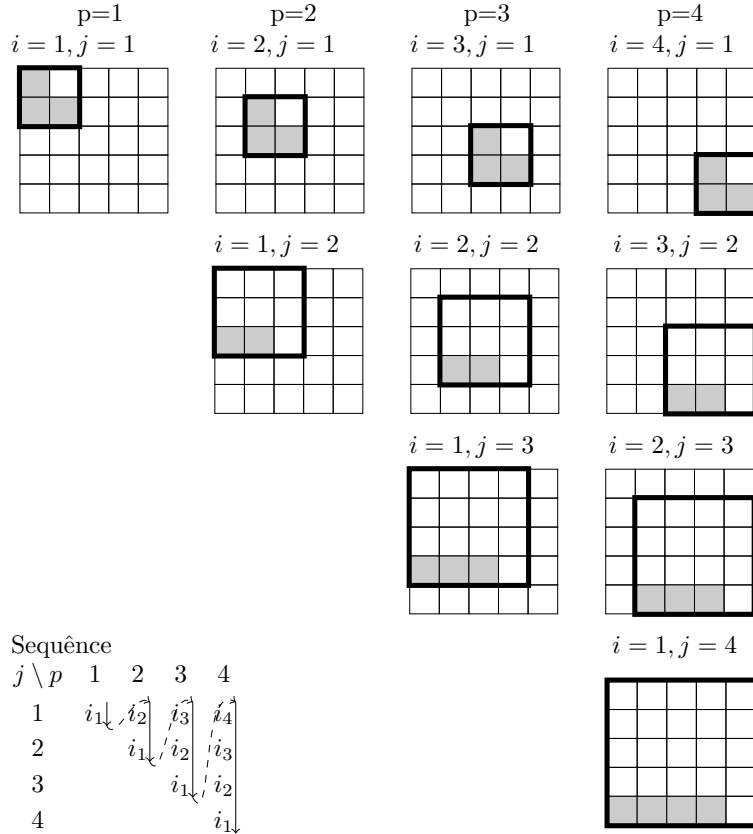


Figura 2.3: Esquema mostrando como as informações por blocos da matriz \mathbf{A} são incorporadas ao algoritmo 3 em cada etapa. De cima para baixo e da esquerda para a direita. O quadrado com a linha preta grossa representa a ordem do sistema a ser resolvido e os pequenos quadrados sombreados são as informações de \mathbf{A} incorporadas no algoritmo.

Por outro lado, na figura 2.4 representamos como o algoritmo 4 incorpora gradativamente as informações da matriz $\mathbf{A}_{i,j}$, RHS (\mathbf{B}_j), fatoração de Cholesky da matriz energética do erro ($LL^*_{(\mathbf{E}_{Fj=0,\dots,L-1})}$) e solução Backward (${}^1\mathcal{F}_{j=1,\dots,L-1}$) para construir a solução final (\mathcal{M}_L) do sistema de NE. A primeira linha da figura corresponde à linha 1 do algoritmo 4, onde é calculada a solução de ordem 1 (\mathbf{M}_{11}). Logo cada uma das 4 colunas representa o loop de $j = 1, L - 1$, onde as 4 primeiras linhas representam a informação necessária para obter a solução de ordem $L = 2$ até 5 (solução final, última linha). Pode-se observar que a matriz solução \mathcal{M}_L , vai progressivamente do branco ao preto, representando a progressão que é necessária para obter a solução final.

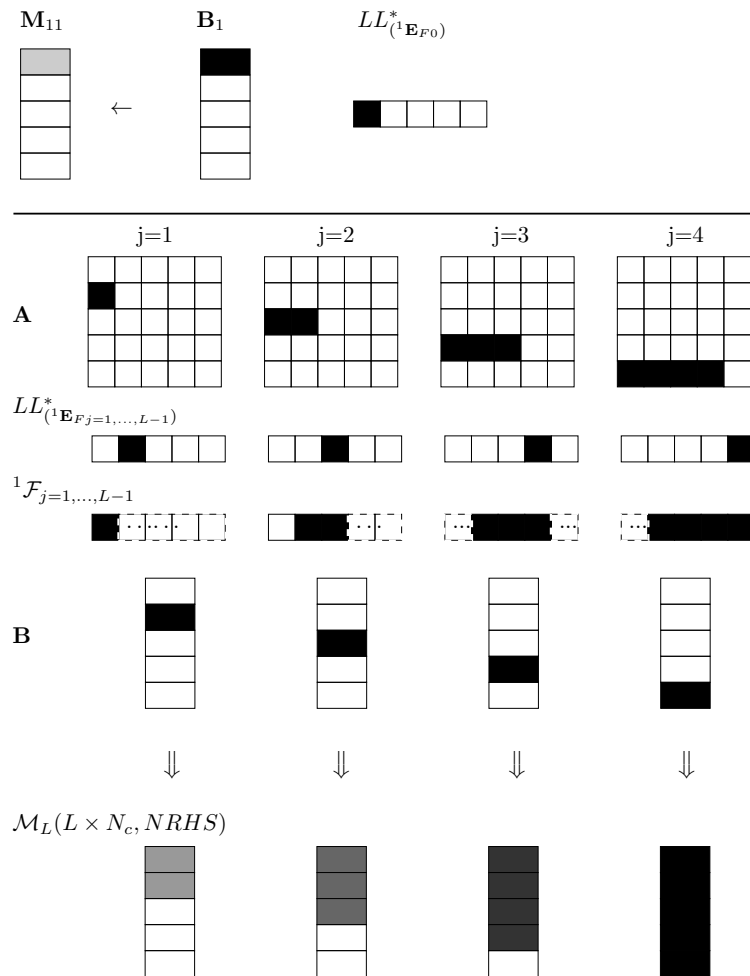


Figura 2.4: esquema de como a informação por blocos da matriz \mathbf{A} , RHS (\mathbf{B}), fatoração de Cholesky da matriz energética do erro ($LL^*_{(\mathbf{E}_{Fj=0,\dots,L-1})}$) e solução reversa (${}^1\mathcal{F}_{j=1,\dots,L-1}$) é incorporado ao algoritmo. Observe que a matriz solução \mathcal{M}_L , vai progressivamente do branco ao preto, representando a progressão que é necessária para obter a solução final.

2.2.5.1 Complexidade

Na tabela 2.1 são apresentados os FLOPs (operações de ponto flutuante) do algoritmo 3, na linha 25, supomos o uso da fatoração de Cholesky. As multiplicações (incluindo divisão) e adições tem contagem separada, e o total é a soma dessas expressões. Sendo operações com números complexos, cada multiplicação contaria como 6 operações (4 multiplicações e 2 adições) e cada adição como 2 operações (2 adições) (Golub e Van Loan, 2013; Blackford e Dongarra, 1999). No Apêndice C são apresentados mais detalhes e a contagem de operações para quando se trabalha com números reais.

linha	uma vez	loop p (loop i)
10: ${}^i\Delta_{Hj-1}$	$(j-1)8N_c^3$	$\frac{1}{6}L(L-1)(L-2)8N_c^3$
13: ${}^i\mathbf{H}_{jj}$	S_m	$\frac{1}{2}(L-1)(L-2)S_m$
14: ${}^i\mathbf{E}_{Hj}$	$4N_c^3 + 4N_c^2$	$\frac{1}{2}(L-1)(L-2)(4N_c^3 + 4N_c^2)$
16: ${}^i\mathbf{F}_{jj}$	S_m	$\frac{1}{2}L(L-1)S_m$
17: ${}^i\mathbf{E}_{Fj}$	$4N_c^3 + 4N_c^2$	$\frac{1}{2}L(L-1)(4N_c^3 + 4N_c^2)$
18: ${}^i\mathcal{F}_j$	$(j-1)8N_c^3$	$\frac{1}{6}L(L-1)(L-2)8N_c^3$
20: ${}^i\mathcal{H}_j$	$(j-1)8N_c^3$	$\frac{1}{6}(L-1)(L-2)(L-3)8N_c^3$
25: $LL^*({}^1\mathbf{E}_{Fj})$	$\frac{1}{3}N_c(N_c+1)(4N_c+5)$	$\frac{1}{3}LN_c(N_c+1)(4N_c+5)$
total		$N_c^3(4L^3 - 12L^2 + \frac{40}{3}L - 4) + N_c^2(4L^2 - 5L + 4) + \frac{5}{3}LN_c + S_m(L^2 - 2L + 1)$

Tabela 2.1: Custo computacional detalhado do algoritmo para sistemas Hermitianos, sub-rotina 1 (Algoritmo 3) tipo Levinson. S_m representa o custo para resolver uma família de N_c sistemas associados à mesma matriz de coeficientes.

Se para o calculo de S_m , usamos a fatoração de Cholesky para um matriz hermitiana, e tendo que para a decomposição os FLOPs são (tendo em conta que para números complexos cada multiplicação contaria como 6 operações e cada adição como 2 operações) $\frac{1}{3}N_c(N_c+1)(4N_c+5)$ e a solução de 2 sistemas triangulares é $8N_c^2NRHS + 4N_cNRHS$ onde $NRHS = N_c$ (Golub e Van Loan, 2013; Blackford e Dongarra, 1999), temos que:

$$\begin{aligned}
 S_{mc} &= \frac{1}{6}N_c(N_c+1)(4N_c+5) + 8N_c^3 + 4N_c^2 \\
 S_{mc} &= \frac{28}{3}N_c^3 + 7N_c^2 + \frac{5}{3}N_c
 \end{aligned} \tag{2.51}$$

portanto, os FLOPs totais:

$$FLOPs_{slv} = \frac{4}{3}N_c^3(3L^3 - 2L^2 - 4L + 4) + N_c^2(11L^2 - 19L + 11) + \frac{5}{3}N_c(L^2 - L + 1) \tag{2.52}$$

tendo para os casos limite: sem particionamento ($L = 1$), o valor corresponde a fatoração de Cholesky de tamanho n , o seja $\frac{4}{3}n^3 + 3n^2 + \frac{5}{3}n$, ja quando $N_c = 1$ e $L = n$, (onde cada

“bloco” é de tamanho 1), temos que:

$$FLOPs_{slv \rightarrow L=n} = 4n^3 + 10n^2 - 22n + 18 \quad (2.53)$$

Para entender melhor o que acontece com o custo computacional quando L é alterado, podemos separar o custo para a multiplicação matriz-matriz (e soma) e o custo para resolver subsistemas, portanto a Eq. 2.52, é a soma entre os FLOPs do produto entre matrizes:

$$FLOPs_{lv=mm} = 4N_c^3 (L^3 - 3L^2 + 3L - 1) + 4N_c^2 (L^2 - 2L + 1), \quad (2.54)$$

e os FLOPs da solução de subsistemas:

$$FLOPs_{lv=slv} = \frac{4}{3}N_c^3 (7L^2 - 13L + 7) + N_c^2 (7L^2 - 11L + 7) + \frac{5}{3}N_c (L^2 - L + 1). \quad (2.55)$$

Na figura 2.5 são apresentados os FLOPs (normalizados respeito a $4n^3 + 10n^2 - 22n + 18$) para $N = 50000$, onde é possível observar que quando o particionamento é aumentado o custo computacional aumenta para o valor máximo que corresponde ao custo de $L = N$, $N_c = 1$. Também é possível mostrar como à medida que L aumenta, o custo computacional para resolver subsistemas lineares ($FLOPs_{lv=slv}$) diminui e o custo das operações entre matrizes ($FLOPs_{lv=mm}$) aumenta (multiplicações entre matrizes e soma de matrizes). Para diferentes tamanhos de matriz, esta distribuição de custo computacional é equivalente.

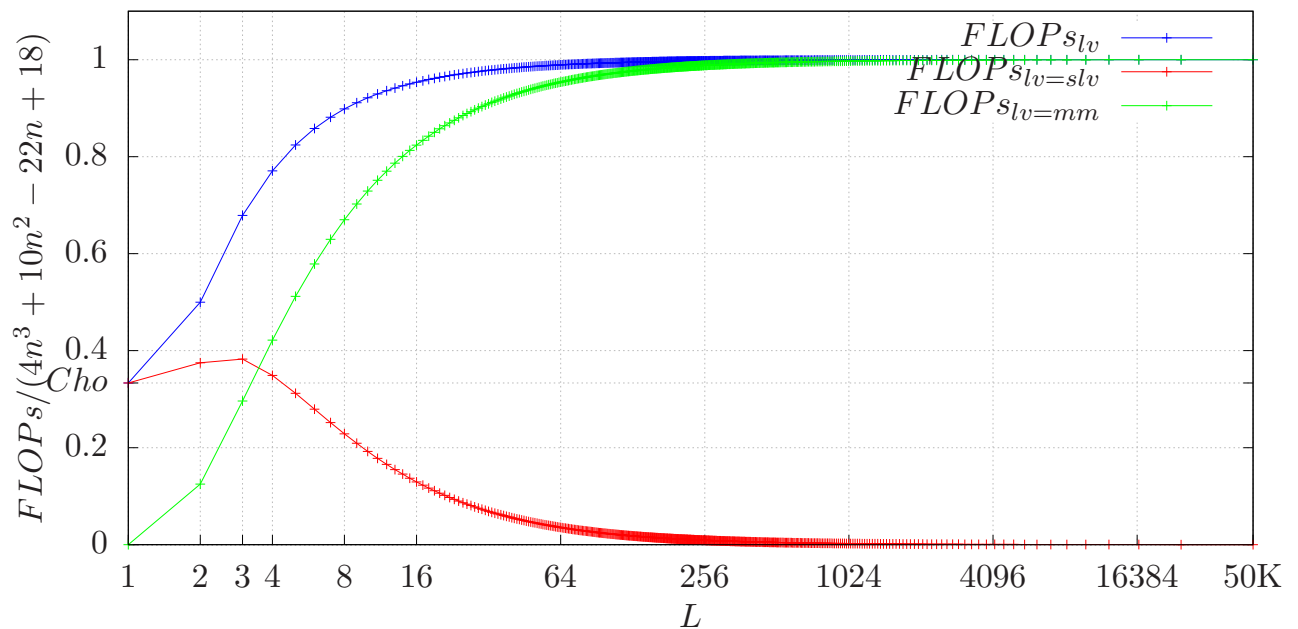


Figura 2.5: Operações de ponto flutuante (FLOP) normalizado para decompor (Algoritmo 3) um sistema $N \times N$ ($N = 50000$) em função do particionamento L .

Para a obtenção da solução (algoritmo 4), os FLOPs são apresentados na tabela 2.2.

linha	uma vez	loop j
1: \mathbf{M}_{11}	$8N_c^2 + 4N_c$	$8N_c^2 + 4N_c$
3: Δ_{Mj}	$j8N_c^2$	$L(L-1)4N_c^2$
4: $\mathbf{M}_{j+1,j+1}$	$8N_c^2$	$(L-1)(8N_c^2 + 4N_c)$
5: \mathcal{M}_{j+1}	$j8N_c^2$	$L(L-1)4N_c^2$
total		$8L^2N_c^2 + 4LN_c$

Tabela 2.2: Custo computacional detalhado do algoritmo, sub-rotina 2 (Algoritmo 4) tipo de Levinson para solução de sistemas Hermitianos para apenas um RHS, para multiplos RHS, basta multiplicar por NRHS.

Lembrando que o tamanho de sistema pode ser entre $(L-1)N_c < n \leq LN_c$; para simplificar suponha que $n = LN_c$, nesse caso o total de operações de ponto flutuante (FLOP) para resolver (sub-rotina 2) um RHS é $8n^2 + 4n$, isso é igual que a solução dos dois sistemas triangulares (substituição direta e reversa para números complexos) na decomposição de Cholesky (Golub e Van Loan, 2013; Blackford e Dongarra, 1999). Em compensação, as operações de multiplicação matriz-matriz (soma das linhas 3 e 5):

$$FLOPs - 2_{slv=mm} = 8L^2N_c^2 - 8LN_c^2, \quad (2.56)$$

são maiores que a substituição direta e reversa (soma das linhas 1 e 4):

$$FLOPs - 2_{slv=slv} = 8LN_c^2 + 4LN_c, \quad (2.57)$$

estas últimas diminuindo, à medida que L aumenta, como se apresenta na figura 2.6. As operações de multiplicação matriz-matriz são mais eficientes e permitem maior paralelismo do que a substituição direta e reversa o que permite melhor desempenho para múltiplos RHS.

O comportamento do custo computacional ao variar L é mantido, conforme o esperado, independentemente de trabalharmos com números reais ou complexos, como pode ser verificado comparando as figuras C.1 e C.2 com as figuras 2.5 e 2.6.

2.2.6 Escalabilidade

A versão paralela do algoritmo proposto por Porsani et al., 2010, e apresentada no pseudo-código 2, tem a vantagem de poder ser aplicada para problemas extremamente grandes, os quais não podem ser resolvidos com a capacidade computacional instalada em um local específico e portanto pode fazer uso de sistemas de computação distribuído pela *web*, como por exemplo computação em nuvem (Coulouris et al., 2013), ou múltiplos centros

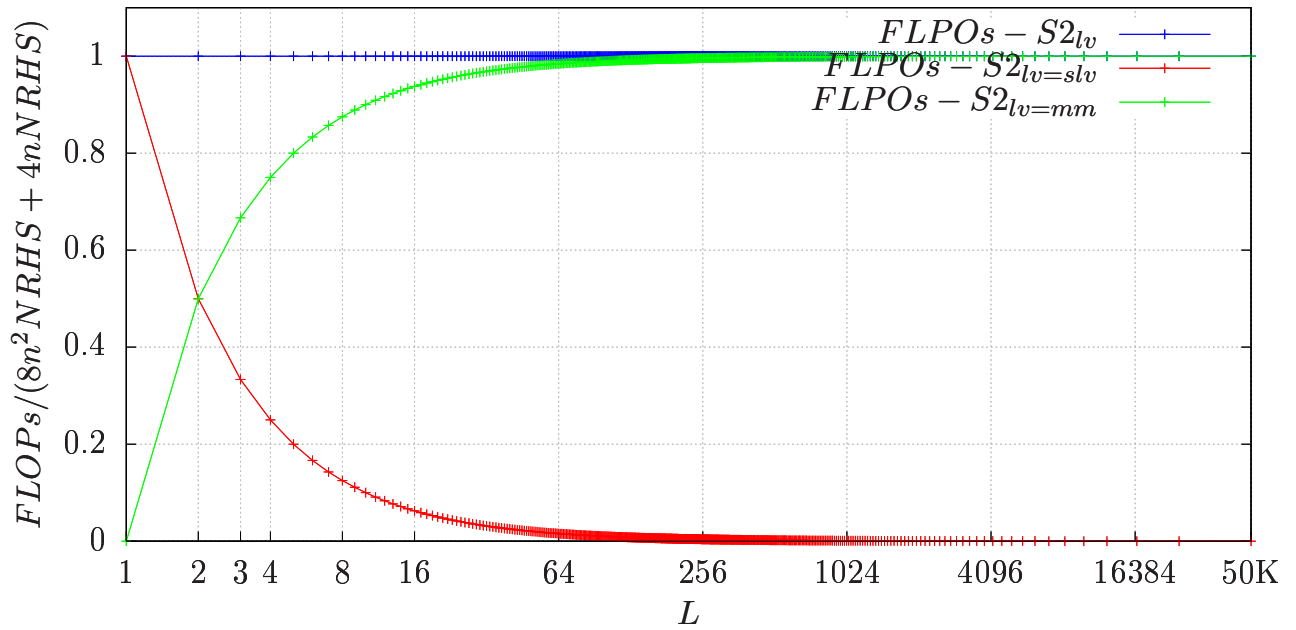


Figura 2.6: Operações de ponto flutuante (FLOP) normalizado para resolver (Algoritmo 4) um sistema de N ($N = 50000$) parâmetros e $NRHS$ lados direitos em função do particionamento L .

de HPC (*High Performance Computing*), mantendo a privacidade do resultado já que só o nó mestre (local) calcula a resposta final. Na contra mão, quando o problema pode ser resolvido com a capacidade de computação instalada, na medida que aumenta a ordem da solução vão ficando livres nós computacionais, o que reduz a eficiência.

Neste trabalho focaremos no caso que o problema possa ser resolvido localmente e apresentamos uma nova implementação para explorar o máximo de capacidade de computação instalada localmente, com bom resultados quando se dispõe de unidades de processamento gráfico (GPU).

Na solução do problema de grande porte, um fator importante é o custo computacional, já que qualquer método usado pode levar dias em tempo de execução. Portanto, a implementação em paralelo oferece uma solução para a redução do tempo de computação.

Existem várias alternativas para paralelizar um problema, usando programação de alto desempenho, que apresenta um crescimento intenso nos últimos tempos, visto que dispositivos como as CPU (*Central Processing Unit*) e os GPU (*Graphics Processing Unit*) evoluíram muito na sua capacidade computacional. A CPU, por exemplo, em contrapartida as limitações da velocidade do *clock* teve a introdução de mais núcleos dentro de um único processador. Já a GPU passou de um simples processador gráfico a um coprocessador paralelo, capaz de executar milhares de operações simultaneamente, gerando grande capacidade

computacional, que muitas vezes supera o poder de processamento das CPUs tradicionais.

Estas arquiteturas, o aparecimento de algumas APIs (*Application Programming Interface*) com suporte ao paralelismo permitiram aproveitar a real capacidade computacional destes dispositivos. Os destaques incluem OpenMP (*Open Multi Processing*) que explora o paralelismo intranodal onde vários processos compartilham a mesma RAM e MPI (*Message-Passing Interface*) que se refere ao paralelismo internodal, que é usado para paralelizar tarefas para a CPU e CUDA (*Compute Unified Device Architecture*) e OpenCL (*Open Computing Language*) para execução paralela na GPU.

Com o intuito de aumentar a capacidade de processamento paralelo aliado a um custo acessível, tem-se nos últimos anos um crescimento do desenvolvimento de algoritmos paralelos de forma híbrida utilizando CPU e GPU. A programação paralela híbrida tem por objetivo unir dispositivos de computação com diferentes arquiteturas que trabalhando em conjunto, permitem atingir um maior desempenho. Além disso, a programação híbrida busca fazer com que cada conjunto de instruções possa ser executado na arquitetura que melhor se adapte. Dessa forma, no presente trabalho vão ser usadas APIs implementadas para CPU utilizando MPI ou openMP e APIs para uso de GPUs.

2.2.6.1 Usando Memória compartilhada e distribuída

A ideia principal desta versão do algoritmo é usar como kernel (para operação dos blocos) uma biblioteca escalonável otimizada para computadores concorrentes com memória compartilhada (Usando OpenBLAS ou a biblioteca MKL) ou distribuída, usamos OpenBLAS (ou a biblioteca MKL) e ScaLAPACK (“Scalable LAPACK”) respectivamente. O algoritmo é o mesmo mostrado em 3. A escalabilidade é dada pelo uso destas bibliotecas para cada uma das linhas do código.

OpenBLAS é uma continuação do GotoBLAS de código aberto (mais detalhes em 1.3.3.2). Ele adiciona implementações otimizadas de kernels de álgebra linear para várias arquiteturas de processador de memória compartilhada, incluindo Intel *Sandy Bridge* e *Loongson*. A biblioteca MKL (mais detalhes em 1.3.3.7) é uma biblioteca desenvolvida na Intel que contém rotinas para computação científica incluindo BLAS, LAPACK, solucionadores de sistemas esparsos e funcionalidades de transformada de Fourier, altamente otimizadas para processadores Intel.

ScaLAPACK usa, para algoritmos densos, um layout de dados cíclicos em bloco (mais detalhes em 1.3.3.6). Escolhemos esta biblioteca principalmente por causa de sua escalabilidade (Dongarra, 1994).

2.2.6.2 Implementação Out-of-Core, usando GPU

A ideia principal deste algoritmo é de permitir utilizar ao máximo as capacidades computacionais da GPU, e ser capaz de rodar, mesmo que todos os dados do sistema completo não possam caber na memória disponível. Para nossa implementação, a fim de evitar comunicações desnecessárias, minimizar as etapas de subida *-Set-* de RAM para GPU) e descida *-Get-* de GPU para RAM, foi necessário alocar na GPU três blocos de $N_c \times N_c$ para $L = 2$ ou seis blocos de $N_c \times N_c$ para $L \geq 3$, para melhor desempenho, e carregar a quantidade máxima de dados na memória da GPU. Esta implementação é útil para grandes sistemas lineares com $L \geq 3$, por exemplo, com uma GPU de 12 GB de memória, é possível alocar 6 matrizes reais de aproximadamente 15.000×15.000 em precisão dupla, nossa implementação é útil para sistema linear $N > 45K$, nesta GPU o tamanho máximo de uma matriz única real, em precisão dupla, é de aproximadamente $N = 38K$.

A codificação é feita em FORTRAN, e para as operações pequenas (bloco de $N_c \times N_c$) feitas por na GPU, utilizamos as bibliotecas otimizadas MAGMA (*Matrix Algebra on GPU and Multicore Architectures*, MAGMA é uma coleção de bibliotecas de álgebra linear para sistemas híbridos de multicore e GPU. Esta biblioteca representa uma metodologia de hibridização onde algoritmos de interesse são divididos em tarefas de granularidade variável e sua execução é agendada sobre os componentes de hardware disponíveis. Usamos, em especial, as sub-rotinas `gpu magmaf_dgemm`, `magmaf_dpotrs_gpu` e `magmaf_dposv_gpu`. No algoritmo 5 está apresentado o pseudocódigo que calcula a solução reversa para as ordens $j = 1, \dots, L$ e a matriz de energia dos erros, e no algoritmo 6 está representado o pseudocódigo que calcula a solução recursivamente para múltiplos RHS.

Algoritmo 5 Algoritmo de GPU Out-of-Core para solução do sistema de bloco-hermitianos de NE, sub-rotina 1

Require: $\mathbf{A}(L \times N_c, L \times N_c)$: sistema de bloco-hermitiano de NE;

$\mathbf{B}(L \times N_c, NRHS)$: $\mathbf{G}^* \mathbf{D}$;

Require: Memória de trabalho no CPU: $LLAjj(N_c, N_c)$, $HH(N_c, N_c, 2L - 3)$, $EEHH(N_c, N_c, (L - 1))$

Require: Memória de trabalho GPU: 6 matrizes (N_c, N_c) : dT1, dT2, dA1, dA2, dB, dC \triangleright No GPU

```

1: Inicializar
2:  ${}^1\mathbf{E}_{F0} \leftarrow \mathbf{A}_{11}$ 
3: for  $i = 1, L$  do
4:      ${}^i\mathbf{E}_{H0} = \mathbf{A}_{ii}$ 
5: end for
6:  $LL^*({}^1\mathbf{E}_{F0}) \leftarrow LL^*({}^1\mathbf{E}_{F0}) \left\{ \begin{array}{l} \text{Set } {}^1\mathbf{E}_{F0} \text{ in dT1} \\ LL^*_{(dT1)} \leftarrow LL^*(dT1) \quad \text{GPU} \\ \text{Get dT1 no } LL^*({}^1\mathbf{E}_{F0}) \text{ e copiar no } LLAjj \end{array} \right.$ 
7: for  $p = 1, L - 1$  do
8:      ${}^{p+1}\mathbf{E}_{F0} = \mathbf{A}_{p+1,p+1} \left\{ \begin{array}{l} \text{Set no dA1} \end{array} \right.$ 
9:     Set  ${}^{p+1}\mathbf{E}_{F0}$  no dA1  $\triangleright$  No GPU
10:    for  $i = p, 1, -1$  do
11:         $j = p - i + 1$ 
12:         ${}^i\Delta_{Hj-1} = \mathbf{A}_{i+j,i} + \sum_{k=1}^{j-1} \mathbf{A}_{i+j,i+k} {}^i\mathbf{H}_{j-1,k} \left\{ \begin{array}{l} \text{Set } \mathbf{A}_{i+j,i} \text{ no dB} \Rightarrow {}^i\Delta_{Hj-1} \\ \text{if } j > 1 \text{ then} \\ \quad \text{for } k = 1, j - 1 \text{ do} \\ \quad \quad \text{Set } \mathbf{A}_{i+j,i+k} \text{ no dT1 e Set } {}^i\mathbf{H}_{j-1,k} \text{ no dT2} \\ \quad \quad \quad dB = dB + dT1 * dT2 \\ \quad \quad \text{end for} \\ \quad \text{end if} \end{array} \right.$ 
13:         ${}^{i+1}\Delta_{Fj-1} = {}^i\Delta_{Hj-1}^* \left\{ \begin{array}{l} dC = dB^* \end{array} \right. \quad \triangleright \text{No GPU}$ 
14:         ${}^i\mathbf{F}_{jj} \leftarrow {}^i\mathbf{E}_{Hj-1} {}^i\mathbf{F}_{jj} = -{}^i\Delta_{Hj-1}^* \left\{ \begin{array}{l} \text{Set } {}^i\mathbf{E}_{Hj-1} \text{ no dT2} \\ \text{if } j = 1 \text{ then} \\ \quad \text{Set } LLAjj \text{ in dT1} \\ \quad dC \leftarrow dT1 {}^i\mathbf{F}_{11} = -dC \quad \text{Xpotrs} \\ \text{else} \\ \quad dT1 = dT2 \\ \quad dC \leftarrow dT1 {}^i\mathbf{F}_{jj} = -dC \quad \text{Xposv} \\ \text{end if} \\ \text{Get } dC \text{ in } {}^i\mathbf{F}_{jj} \end{array} \right.$ 
15:         ${}^i\mathbf{E}_{Fj} = {}^{i+1}\mathbf{E}_{Fj-1} + {}^i\Delta_{Hj-1} {}^i\mathbf{F}_{jj} \left\{ \begin{array}{l} dT1 = dA1 \\ {}^i\mathbf{E}_{Fj} == dA1 = dA1 + dB * dC \end{array} \right.$ 

```

```

16:     if  $p < L - 1$  then
17:          ${}^i\mathbf{H}_{jj} \leftarrow {}^{i+1}\mathbf{E}_{F_{j-1}} {}^i\mathbf{H}_{jj} = -{}^i\Delta_{H_{j-1}}$ 
        {
             $dA2 = -dB$ 
            if  $j = 1$  then
                Set  ${}^{i+1}\mathbf{E}_{F_{j-1}}$  no  $dT1$ 
                 $dA2 \leftarrow dT1 {}^i\mathbf{H}_{11} = dA2$  Xposv
                Get  $dT1$  no  $LLAj_j$ 
            else
                 $dA2 \leftarrow dT1 {}^i\mathbf{H}_{jj} = dA2$  Xposv
            end if
            Get  $dA2$  no  ${}^i\mathbf{H}_{jj}$ 
        }
18:          ${}^i\mathbf{E}_{H_j} = {}^i\mathbf{E}_{H_{j-1}} + {}^i\Delta_{H_{j-1}}^* {}^i\mathbf{H}_{jj}$  {
             $dT2 = dT2 + dB * dA2$ 
            Get  $dT2$  no  ${}^i\mathbf{E}_{H_j}$ 
        }
19:     end if
20:      $\{ {}^i\mathcal{H}_j, {}^i\mathcal{F}_j \} \leftarrow \begin{bmatrix} \mathbf{I} & {}^i\mathcal{F}_j \\ {}^i\mathcal{H}_j & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \oplus \\ {}^i\mathcal{H}_{j-1} & {}^{i+1}\mathcal{F}_{j-1} \\ \oplus & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {}^i\mathbf{F}_{jj} \\ {}^i\mathbf{H}_{jj} & \mathbf{I} \end{bmatrix}$ 
        {
            for  $k = 1, j - 1$  do
                Set  ${}^i\mathbf{H}_{j-1k}$  no  $dT1$ 
                Set  ${}^{i+1}\mathbf{F}_{j-1k}$  no  $dT2$ 
                if  $k < L - 1$  e  $j > 1$  then
                     $dB = dT2$ 
                end if
                 $dT2 = dT2 + dT1 * dC$ 
                Get  $dT2$  no  ${}^i\mathbf{F}_{jk}$ 
                if  $k < L - 1$  e  $j > 1$  then
                     $dT1 = dT1 + dB * dA2$ 
                    Get  $dT1$  no  ${}^i\mathbf{H}_{jk}$ 
                end if
            end for
        }
21:     end for
22:      $LL_{({}^1\mathbf{E}_{F_j})}^* \leftarrow LL_{({}^1\mathbf{E}_{F_j})}^* \left\{ \begin{array}{l} LL_{dA1}^* \leftarrow LL^*(dA1) \\ \text{Get } dA1 \text{ no } LL_{({}^1\mathbf{E}_{F_j})}^* \end{array} \right.$ 
23: end for
24: return  ${}^1\mathcal{F}_{j=1, \dots, L-1}; LL_{({}^1\mathbf{E}_{F_j=0, \dots, L-1})}^*$ 

```

Algoritmo 6 Algoritmo Out-of-Core em GPU para solução de sistema bloco-hermitiano de NE, sub-rotina 2, solução de múltiplos RHS

Require: $\mathbf{A}(L \times N_c, L \times N_c)$: sistema de bloco-hermitiano de NE;

${}^1\mathcal{F}_{j=1,\dots,L-1}$;

$LL^*({}^1\mathbf{E}_{F_j=0,\dots,L-1})$;

$\mathbf{B}(L \times N_c, NRHS)$: $\mathbf{G}^*\mathbf{D}$;

Require: Memória de trabalho no GPU: 3 matriz $(N_c, maxRHS)$: dA, dB, dC \triangleright In GPU

1: Cal maxRHS(Nc, Memória da GPU,bytes)

2: $\mathbf{M}_{11} \leftarrow LL^*({}^1\mathbf{E}_{F_0})\mathbf{M}_{11} = \mathbf{B}_1$ $\left\{ \begin{array}{l} \text{Set } LL^*({}^1\mathbf{E}_{F_0}) \text{ in } dA \\ \text{for } jr = 1, nrhs, maxNRHS \text{ do} \\ \quad \text{Set } \mathbf{B}_{1,jr} \text{ in } dC \\ \quad dC \leftarrow dA\mathbf{M}_{11} = dC \\ \quad \text{Get } dC \text{ in } \mathbf{M}_{1,jr} \\ \text{end for} \end{array} \right.$

3: **for** $j = 1, L - 1$ **do**

4: **for** $jr = 1, nrhs, maxNRHS$ **do**

5: $\Delta_{Mj} = [-\mathbf{B}_{j+1} \quad \mathbf{A}_{j+1,1} \quad \cdots \quad \mathbf{A}_{j+1,j}] \begin{bmatrix} \mathbf{I}_{nr} \\ \mathcal{M}_j \end{bmatrix}$ $\left\{ \begin{array}{l} \text{Set } -\mathbf{B}_{j+1,jr} \text{ in } dB \\ \text{for } k = 1, j \text{ do} \\ \quad \text{Set } \mathbf{A}_{j+1,k} \text{ in } dA \\ \quad \text{Set } \mathbf{M}_{jk} \text{ in } dC \\ \quad dB = dB + dA * dC \\ \text{end for} \end{array} \right.$

6: $\mathbf{M}_{j+1,j+1} \leftarrow LL^*({}^1\mathbf{E}_{F_j})\mathbf{M}_{j+1,j+1} = \Delta_{Mj}$

$\left\{ \begin{array}{l} \text{Set } LL^*({}^1\mathbf{E}_{F_j}) \text{ in } dA \\ dB \leftarrow dA\mathbf{M}_{j+1,j+1} = dB \quad \text{Xpotrs} \\ \text{Get } dB \text{ in } \mathbf{M}_{j+1,j+1} \end{array} \right.$

7: $\begin{bmatrix} \mathbf{I}_{nr} \\ \mathcal{M}_{j+1} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{nr} & \emptyset^T \\ \mathcal{M}_j & {}^1\mathcal{F}_j \\ \emptyset & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{M}_{j+1,j+1} \end{bmatrix}$ $\left\{ \begin{array}{l} \text{for } k = 1, j \text{ do} \\ \quad \text{Set } {}^1\mathbf{F}_{jk} \text{ in } dA \\ \quad \text{Set } \mathbf{M}_{jk} \text{ in } dC \\ \quad dC = dC + dA * dB \\ \quad \text{Get } dC \text{ in } \mathbf{M}_{j+1,k} \\ \text{end for} \end{array} \right.$

8: **end for**

9: **end for**

10: **return** $\mathcal{M}_L(L \times N_c, NRHS)$

3

Solução De Sistemas Lineares Hermitianos, Positivos Definidos (PD) E Densos De Grande Porte Com Múltiplos Lados Direitos: Resultados Numéricos

3.1 Introdução

Neste capítulo, vamos verificar a viabilidade do algoritmo proposto neste trabalho, bem como sua comparação com a fatoração Cholesky, usando bibliotecas amplamente usadas (LAPACK, SCALAPACK e MAGMA) para obter a solução do sistema completo. Analisando os resultados vamos concluir que quando temos um número de RHS grande (maior a ordem N do SL) o algoritmo proposto se mostrou eficiente na resolução de sistemas lineares.

Na seção 3.2 são referidas as formas em que foram gerados sinteticamente os sistemas lineares e como foi validada a solução.

Na seção 3.3 são apresentadas os resultados da implementação em um *clusters* da versão original proposta por (Porsani et al., 2010), posteriormente, nas seções 3.4, 3.5 e 3.6 são mostrados os resultados do código proposto neste trabalho nas diferentes implementações para diferentes arquiteturas (SMP, *cluster* e GPU).

3.2 Construção sintética do problema e validação do resultado

3.2.1 Simulação do problema

Como já foi mencionado, a implementação proposta tem como objetivo resolver o sistema linear $\mathbf{A}\mathbf{x} = \mathbf{B}$, onde \mathbf{A} é uma matriz positiva definida, então, a simulação do problema foi realizada de duas formas:

- $\mathbf{A} = \mathbf{G}^T * \mathbf{G}$, onde \mathbf{G} é uma matriz com valores aleatórios (uniformemente distribuídos) entre -10 e 10 . A matriz \mathbf{B} é gerada diretamente com números aleatórios (uniformemente distribuídos) entre 0 e 1
- A matriz \mathbf{A} é gerada diretamente usando a função `_lagsy` (disponível nas ferramentas de *testing* do LAPACK ¹) que gera, no caso real, uma matriz simétrica real \mathbf{A} , por pré e pós-multiplicação de uma matriz diagonal real \mathbf{D} com uma matriz ortogonal aleatória: $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^T$. A matriz diagonal \mathbf{D} é composto de números aleatórios entre 0 e 1000 .

Para uma apropriada comparação entre o algoritmo proposto e a referência, a semente geradora dos números aleatórios é a mesma. Em todos os casos foi usada dupla precisão.

3.2.2 Estabilidade do algoritmo

A análise do erro e da estabilidade do algoritmo de resolução de equações lineares levou em consideração os seguintes aspectos: seja $\mathbf{A}\mathbf{x} = \mathbf{b}$ o sistema a ser resolvido. Seja $\hat{\mathbf{x}}$ a solução calculada pelo algoritmo. Seja *mathbfr* o vetor resíduo $\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$. Na ausência de erro de arredondamento, \mathbf{r} seria zero e $\hat{\mathbf{x}}$ seria igual a \mathbf{x} ; com erro de arredondamento isso não vai acontecer, o que faz necessário avaliar a estabilidade do algoritmo. Neste trabalho avaliamos a estabilidade *backward* (Björck (1996) Capítulo 2, definição 2.1.1 e teorema 2.1.1), onde a solução é estável se e somente se a solução calculada tem um pequeno resíduo.

$$\|\mathbf{B} - \mathbf{A}\hat{\mathbf{x}}\| \leq \epsilon c \|\mathbf{A}\| \|\hat{\mathbf{x}}\| \quad (3.1)$$

onde c é uma constante não muito grande e ϵ faz referência é precisão relativa da máquina (o menor número de ponto flutuante).

¹Esta função permite definir um possível bandejamento k , mais neste trabalho foi usado Uma matriz completamente densa $k = N - 1$. Para o caso real de dupla precisão a função tomo o nome de `Dlagsy`. Maiores informações em: http://www.netlib.org/lapack/explore-html/d1/dc0/group__double_matgen__ga5f743c86cc2e595aef8e6f8662562026.html

Neste trabalho consideramos um resíduo relativo que é calculado como se mostra na eq 3.2 para um sistema de múltiplos lados direitos:

$$r = \frac{\|\mathbf{B} - \mathbf{A}\hat{\mathbf{X}}\|_{\infty}}{\epsilon N \|\mathbf{A}\|_{\infty} \|\hat{\mathbf{X}}\|_{\infty}} \quad (3.2)$$

onde ϵ refere-se à precisão relativa da máquina (o menor número de ponto flutuante) e N é o número de incógnitas do SL. Neste trabalho se considera que a la solução é correta se o resíduo relativo for menor que 3 ($r \leq 3$).

O erro de predição relativo da solução e calculado como se mostra na eq 3.3

$$E = \frac{\|\mathbf{B} - \mathbf{A}\hat{\mathbf{X}}\|_{\infty}}{N \left(\|\mathbf{A}\|_{\infty} \|\hat{\mathbf{X}}\|_{\infty} + \|\mathbf{B}\|_{\infty} \right)} \quad (3.3)$$

Este erro de predição relativo tem que ser da ordem de magnitude do ϵ , e é considera como solução valida se $E \leq 10\epsilon$.

3.3 Resultados da Versão original

A seguir são mostrados resultados preliminares da implementação MPI do algoritmo 2 (Porsani et al., 2010), cabe salientar que para obter as EN ($\mathbf{A} = \mathbf{G}^* \mathbf{G}$ e $\mathbf{b} = \mathbf{G}^* \mathbf{d}$) foi implementada uma estratégia MPI-openMP, usando esta API é possível aproveitar ao máximo a memória RAM do equipo sem precisar operações I/O no disco já que todos os processos de uma só máquina (openMP) trabalham em uma única cópia de \mathbf{G} . Ao final, a matriz $\mathbf{A} = \mathbf{G}^* \mathbf{G}$ e o vetor $\mathbf{b} = \mathbf{G}^* \mathbf{d}$ só estão disponíveis no nó 0.

A solução do sistema $\mathbf{A}\mathbf{m} = \mathbf{b}$ foi implementada com MPI. Os resultados correspondem ao caso que $\mathbf{G}\mathbf{m} = \mathbf{d}$ são reais. Já que só o processo 0 tem $\mathbf{A} = \mathbf{G}^T \mathbf{G}$ ele faz envio dos blocos de informação que cada processo precisa em cada ordem da solução. Isto permite fazer um melhor uso da memória que possibilita ter a resolução de matrizes mais grandes sem precisar de operações I/O em disco durante todo o processo (cálculo de $\mathbf{G}^T \mathbf{G}$ e o cálculo de \mathbf{m}), mas aumenta o número de comunicações que são necessárias entre o *Máster* (rank=0) e os *Slaves* (rank \geq 1).

O algoritmo foi escrito para resolver qualquer tamanho de matriz e qualquer número de processadores.

Nos passos $\{\mathbf{m}_{j+1,j+1} \leftarrow {}^1\mathbf{E}_{F_j} \mathbf{m}_{j+1,j+1} = \Delta_{m_j}\}$, $\{\mathbf{H}_{jj} \leftarrow {}^{i+1}\mathbf{E}_{F_{j-1}} \mathbf{H}_{jj} = {}^i\Delta_{H_{j-1}}\}$ e $\{\mathbf{F}_{jj} \leftarrow {}^i\mathbf{E}_{H_{j-1}} \mathbf{F}_{jj} = {}^i\Delta_{H_{j-1}}^*\}$ foi usado a fatoração Cholesky (usando o algoritmo de Cholesky-Crout).

3.3.1 Resultados com a máquina MARRECA

3.3.1.1 Para diferentes tamanhos de matriz ($G_{m=n}$), tempo vs $L=Np$

Nas figuras 3.1 e 3.2 é apresentado o tempo de execução necessário para encontrar a solução do sistema $\mathbf{A}\mathbf{m} = \mathbf{b}$, onde \mathbf{A} é uma matriz quadrada simétrica $N \times N$, onde pode-se evidenciar a redução de tempo em função do número de processos ($Thread(s)$), quando o número de processos é aumentado para 24 não é observada uma redução significativa do tempo, já que os dois $Thread(s)$ por $Core$ não são concorrentes. Na figura 3.2 é possível observar a dependência linear entre o logaritmo do tempo e o logaritmo do N para tamanhos de matriz maiores a 1000×1000 aproximadamente.

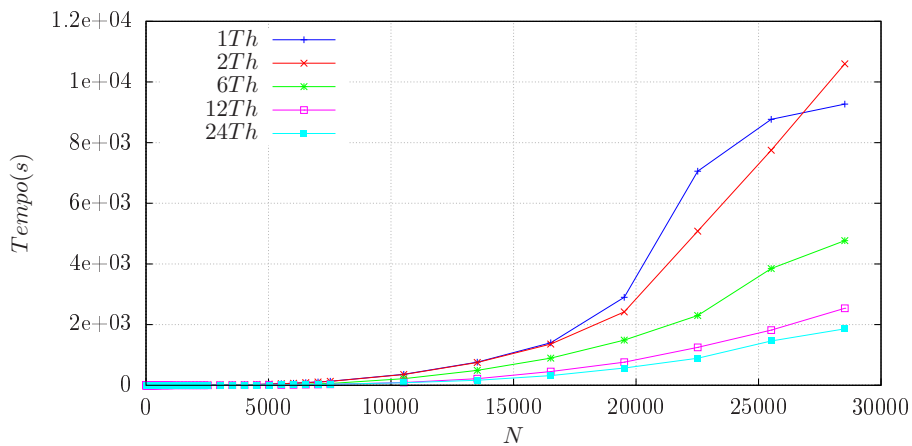


Figura 3.1: Tempo total (segundos) em função do tamanhos de matriz \mathbf{A} de $N \times N$. Th faz referência aos $Thread(s)$. O particionamento L é igual ao número de $Thread(s)$

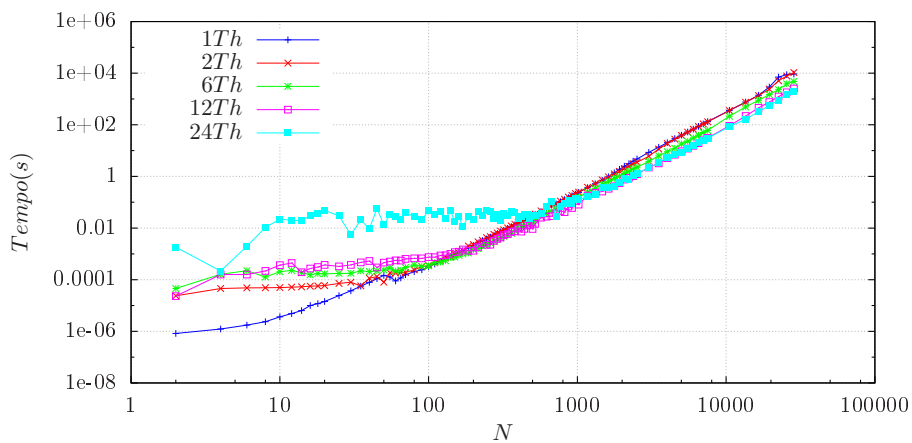


Figura 3.2: Tempo total em função do tamanhos de matriz \mathbf{A} de $N \times N$ em escala logarítmica base 10. Th faz referência aos $Thread(s)$. O particionamento L é igual ao número de $Thread(s)$

3.3.1.2 *SpeedUp* e eficiência

A velocidade aumenta proporcionalmente com o número de processadores, porém, para as matrizes menores, o tempo de comunicação gera atrasos que não permite um aumento proporcional (ver Figura 3.3) e reduzem a eficiência (Figura 3.4). Cabe salientar que o particionamento L deve ser igual ao número de processadores a usar, e como pode ser visto na figura 2.2, cada vez que é incrementada a ordem da solução, um processador queda livre, então na melhor das hipóteses a eficiência máxima possível seria 0.5 sem ter em conta os tempos de comunicação entre processos.

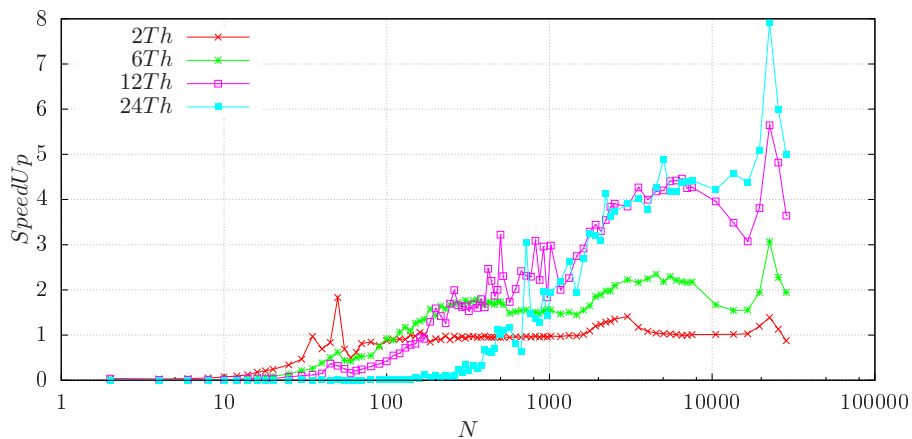


Figura 3.3: Aumento da velocidade total em função do tamanho da matriz, para diferentes níveis de particionamento ($L = th$), $L =$ número de processadores = ordem de particionamento .

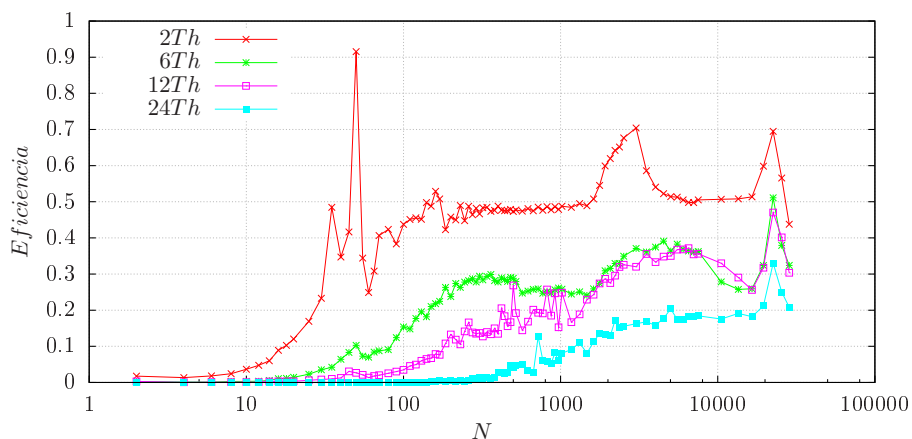


Figura 3.4: Eficiência em função do tamanho da matriz, para diferentes níveis de particionamento ($L=th$), $L =$ número de processadores.

Na figura 3.4 para quando são usados 2Th, temos valores de eficiência muito perto de 0.5, incluso para alguns tamanhos de matriz, a eficiência é maior a 0,5 (máxima possível teoricamente), isto se deve a que cada Thread(s) tem sua própria memória *cache* , então

temos um aumento efetivo da memória cache disponível para o processamento (Chapman et al., 2008). A eficiência poderia ser ainda maior se no fosse por a latência² no acesso a memória já que o sistema tem um acesso no uniforme a memória (NUMA³).

3.3.2 Resultados com a máquina Águia

As características das matrizes usadas são as mesmas que as usadas na seção anterior (reais de precisão dupla)

3.3.2.1 Para diferentes tamanhos de matriz ($G_{m=n}$), tempo vs $L=Np$

Nas figuras 3.5 e 3.6 é apresentado o tempo de execução que é preciso para calcular a solução do sistema $\mathbf{A}\mathbf{m} = \mathbf{b}$, onde \mathbf{A} é uma matriz quadrada simétrica $N \times N$, onde pode-se evidenciar a redução de tempo em função do número de processos (*cpu*). Na figura 3.6 é possível observar a dependência linear entre o logaritmo do tempo e o logaritmo do N para tamanhos de matriz maiores a 2000×2000 aproximadamente.

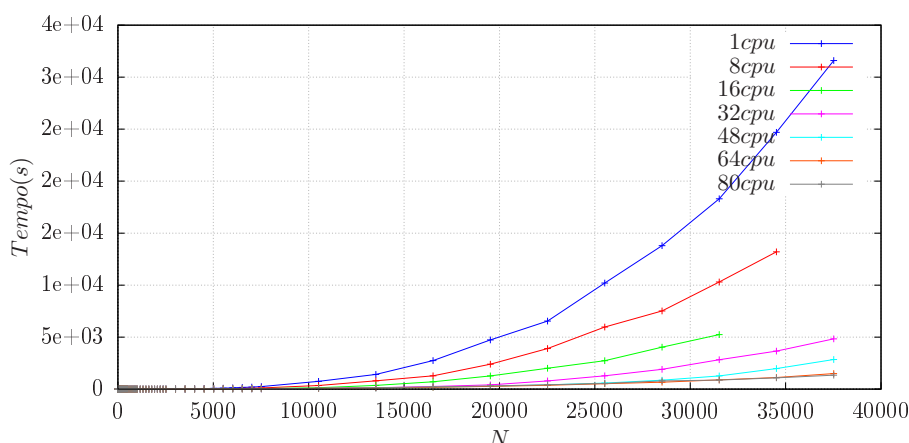


Figura 3.5: Tempo total em função do tamanhos de matriz \mathbf{A} de $N \times N$. O particionamento L é igual ao número de *cpu* usados

²Latência: Tempo gasto esperando por uma resposta. Latência de memória é o tempo que leva para os dados chegarem após o início de uma referência de memória. Um caminho de dados com alta latência de memória seria inadequado para movimentar pequenas quantidades de dados (Chapman et al., 2008).

³NUMA: *non-uniform memory access*: Acesso não uniforme à memória. Normalmente, processadores individuais em pequenas máquinas de memória compartilhada podem acessar qualquer local de memória com a mesma velocidade. Isso não é necessariamente o caso de sistemas maiores e também não é verdadeiro para todas as plataformas pequenas. Uma maneira de permitir que muitas CPUs compartilhem uma grande quantidade de memória é conectar clusters de CPUs com uma rede rápida (por exemplo, um barramento de memória compartilhada) a um determinado pedaço de memória enquanto esses clusters são conectados por uma rede menos custosa. O efeito de tal estrutura é que alguma memória pode estar mais próxima de um ou mais dos processadores e, portanto, acessada mais rapidamente por eles. A diferença no tempo de acesso à memória pode afetar o desempenho de um programa OpenMP (Chapman et al., 2008).

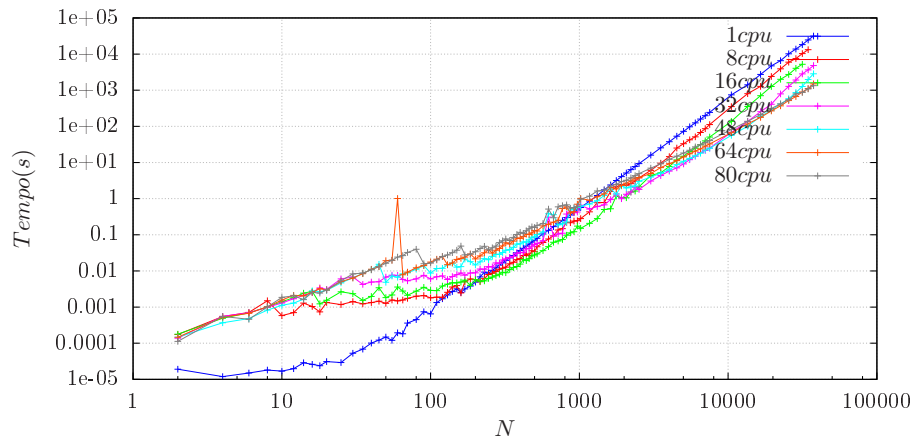


Figura 3.6: Tempo total em função do tamanhos de matriz \mathbf{A} de $N \times N$ em escala logarítmica base 10. O particionamento L é igual ao número de *cpu* usados

3.3.2.2 *SpeedUp* e eficiência

A velocidade aumenta proporcionalmente com o número de processadores, para as matrizes menores, o tempo de comunicação gera atrasos que não permite um aumento proporcional (ver figura 3.7) e reduzem a eficiência (Figura 3.8).

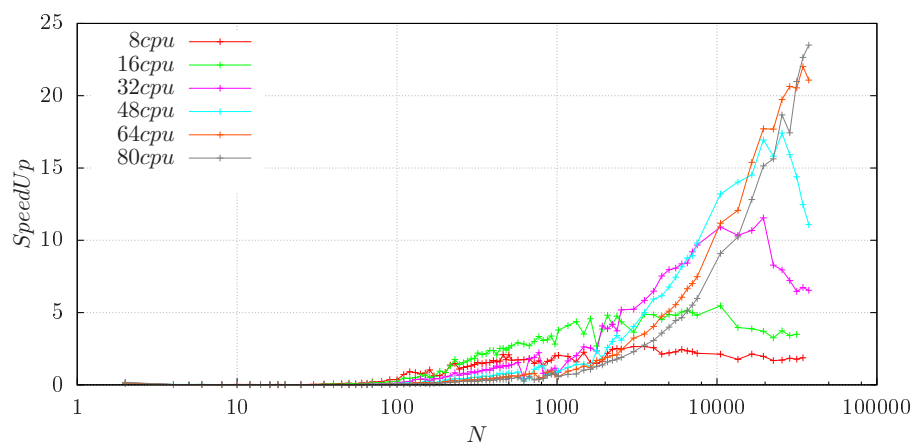


Figura 3.7: Aumento da velocidade total em função do tamanho da matriz, para diferentes tamanhos de particionamento (L). O particionamento L é igual ao número de processadores.

A eficiência da máquina MARRECA foi maior que na máquina AGUIA, isto é devido basicamente a que todos os *Thread(s)* compartilham a totalidade da memória RAM da máquina, sendo então muito mais eficiente a troca de informação entre *Thread(s)*, além de que a MARRECA tem maior memória cache L3. Na AGUIA a eficiência máxima foi perto de 0,3 e na marreca entre 0,3 e 0,7. Conforme mencionado anteriormente na seção 2.2.4, esta baixa eficiência torna o algoritmo pouco atraente, exceto em problemas extremadamente grandes, os quais não podem ser resolvidos com a capacidade computacional instalada em

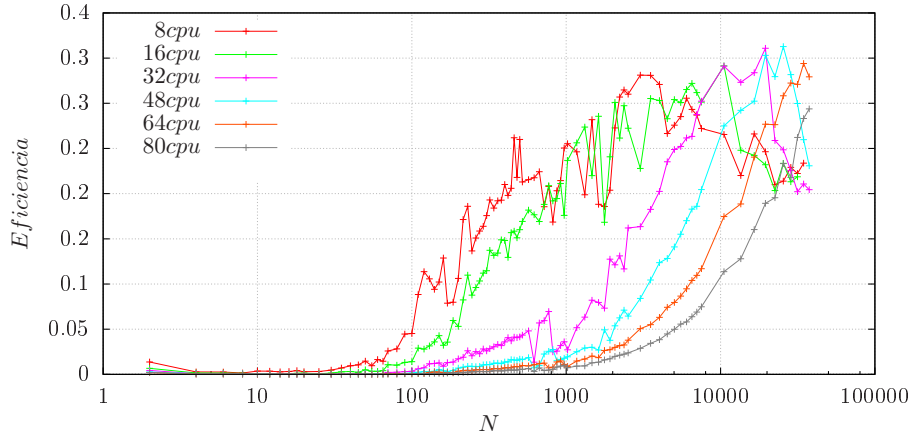


Figura 3.8: Eficiência em função do tamanho da matriz, para diferentes tamanhos de particionamento (L). O particionamento L é igual ao número de processadores.

um local específico e portanto pode fazer uso de sistemas de computação distribuída pela *web* (Seção 2.2.6).

Nas seções seguintes apresentamos a versão proposta com 3 diferentes sistemas de computação paralela: memória compartilhada (seção 3.4), memória distribuída (seção 3.5) e com uso de GPU (seção 3.6), que permitem o aproveitamento total das capacidades de processamento em todo momento, desde o início até a obtenção da solução final.

3.4 Versão memória compartilhada

Em sistemas com memória compartilhada para sistemas com processamento *multithreading* (SMT ⁴), no algoritmo proposto (Algoritmos 3 e 4) cada uma das operações (Multiplicações de matrizes e resolução de sistemas lineares de ordem $N_c \times N_c$) é realizada usando a biblioteca OpenBLAS ou MKL. É o mesmo código que de forma serial (um so CPU), mas compilado com os *flags* correspondentes para que cada subrotina (LAPACK da Intel -MKL-) seja executada de maneira paralela usando todos o *threads* disponíveis no ambiente de execução. Para esta seção usamos um nó do *cluster* OGUN (Especificações descritas na seção 1.3.2.3.1) com a máxima capacidade de *threads*.

Nas provas de rendimento, o tamanho dos problemas (Gigabytes) usados estão condicionados a poder-se acomodar por completo na memória RAM (incluindo matrizes auxiliares), mas se o problema for maior que a memória RAM disponível, é perfeitamente possível adaptar o código implementando operações de I/O com o uso de memórias externas ou discos

⁴SMT: *Simultaneous multithreading*

SSD ⁵ o HDD ⁶.

Na implementação do algoritmo em memória compartilhada, al igual que em memória distribuída, assumindo que todo o problema incluindo matrizes auxiliares cabe na RAM, não temos limitações para definir o nível de particionamento (caso contrário para sistemas com GPU, onde a memória da GPU define o nível de particionamento), portanto se faz necessário mostrar o que acontece com o desempenho para um mesmo problema quando se varia o nível de particionamento (Variando o L e portanto Nc). Na figura 3.9 se apresenta esse efeito para $N = NRHS = 60k$, pode-se apreciar que o tempo Fac_{BH} , que se refere à sub-rotina 3, tende a aumentar com L (como era esperado, ver seção 2.2.5.1). Em quanto o tempo de solução slv_{BH} , que se refere à sub-rotina 4, tem uma leve queda com ou aumento de L . O tempo total Tot_{BH} , de modo geral cresce levemente. O mesmo efeito é percebido para diferentes tamanhos de N e para quando $NRHS$ é maior em relação com N (veja figuras E.1 e E.2 na seção E.1).

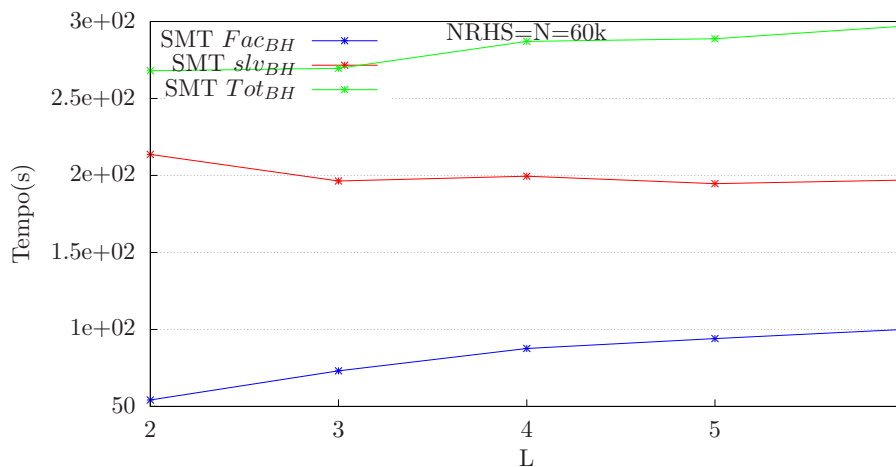


Figura 3.9: Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória compartilhada para um sistema quando $N = NRHS = 60k$ e variando o particionamento ($L = 2, 3, 4, 5, 6$), usando um CPU de 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

Ao comparar o desempenho com ou algoritmo de referência, usaremos um nível de particionamento $L = 2$, e para um valor constante de N , variamos $NRHS$ até valores muito superiores a N . Na figura 3.10 se mostra a comparação do desempenho do nosso algoritmo

⁵SSD: *Solid-State Drive*. Uma unidade de estado sólido é um dispositivo de armazenamento de estado sólido que usa conjuntos de circuitos integrados para armazenar dados de forma persistente

⁶HDD: *Hard Disk Drive*. Uma unidade de disco rígido popularmente chamado também de HD é um dispositivo de armazenamento de dados eletromecânico que armazena e recupera dados digitais usando armazenamento magnético e um ou mais pratos rígidos de rotação rápida revestidos com material magnético

para o caso de $N = 60k$ e com valores de $NRHS$ variando de $12k$ ($NRHS = N/5$) ate $NRHS = 180k$ ($NRHS = 3N$). As duas linhas horizontais, que correspondem aos tempos Fac_{BH} e $dpotrf$ (5 repetições, das quais é calculada a média das 3 intermediárias), este valor é replicado na figura para todos os valores de $NRHS$. Os tempos necessários para obter a solução slv_{BH} e $dpotrs$ para cada valor de $NRHS$ estão representados nas linhas inclinadas (onde também foram realizadas 5 repetições, o pior e o melhor tempos foram eliminados e os 3 restantes foram promediados, isto para cada valor de $NRHS$), bem como o tempo total (soma de $Fac_{BH} + slv_{BH}$ e $dpotrf + dpotrs$).

É possível ver na figura (3.10) que embora o tempo Fac_{BH} seja maior que $dpotrf$ (como é esperado), o comportamento global é semelhante. Diante disso, foram realizados testes utilizando $NRHS$ muito maiores que N (diminuindo N devido a limitações de memória RAM). Esses testes estão representados nas figuras 3.11 e 3.12 ($N = 45k$ e $N = 30k$), onde $NRHS$ máximo vai ate é igual a $9N$ e $21N$ respectivamente.

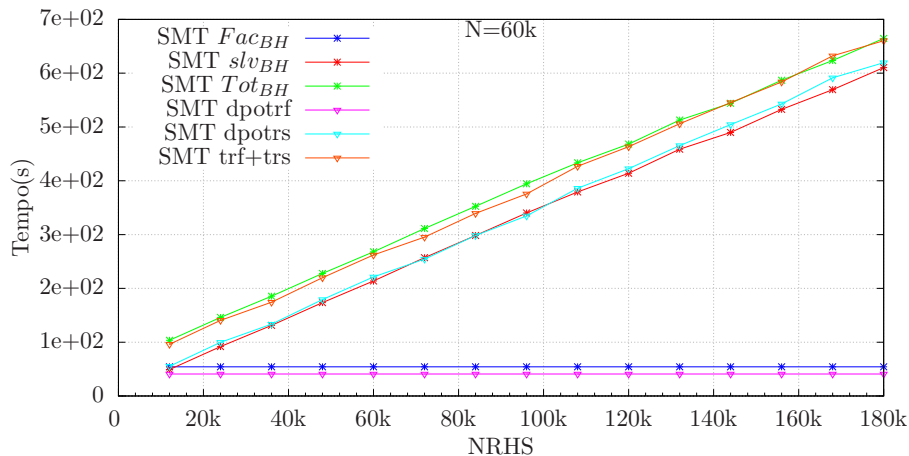


Figura 3.10: Comparação entre o algoritmo de referência para sistema de memória compartilhada: ($dpotrf + dpotrs$) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 60k$ ($L = 2$ e $N_c = 30k$) e dimensões diferentes de RHS ($12k$ a $180k$), usando um CPU de 40 núcleos, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

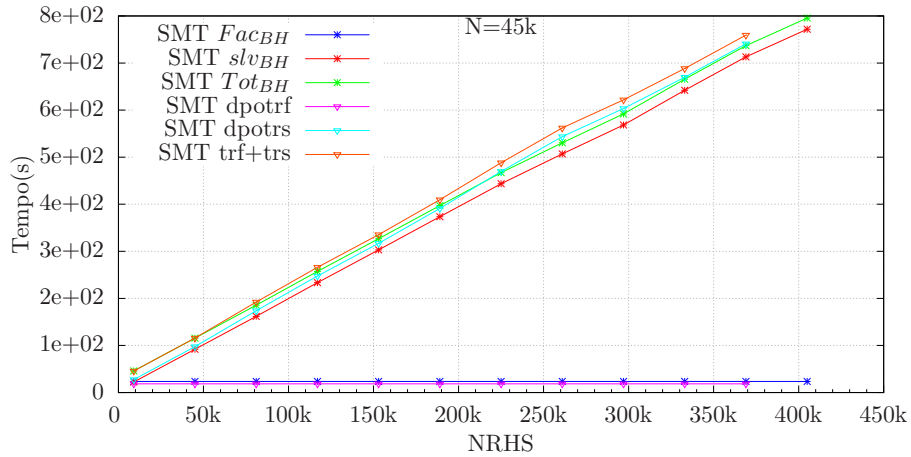


Figura 3.11: Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 2$ e $N_c = 22.5k$) e dimensões diferentes de RHS ($9k$ a $405k$), usando um CPU de 40 núcleos, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

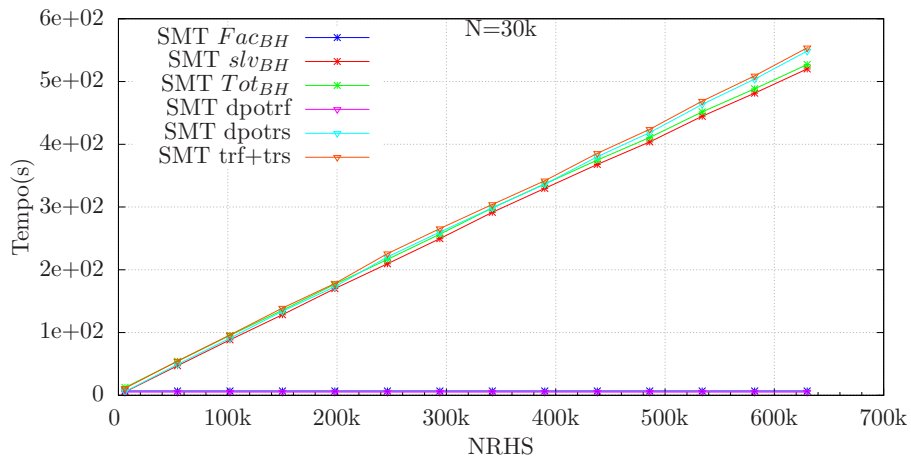


Figura 3.12: Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS ($9k$ a $630k$), usando um CPU de 40 núcleos, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

À medida que aumentamos o NRHS, podemos perceber que o desempenho da segunda parte do algoritmo que incorpora a parte direita da equação, responsável por encontrar a solução, tem um desempenho melhor, o que compensa em maior tempo na primeira parte. A partir das figuras (3.11 e 3.12) pode-se observar uma tendência: quanto maior o NRHS, a diferença de tempo entre os dois algoritmos aumentará. Ressalta-se que essa diferença dependerá tanto das características do equipamento quanto do tamanho do problema.

3.5 Versão memória distribuída

Em sistemas com memória distribuída, no algoritmo proposto (Algoritmos 3 e 4) cada uma das operações (Multiplicações de matrizes e resolução de sistemas lineares de ordem $N_c \times N_c$) e realizada usando a biblioteca SCALAPACK. O código é diferente, pois é necessário levar em consideração as sub-rotinas e particularidades da biblioteca SCALAPACK, como o esquema cíclico de distribuição dos dados ⁷.

A implementação do nosso algoritmo para sistemas de memória distribuída e assumindo que a memória RAM é suficiente para acomodar todo o problema (e matrizes auxiliares), implica que o particionamento (valor de L) pode ser escolhido arbitrariamente, ao contrário de quando é implementado com o uso de GPU onde a memória disponível e o tamanho do problema determinarão o nível de particionamento, como veremos mais adiante. Por conta disso, propomos inicialmente alguns testes de tempo de execução variando o L de 2 a 5. A figura 3.13 mostra esse resultado para $N = NRHS = 30k$, onde pode-se observar que o tempo Fac_{BH} aumenta dependendo do nível de particionamento (ver seção 2.2.5.1). Em quanto a o tempo de solução slv_{BH} , tem uma aparente queda ou estabilização com ou aumento de L . Figuras semelhantes para outros tamanhos de SL são mostrados no Apêndice E.2 (Figura E.9), que apresentam uma tendência semelhante.

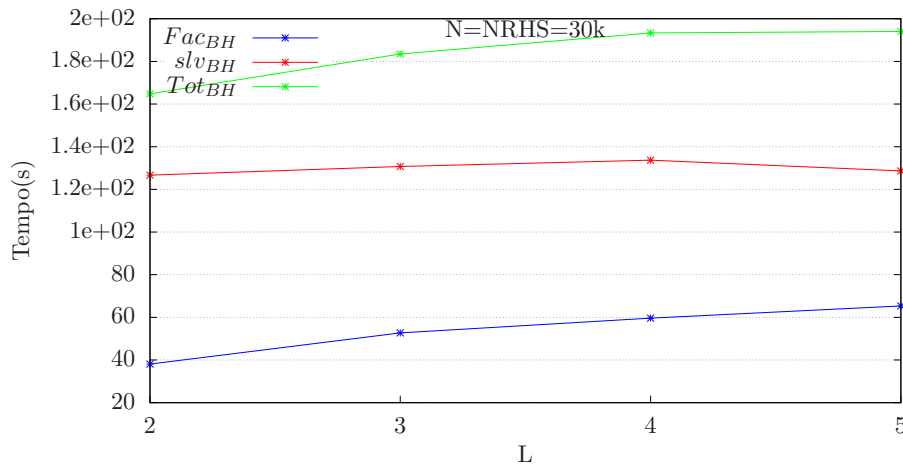


Figura 3.13: Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória distribuída para um sistema quando $N = NRHS = 30k$ e variando o particionamento ($L = 2, 3, 4, 5$), usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

⁷Por simplicidade nos usamos $mb = nb$, onde $nb = mb = 64$. Para a definição da grade, nos usamos uma distribuição que procure que P seja igual a Q em cada caso

Tendo em mente os resultados anteriores, para comparar nossa implementação com o algoritmo de referência, foi escolhido o valor de $L = 2$, por apresentar melhores resultados. As figuras 3.14 e 3.15 mostram os resultados utilizando 2 sistemas computacionais muito diferentes (Marreca e OGUN com velocidades de processamento e número de CPUs disponíveis diferentes). Pode-se notar que na Figura 3.14, nossa implementação apresenta um desempenho um pouco inferior, permanecendo na mesma ordem de grandeza. Na figura 3.15, pode-se observar um aparente melhor desempenho para valores de NRHS maiores que N . Figuras semelhantes para diferentes tamanhos são apresentados na seção E.2.

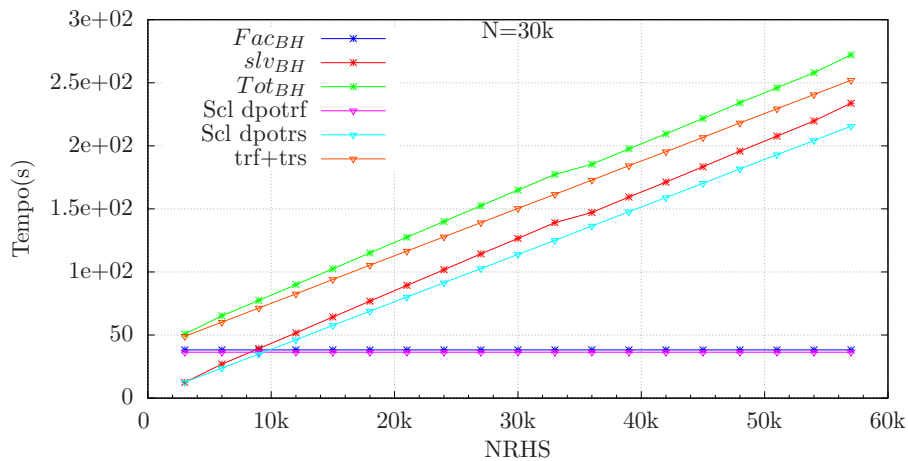


Figura 3.14: Comparação entre o algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dpotrf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS ($3k$ a $60k$), usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

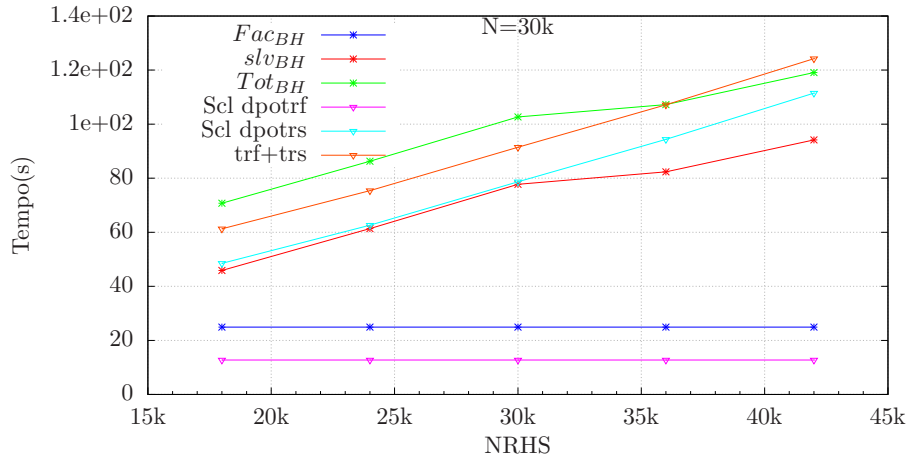


Figura 3.15: Comparação entre o algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dpotrf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS ($3k$ a $60k$), usando um CPU de 300 núcleos (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

Utilizando o sistema marreca, com maior disponibilidade de memória RAM, foram realizados testes onde o número de RHS foi aumentado (ver figura 3.16), onde fica evidente que nossa implementação tem um desempenho inferior ao algoritmo de referência. Este resultado pode ser devido à distribuição cíclica interna dos dados pela biblioteca Scalapack, onde o tamanho do bloco permanece constante mesmo que o tamanho do problema aumente, o que implica uma maior atomização do problema e, geralmente, degrada o desempenho da nossa implementação. Apesar disso, vale ressaltar que as ordens de grandeza permanecem na mesma escala. No Anexo E.2 apresenta resultados com diferentes tamanhos para o mesmo sistema computacional Marreca (Gráficas E.10 e E.11).

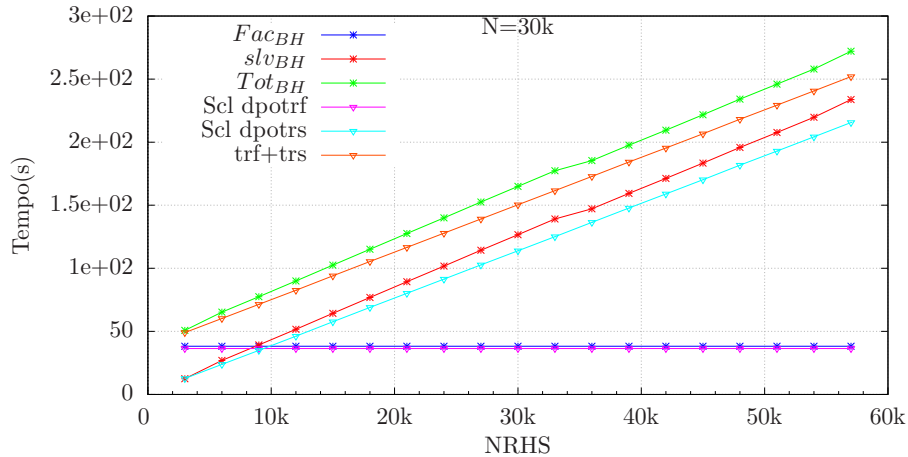


Figura 3.16: Comparação entre o algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dpotrf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e dimensões diferentes de RHS , usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

3.6 Versão OUT-OF-CORE usando GPU

3.6.1 Desempenho

Nesta seção, detalhamos os resultados obtidos com nosso algoritmo Out-Of-Core. As matrizes são de precisão dupla e positiva definida. A matriz \mathbf{A} é construída através de uma matriz pseudo-aleatória \mathbf{G} (subrotina fortran random_number; com valores entre -10 e 10) e realizando a multiplicação $\mathbf{G}^T\mathbf{G}$. O lado direito da equação (\mathbf{D}) é uma matriz pseudo-aleatória com valores entre 0 e 1. Utilizamos duas GPU NVIDIA bem diferentes, a GPU Tesla k40c e Tesla P100 com 12GB e 16GB de memória respectivamente, para executar os experimentos. Os resultados da nossa implementação: tempo, erro relativo da predição e resíduo relativo, serão comparados com aqueles obtidos com magma_dpotrf (Out-Of-Core) + magma_dpotrs (Ver Apêndice D para mais detalhes) da Biblioteca MAGMA, que é o resultado referência para fim de comparação.

3.6.1.1 GPU Tesla k40c

Esta GPU está conectada a uma CPU com processador Intel® Xeon® E5-2643 v3 que possui 524 GB de RAM instalada (Mais detalhes na seção 1.3.2.1). A figura 3.17 mostra a comparação entre os tempos para resolver um sistema NE com $N = 90K$ ($L = 7$ e $N_c = 12864$) para o qual variamos o RHS entre $9K$ e $270K$. Nossa implementação é

mostrada nas legendas: Fac_{BH} , slv_{BH} e Tot_{BH} ; onde Fac_{BH} corresponde ao tempo gasto pelo algoritmo 5; slv_{BH} corresponde ao tempo do algoritmo 6 e Tot_{BH} é o tempo total. Pode-se notar que o tempo registrado para Fac_{BH} é aproximadamente 2,5 vezes maior que o tempo necessário para realizar uma fatoração de Cholesky (`magma_dpotrf`), isso é o esperado, pois o número de operações realizadas por O algoritmo 5 é aproximadamente 3 vezes maior que o da função `magma_dpotrf` (Fatoração de Cholesky).

No caso do algoritmo para slv_{BH} , pode-se observar que ele leva menos tempo que o algoritmo de referência (`magma_dpotrs`) para todos os valores RHS apesar do número de operações ser maior, isso se deve principalmente à maior dependência dos dados na substituição *Forward* e *Backward* para a solução de dois sistemas triangulares. A maior eficiência do algoritmo 6 (ate NRHS igual a $270k$) não é suficiente para compensar o maior tempo gasto no primeiro algoritmo, porem a diferença entre (Tot_{BH}) e o tempo total de referência ($trf+trs$) é pouca. Este mesmo efeito é evidenciado em sistemas NE com $N = 45k$, $N = 60k$ e $N = 75k$, onde a maior eficiência do algoritmo 6 está sempre presente o que permite ter um desempenho similar com o algoritmo de referência (Ver Apêndice E.3.1).

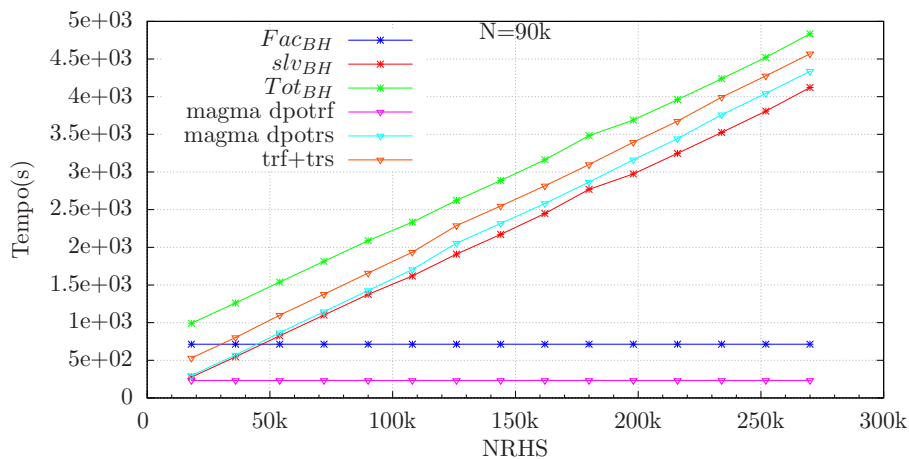


Figura 3.17: Comparação entre `magmaf` (`magmaf_dpotrf` + `magmaf_dpotrs`) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 90k$ ($L = 7$ e $N_c = 12864$) e dimensões diferentes de *RHS* ($9k$ a $270k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

Para ratificar o melhor desempenho do algoritmo 6 mostramos na figura 3.18 o desempenho em Gflops, onde podemos evidenciar que para todos os valores de RHS, a velocidade é maior. Figuras adicionais com valores de RHS anteriores ($N = 45k$, $N = 60k$ e $N = 75k$), são apresentados no apêndice E.3.2).

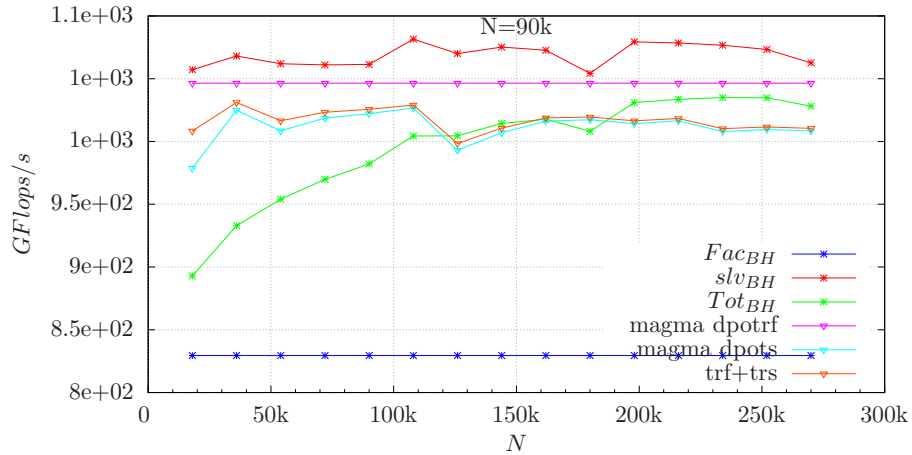


Figura 3.18: Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 90K$ e diferentes valores de $NRHS$ usando a GPU tesla k40c

3.6.1.2 GPU Tesla P100

Esta GPU está conectada a uma CPU com processador Intel® Xeon® Gold 6148 (20 núcleos) que possui 256 GB de RAM instalados (Mais detalhes na seção 1.3.2.3.2). Este sistema possui desempenho consideravelmente superior ao da configuração anterior, mas por possuir menos RAM, o tamanho do sistema NE utilizado foi menor. A figura 3.19 mostra a comparação entre os tempos para resolver um sistema NE com $N = 70k$ ($L = 4$ e $N_c = 17504$) para o qual variamos o RHS entre $14k$ e $217k$. Neste caso, o melhor desempenho do algoritmo 6 para todos os valores de $NRHS$ permitiu manter o tempo total no mesmo ordem de magnitude, solo ligeiramente superior ao algoritmo referência (magma_dpotrf + magma_dpots). Outras provas com diferente tamanho do N são apresentados no Apêndice E.3.3, que ratificam o comportamento.

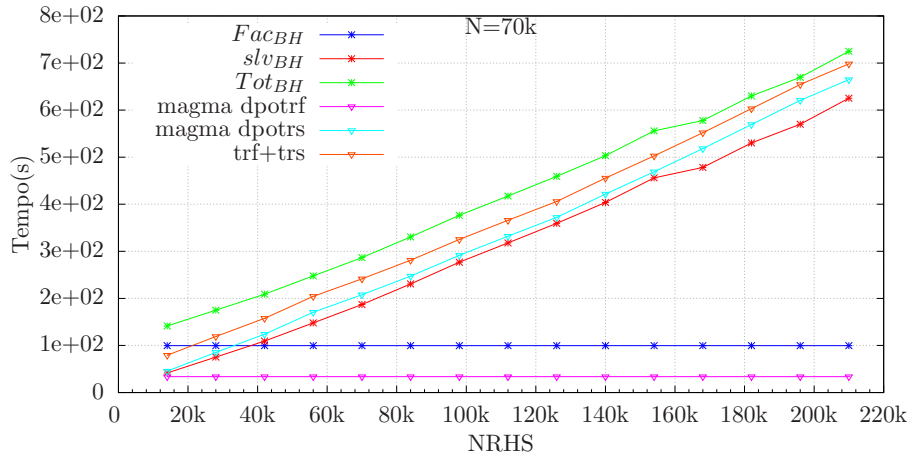


Figura 3.19: Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 70k$ ($L = 5$ e $N_c = 14080$) e dimensões diferentes de RHS ($14k$ a $210k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes

3.6.1.3 Incrementando o NRHS para Tesla k40c

Com o objetivo de avaliar como se comporta o desempenho do algoritmo proposto para valores muito grandes de RHS, foram realizados testes adicionais aumentando o número de RHS, tomando cuidado para não ultrapassar 90% da memória RAM disponível, que para este sistema é de 512 GB. A Figura 3.20 mostra o resultado para um SL de $N = 75k$ com número RHS variando de $15k$ até $615k$. É possível mostrar que à medida que o número de RHS aumenta, as linhas de tempo do magma_dpots e slb_{BH} se separam (inclinação diferente), onde a menor inclinação de slb_{BH} indica um melhor desempenho. Esse melhor desempenho produz que a partir de aproximadamente $500k$ (onde as linhas Tot_{BH} e $trf+trs$ se cruzam) o desempenho geral do nosso algoritmo seja um pouco melhor. Figuras adicionais, para diferentes valores de N , são apresentados no apêndice E.3.4.

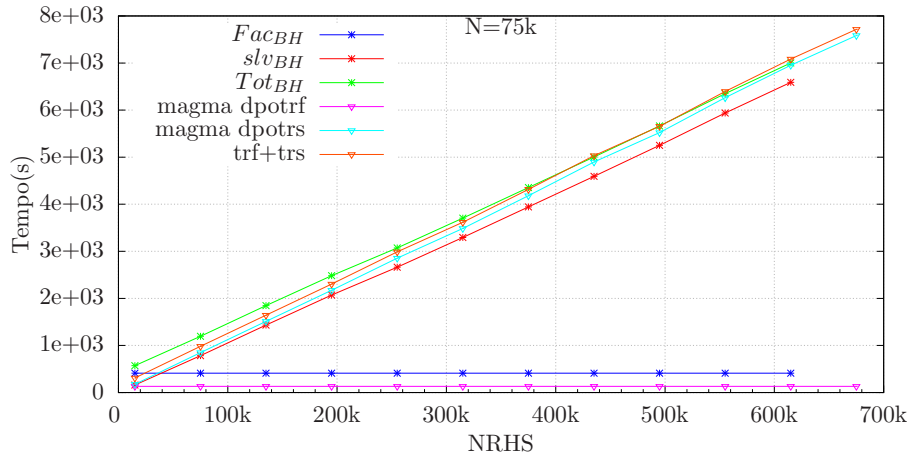


Figura 3.20: Comparação entre magmaf (magmaf dpotrf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 75k$ ($L = 6$ e $N_c = 12544$) e dimensões diferentes de RHS (15k a 615k), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

3.6.2 Resíduo e erro da solução

Além do tempo que o algoritmo leva para encontrar a solução, é importante determinar e comparar o resíduo e o erro da solução. Estes valores são calculados de acordo com o que está expresso em na seção 3.2.2. A figura 3.21 mostra o resíduo e o erro da solução dos dois algoritmos, onde se percebe que são equivalentes.

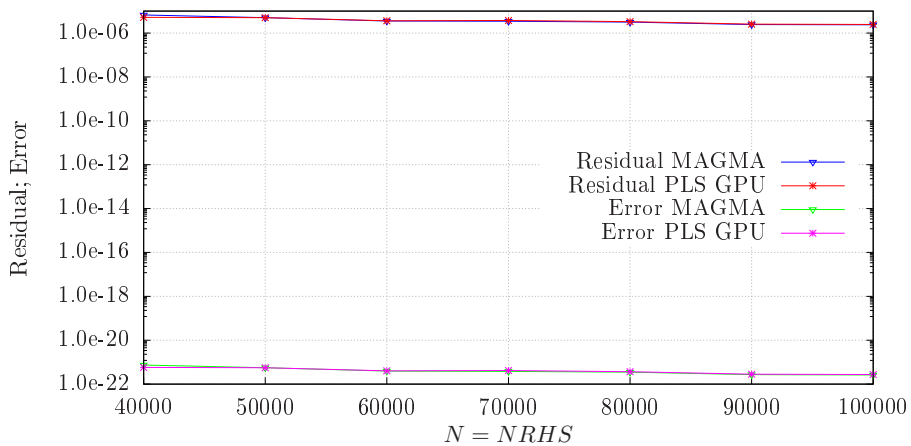


Figura 3.21: Comparação do Resíduo e ou Erro entre a solução magma (MAGMA) e nossas implementações (PLS GPU) para um sistema quando $N = NRHS$ (de 40k a 100k), usando um NVIDIA k40c de 12 GB.

3.6.3 Simulando uma GPU de 2 GB, para testar o efeito de um particionamento maior

Para simular particionamentos maiores (valores L maiores), simulamos uma GPU de apenas 2GB, limitando a memória máxima a ser usada na GPU Tesla k40c. Neste experimento variamos N de $25k$ a $50k$ a cada $5k$, os valores de RHS variaram de $N/5$ a $3N$ em cada caso. Na figura 3.22 é apresentado o resultado para $N = 50k$, os demais podem ser consultados no apêndice E.3.5. Os resultados mostram a mesma tendência indicada nas seções anteriores.

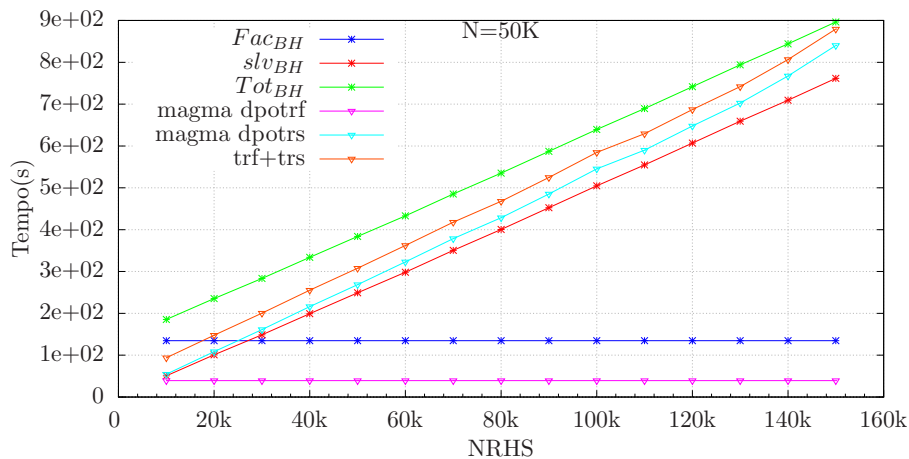


Figura 3.22: Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L . Comparação entre magma f (magma f dpotrf + magma f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 50k$ ($L = 8$ e $N_c = 6272$) e dimensões diferentes de RHS ($10k$ a $150k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

3.7 Por que tem uma melhora para vários RHS?

Para muitos RHS ($NRHS \approx N_{size}$), a maioria do tempo é gasto na substituição direta e reversa (*forward and backward*). O desempenho da multiplicação de matrizes ($C = C + AB$ e $C = AB$) é mais eficiente do que a fatoração de Cholesky (LU) e a substituição direta e reversa na solução dos dois sistemas triangulares $FB_{NRHS=N}$. Na Gráfica 3.23 se apresenta a comparação, para o caso serial usando um único processador, da velocidade ($GFLOPS/s$) entre essas diferentes operações. Pode-se notar que a operação de multiplicação de matrizes sempre tem um desempenho superior (maior velocidade em $GFLOPS/s$).

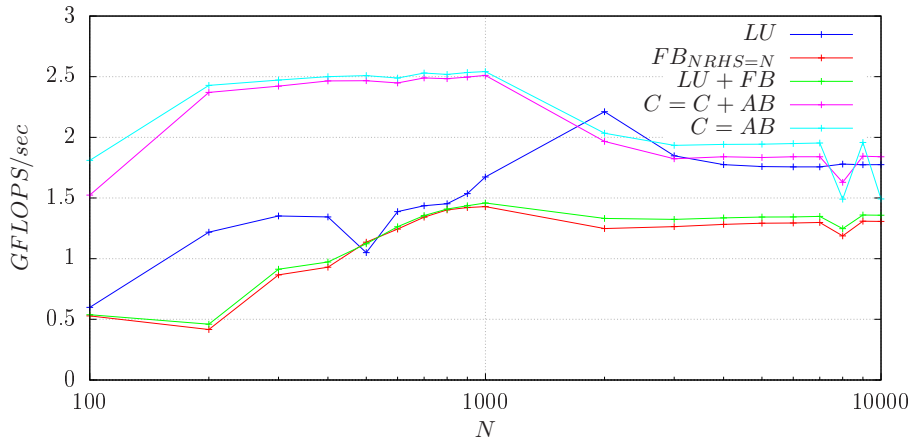


Figura 3.23: Comparação entre o desempenho da fatoração de Cholesky (LU, usando DPOTRF do LAPACK), resolve um sistema de equações lineares usando fatoração de Cholesky (FB, substituição direta e reversa usando DPOTRS do LAPACK) para a matriz $N \times N$ e $NRHS = N$. Desempenho para multiplicação de matrizes, com implementação explícita, para tamanho de matriz $N \times N$

Para sustentar nossa afirmação, realizamos alguns testes, usando uma implementação serial usando um único processador, onde comparamos nosso algoritmo com a versão do LAPACK (DPOTRF + DPOTRS). Nas figuras 3.24 e 3.25, onde variamos o tamanho do bloco N_c para um $N = 10k$, variamos o NRHS de 10 até 10k, podemos notar que nosso algoritmo demora mais na primeira parte (Fac_{BH}) ao ser comparado com a referência DPOTRF (LL_{lapack}^T) como é de esperar, já que a quantidade de operações é maior. Na hora de calcular a solução slv_{BH} tem o melhor desempenho da referência DPOTRS (FB_{lapack}).

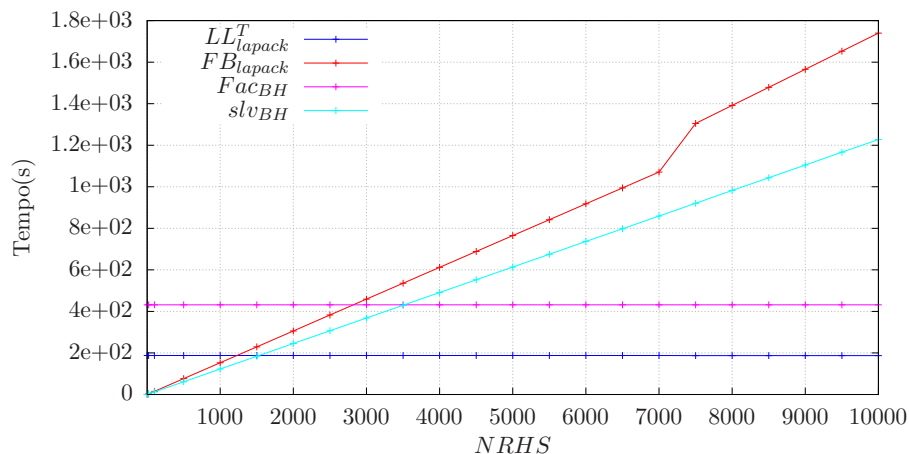


Figura 3.24: Comparação entre LAPACK (DPOTRS, substituição direta e reversa) e nossas implementações (usando $N_c = 500$) para dimensões diferentes de $NRHS$ para NE $N \times N$ quando $N = 10000$.

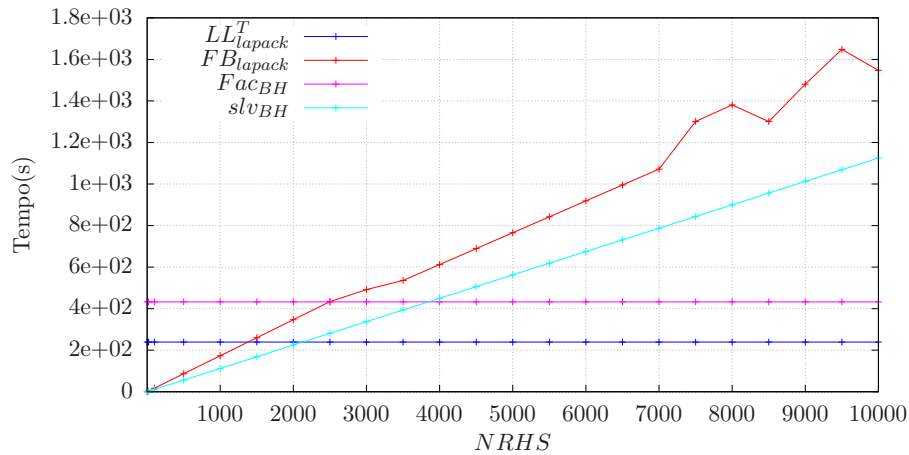


Figura 3.25: Comparação entre LAPACK (DPOTRS, substituição direta e reversa) e nossas implementações (usando $N_c = 1000$) para dimensões diferentes de $NRHS$ para NE $N \times N$ quando $N = 10000$.

Na figura 3.26 realizamos um teste diferente, para cada valor de N, usamos um único valor de NRHS e N as variações de 2k até 18k para diferentes tamanhos de N, onde NRHS é igual a N. Nesta figura você pode notar que a medida que o tamanho ($N = NRHS$) cria diferenças e o desempenho se torna mais evidente. Por exemplo, para resolver um sistema de tamanho 18.000 com $NRHS = 18.000$, usando precessão dupla, nosso algoritmo leva 18% menos tempo que as sub-rotinas LAPACK. Os mesmos dados são apresentados na figura 3.27, onde o eixo x $N = NRHS$ é mostrado na escala \log_2 e o eixo y (tempo) na escala \log_{10} para melhor visualização. Nosso algoritmo é mais caro para a fatoração, mas é mais eficiente para resolver vários NRHS

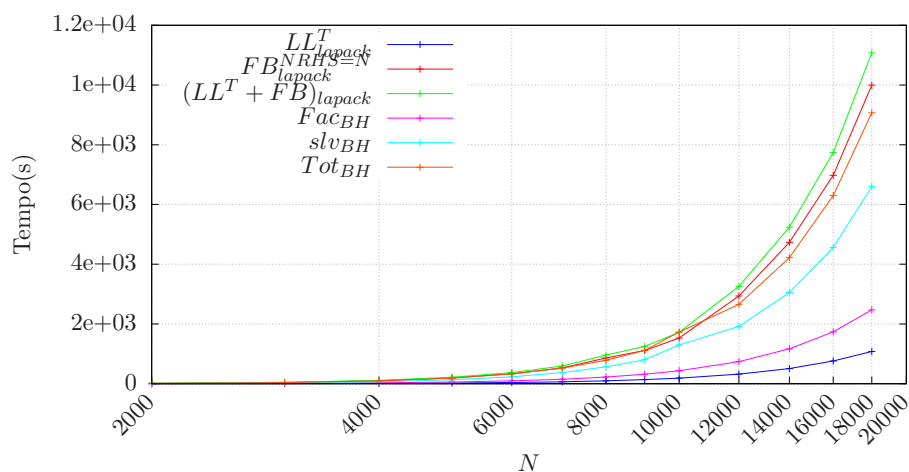


Figura 3.26: Comparação entre LAPACK (DPOTRF+DPOTRS) e nossas implementações para sistemas de diferentes tamanhos N e $NRHS = N$.

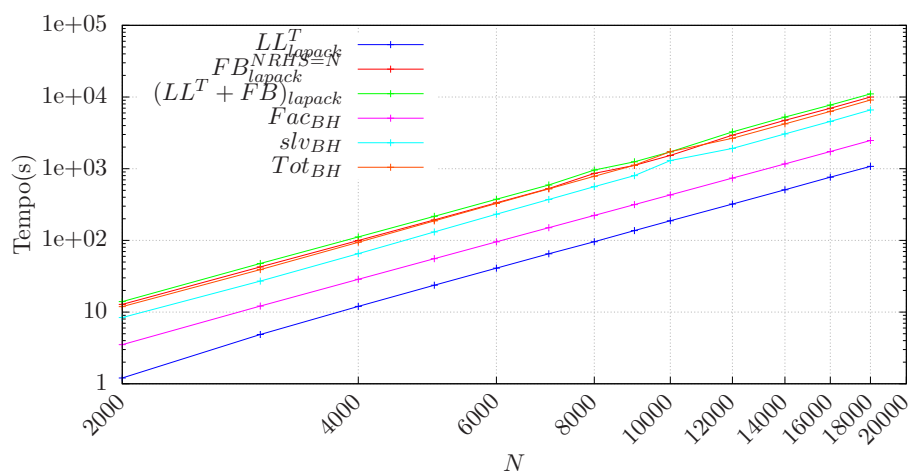


Figura 3.27: Comparação entre LAPACK (DPOTRF+DPOTRS) e nossas implementações para sistemas de diferentes tamanhos N e $NRHS = N$.

4

Modelagem acústica usando diferenças finitas no domínio da frequência

Neste capítulo apresentaremos resultados da utilização do algoritmo recursivo bloco particionado do tipo Levinson para modelagem sísmica 2-D no domínio da frequência. Para isso começaremos por fazer uma pequena introdução ao método de Modelado acústico usando diferenças finitas no domínio da frequência (FDFD pela sua sigla em inglês de *Frequency-Domain Finite-Difference*).

A modelagem acústica usando diferenças finitas no domínio da frequência é usada na migração sísmica (Mulder e Plessix, 2004, Kim et al., 2011, Ma et al., 2023) e na inversão sísmica usando a equação completa da onda, FWI das siglas do inglês *Full Waveform Inversion* (Pratt e Worthington, 1990, Pratt, 1990, Ben-Hadj-Ali et al., 2011, Luo e Xie, 2017). Comparado com o método de diferenças finitas no domínio do tempo (Dablain, 1986, Liao et al., 2018), o método FDFD tem as vantagens de ter menor erro cumulativo, alta eficiência para modelagem de dados sísmicos multi-shot (Jo et al., 1996, Ben-Hadj-Ali et al., 2011), conveniência da computação paralela e flexibilidade para modelar os efeitos da atenuação (Pratt e Worthington, 1990). A modelagem acústica usando FDFD resolve sistemas lineares de grande porte. Geralmente, são usados *solvers* diretos como a fatoração LU. Quando o modelado envolve múltiplos tiros, cada um deles é um vetor no lado direito do sistema linear, portanto se houver milhares de tiros, envolveria milhares de RHS.

4.1 Modelagem Acústica usando FDFD

A equação de onda 2D acústica no domínio do tempo (meio com densidade constante) contempla somente a propagação de ondas compressoriais (*compressional waves*), ou seja ondas P, ela descreve a evolução temporal das ondas compressoriais P no modelo geológico é conhecida na literatura como a equação acústica da onda. Ela é uma equação hiperbólica de segunda ordem e pode ser deduzida a partir das leis de Newton e de Hooke, considerando um meio com densidade constante, é dada pela seguinte expressão:

$$\nabla^2 p(x, z, t) - \frac{1}{c^2(x, z)} \frac{\partial p(x, z, t)}{\partial t^2} = -s(x_s, z_s, t) \quad (4.1)$$

onde

- $c(x, z)$ = Velocidade de propagação da onda P
- $p(x, z, t)$ = Campo de pressão no ponto (x, z) num dado tempo t .
- $s(x_s, z_s, t)$ = termo fonte no domínio do tempo localizado na posição (x_s, z_s) num dado tempo t
- $\nabla^2 = \frac{\partial}{\partial x^2} + \frac{\partial}{\partial z^2}$ Operador Laplaciano

Aplicando a transformada de Fourier temporal na Eq. 4.1, obtém-se a equação da onda acústica 2-D no domínio da frequência, chamada de equação de Helmholtz:

$$\nabla^2 P(x, z, w) + \frac{w^2}{c^2(x, z)} P(x, z, w) = -S(x_s, z_s, w) \quad (4.2)$$

onde $w = 2\pi f$ é a frequência angular. Entretanto, $P(x, z, w)$ e $S(x_s, z_s, w)$ são o campo de pressão e o termo fonte no domínio da frequência angular, os quais são calculados através das transformadas de Fourier de $p(x, z, t)$ e $f(x_s, z_s, t)$ respetivamente.

$$\begin{aligned} P(x, z, w) &= \int_{-\infty}^{\infty} p(x, z, t) e^{-iwt} dt \\ S(x_s, z_s, w) &= \int_{-\infty}^{\infty} s(x_s, z_s, t) e^{-iwt} dt \end{aligned} \quad (4.3)$$

Informações adicionais do par $s(x, z, t)$ e $S(x_s, z_s, w)$ serão descritas na secção 4.2.1.

4.1.1 Modelagem FDFD: Discretização da Equação de Helmholtz

A discretização da equação de onda no domínio da frequência (Eq 4.2) resulta em um sistema linear de equações representados como (Brossier et al., 2010):

$$\mathbf{MP} = -\mathbf{S} \quad (4.4)$$

onde

- \mathbf{M} = é a matriz impedância complexa
- \mathbf{P} e \mathbf{S} são os vetores campo de pressão e a fonte respectivamente.

Esta estrutura de sistema linear de equações evidencia uma das vantagens do FDFD em comparação com o TDFD (*Time-Domain Finite-Difference*), pois permite a modelagem monocromática de uma aquisição de famílias de tiro comum usando a mesma matriz de impedância e adicionando múltiplos lados direitos (neste caso, \mathbf{S} é uma coleção de vetores, um para cada tiro), portanto \mathbf{P} será uma coleção de vetores o campo de pressão monocromático para cada tiro (Pratt e Worthington, 1990, Jo et al., 1996, Ajo-Franklin, 2005). Além disso, como a modelagem é na frequência, não tem a necessidade de definir um *time-stepping* que satisfaça o critério de *Courant-Friedrichs-Lewy*. Por outro lado, uma das desvantagens do FDFD em comparação com o TDFD é que apenas campos de onda monocromáticos são modelados, o que faz necessário modelar diferentes frequências para obter os sismogramas de domínio do tempo o que geralmente requer mais recursos computacionais do que TDFD, No entanto, esta particularidade torna-se uma vantagem no momento de realizar FWI multifrequência (Pratt e Worthington, 1990, Pratt, 1990).

Para o caso de meios não atenuantes e sem fronteira absorvente, usando um operador de 2^{a} e 4^{a} ordem, a matriz de impedância (\mathbf{M}) é quadrada, esparsa, simétrica, os elementos da diagonal \mathbf{M} dependem w , c , Δx , Δz e os elementos fora da diagonal de \mathbf{M} só depende do espaçamento da malha (Ajo-Franklin, 2005).

O fluxo para a geração de sismogramas sintéticos no domínio do tempo a partir da solução da equação de Helmholtz (Eq. 4.2) é resumido na Figura 4.1.

O primeiro bloco refere-se à parametrização do problema, o que implica escolher qual é o esquema de diferenças finitas a ser utilizado, tipo de borda de atenuação, tipo de fonte e sua respectiva FFT, assim como o fornecimento do modelo de velocidade.

O segundo bloco corresponde ao início do laço na frequência que termina na frequência de corte, definida pelo usuário. A construção da matriz de impedância para uma dada frequência dependerá do operador de diferenças finitas a ser usado, largura e tipo de borda absorvente assim como, o modelo de velocidade. Fatoração LU (ou o algoritmo usado nesta tese) permite obter o campo de pressão $P(x, z, w)$ para uma fonte ou múltiplas fontes. É necessário pré-definir as estações de registro para a correta seleção dos campos de pressão.

Depois de terminar o laço na frequência, a obtenção do sismograma no domínio do tempo é obtida mediante a aplicação da transformada inversa de Fourier (IFFT) sobre o

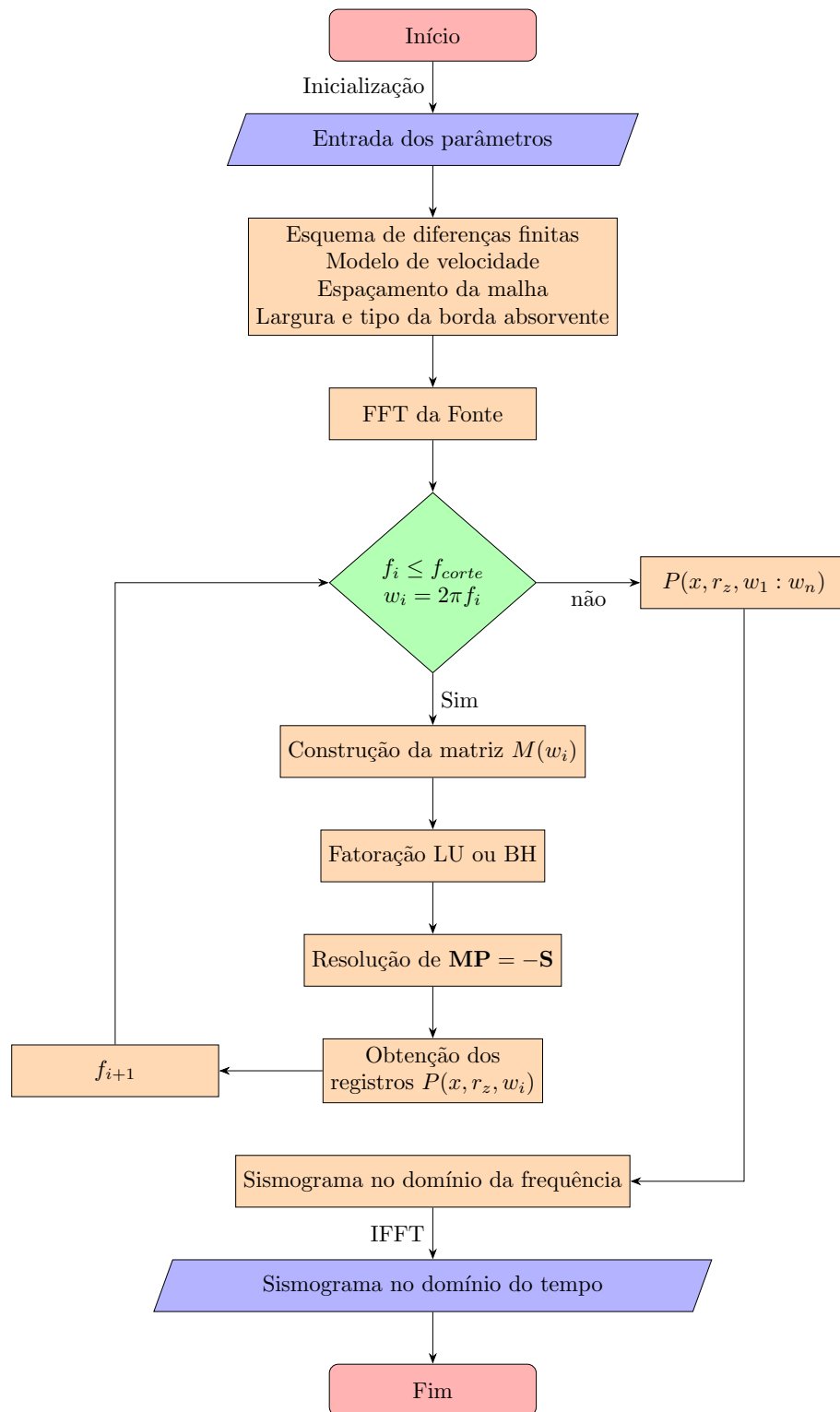


Figura 4.1: Fluxograma para obtenção de sismogramas no domínio do tempo a partir da modelagem sísmica no domínio da frequência. Adaptado de Revelo (2015).

sismograma no domínio da frequência, que por sua vez é obtido agrupando os registros em cada frequência.

4.2 Parametrização

Nesta seção apresentaremos o tipo de fonte sísmica, o tipo de fronteira absorvente, o modelo de velocidade e os tipos de operador Laplaciano usado na discretização.

4.2.1 Fonte sísmica: Ricker

Uma *wavelet* Ricker é frequentemente usada como fonte de ondas sísmicas por ser de fase zero, estar limitada no domínio do tempo e no domínio da frequência. A *wavelet* Ricker é a segunda derivada da função gaussiana ou a terceira derivada da função de densidade de probabilidade normal¹. O parâmetro da fonte usada foi uma frequência pico de $f_{\text{peak}} = 15\text{Hz}$

A amplitude $S(t)$ da *wavelet* Ricker no tempo t com uma frequência pico $f_{\text{peak}} = 15\text{Hz}$ é dada por:

$$s(t) = (1 - 2\pi^2 f_{\text{peak}}^2 t^2) e^{-\pi^2 f_{\text{peak}}^2 t^2} \quad (4.5)$$

e a representação no domínio da frequência da *wavelet* Ricker é dada por,

$$S(f) = \left(\frac{2f^2}{\sqrt{\pi} f_{\text{peak}}^3} \right) e^{-\frac{f^2}{f_{\text{peak}}^2}} \quad (4.6)$$

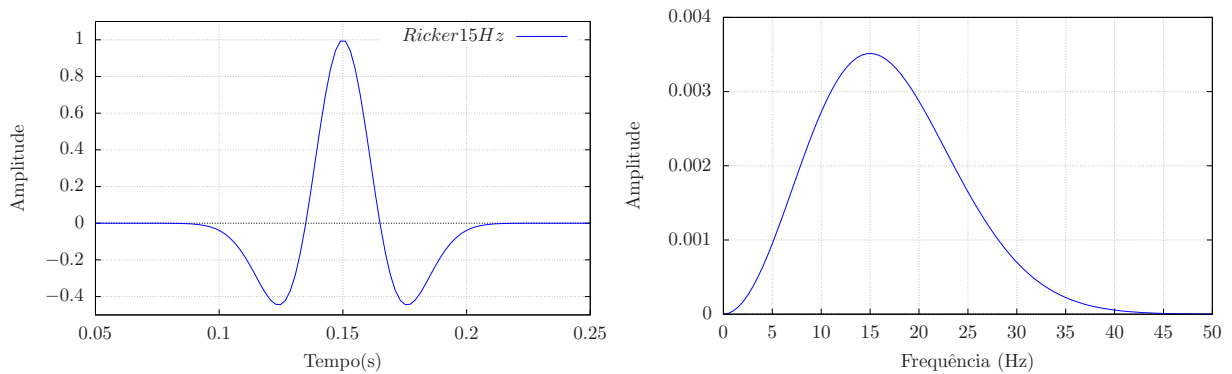


Figura 4.2: Representação de uma *wavelet* tipo Ricker com uma frequência pico igual a 15Hz e seu correspondente espectro de amplitude

4.2.2 Fronteira absorvente

Para obter a solução da equação discretizada (Eq 4.2), é necessário aplicar condições de contorno, de forma que não haja energia refletida nos limites do domínio. Neste trabalho,

¹SEG Dictionary:Ricker wavelet https://wiki.seg.org/wiki/Dictionary:Ricker_wavelet

foi usada uma ABC (*Absorbing Boundary Conditions*) determinada pelas seguintes equações (Cerjan et al., 1985):

$$\begin{aligned} taper_x(i) &= e^{-(F_x * (nx_b - i))^2}, & 1 \leq i \leq nx_b \\ taper_z(k) &= e^{-(F_z * (nz_b - k))^2}, & 1 \leq k \leq nz_b, \end{aligned} \quad (4.7)$$

onde, F_x e F_z são os coeficientes de absorção, y nx_b , nz_b é o comprimento da fronteira em as direções x e z . Os valores usados nesta secção são $F_x = F_z = 0.015$ e $nx_b = nz_b = 20$.

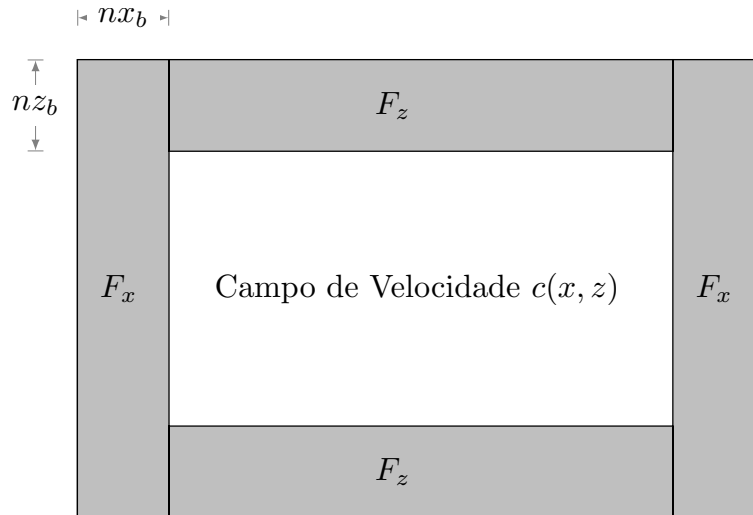


Figura 4.3: Modelo da borda absorvente (ABC), onde nx_b e nz_b indica o número de pontos da borda em as direções x e z .

4.2.3 Esquemas Explícitos de Diferenças Finitas

Nesta seção, são apresentados os esquemas explícitos de diferenças finitas usados para expressar as derivadas espaciais de segunda ordem contidas no Laplaciano da equação 4.2. Apresentaremos os esquemas convencionais de diferenças finitas de 2ª e 4ª ordens. Além disso, também foi utilizado o esquema compacto de 9 pontos.

As discretizações realizadas nesta seção fazem uso de uma malha com espaçamento regular entre os pontos nas direções x e z ($\Delta x = \Delta z = \Delta h$).

4.2.3.1 Operador Laplaciano de segunda ordem

O esquema de diferenças finitas para o operador Laplaciano de segunda ordem é mostrado na Figura 4.4, as referências locais dos pontos da malha são realizadas mediante os índices i e j , onde i varia na direção x e j varia em z .

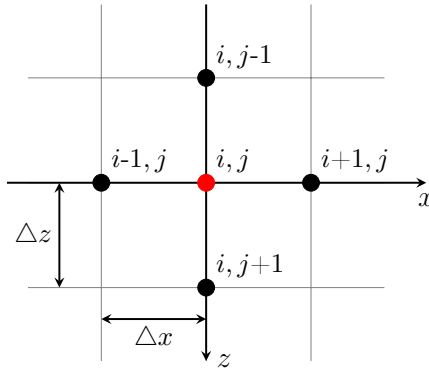


Figura 4.4: Esquema de diferenças finitas para o operador Laplaciano de segunda ordem

Reescrevendo a equação 4.2 (sim ter em conta a borda absorvente) em termos dos índices i, j , temos:

$$\nabla^2 P + \frac{w^2}{c_{i,j}^2} P_{(j,i)} = -S_{i,j} \quad (4.8)$$

As derivadas parciais em x e z do campo $P(x, z, w)$ podem ser discretizadas mediante sua expansão em serie de Taylor, e eliminando aquelas de ordem superior a dois, obtém-se a seguinte aproximação para a segunda derivada

$$\nabla^2 P \approx \frac{P_{i,j-1} + P_{i,j+1} + P_{i-1,j} + P_{i+1,j} - 4P_{i,j}}{(\Delta h)^2} \quad (4.9)$$

rescrevendo a 4.8, temos

$$\frac{P_{i,j-1} + P_{i,j+1} + P_{i-1,j} + P_{i+1,j} - 4P_{i,j}}{(\Delta h)^2} + \frac{w^2}{c_{i,j}^2} P_{i,j} = -S_{i,j} \quad (4.10)$$

Reorganizando temos a discretização da equação 4.8:

$$\frac{P_{i,j-1} + P_{i-1,j}}{(\Delta h)^2} + \left(\frac{w^2}{c_{i,j}^2} - \frac{4}{(\Delta h)^2} \right) P_{i,j} + \frac{P_{i,j+1} + P_{i+1,j}}{(\Delta h)^2} = -S_{i,j} \quad (4.11)$$

que pode ser reescrito na forma matricial da forma

$$\mathbf{MP} = -\mathbf{S} \quad (4.12)$$

onde

- \mathbf{M} = matriz impedância
- \mathbf{P} vetores campo de pressão
- \mathbf{S} vetor da fonte

\mathbf{M} é construída indexando a função discretizada de cada nó do domínio computacional mediante um novo índice l , dado por:

$$l = (i - 1)nz + j \quad i = 1, \dots, nx \quad j = 1, \dots, nz \quad (4.13)$$

Como resultado da geometria do esquema utilizado na discretização, a matriz \mathbf{M} apresenta um padrão esparsa, de grande porte e, por conta da condição de fronteira implementada neste trabalho, é simétrica e tem elementos complexos, ver na cor verde na Figura 4.5, nesta figura, n_{xe} e n_{ze} faz referência ao modelo estendido com borda absorvente, onde o modelo é 6×6 pontos e o tamanho da borda é 2 pontos em cada direção.

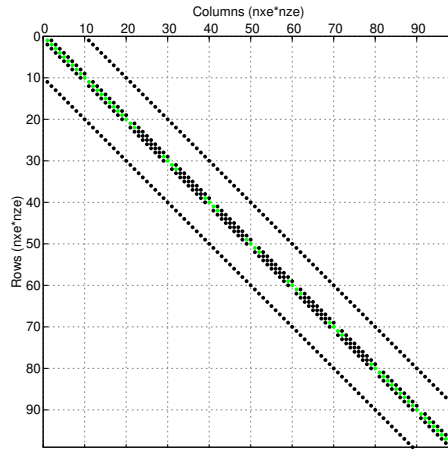


Figura 4.5: Matriz de impedância usando um operador de segunda ordem. O modelo $n_{xe} = 10$ $n_{ze} = 10$, com $\Delta x = \Delta z = \Delta h$

4.2.3.2 Operador Laplaciano de quarta ordem

As derivadas podem ser estimadas com maior exatidão se mais pontos, além dos adjacentes, forem usados, o que dá origem aos operadores de diferenças finitas de mais alta ordem. Por exemplo, se os dois vizinhos foram usados em cada direção, ou seja, um operador de 4ª ordem (Figura 4.6), temos que a discretização da equação 4.8 seria dada por:

$$\begin{aligned} \frac{-P_{i,j-2} - P_{i-2,j}}{12(\Delta h)^2} + \frac{4(P_{i,j-1} + P_{i-1,j})}{3(\Delta h)^2} + \left(\frac{w^2}{c_{i,j}^2} - \frac{5}{(\Delta h)^2} \right) P_{i,j} + \\ \frac{4(P_{i,j+1} + P_{i+1,j})}{3(\Delta h)^2} + \frac{-P_{i,j+2} - P_{i+2,j}}{12(\Delta h)^2} = -S_{i,j} \end{aligned} \quad (4.14)$$

que pode ser reescrito na mesma forma matricial expressada na equação 4.13.

No nosso caso, a matriz de impedância seria semelhante à mostrada na Figura 4.7, novamente n_{xe} e n_{ze} faz referência ao modelo estendido com borda absorvente, onde o modelo

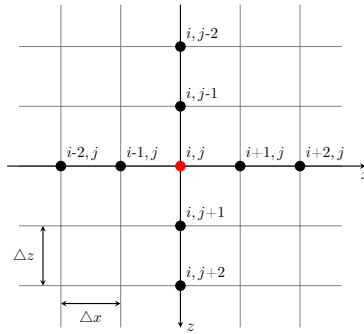


Figura 4.6: Esquema de diferenças finitas para o operador Laplaciano de segunda ordem, com $\Delta x = \Delta z = \Delta h$

é 6×6 pontos e o tamanho da borda é 2 pontos em cada direção.

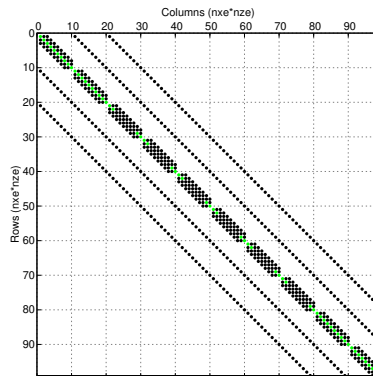


Figura 4.7: Matriz impedância usando um operador de segunda ordem, o modelo $n_{xe} = 10$ $n_{ze} = 10$, com $\Delta x = \Delta z = \Delta h$

4.2.3.3 Operador Laplaciano de 9 pontos

Operadores de ordem superior são utilizados com o objetivo de obter maior precisão, porém, no domínio da frequência a utilização destes operadores leva a uma maior largura de banda da matriz de impedância, conseqüentemente maior consumo de memória e maior custo computacional, o que implica um menor interesse para modelagem usando FDFD. Para superar esta desvantagem, Jo et al. (1996) propuseram um esquema compacto de 9 pontos o qual faz uma aproximação do termo Laplaciano e o termo $w^2/c^2 P$, da equação 4.2.

No esquema proposto por Jo et al. (1996), o operador Laplaciano é expressado como a combinação linear de dois operadores de 2ª ordem, construídos em dois sistemas diferentes de coordenadas rotacionadas. Por exemplo, através da média ponderada de dois esquemas de 5 pontos, formulados nos sistemas de coordenadas cartesianas clássico e rotacionado de 45° (Figura 4.8). Com o aumento da velocidade de processamento dos sistemas computacionais

nas últimas décadas, esquemas semelhantes com mais pontos foram propostos recentemente (Shin e Sohn, 1998, Gu et al., 2013, Xu et al., 2021).

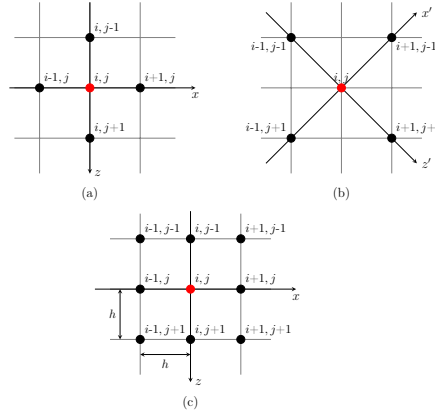


Figura 4.8: (a) Operador de 5 pontos convencional, (b) operador de 5 pontos rotacionado 45° e (c) esquema compacto de 9 pontos. Tomada de Revelo (2015)

A precisão da malha compacta de 9 pontos mostrada na Figura 4.8c é melhorada através da distribuição ponderada do termo de aceleração de massa sobre os diferentes pontos envolvidos na obtenção do mesmo. A discretização da equação 4.8 seria dada por (Jo et al., 1996):

$$\begin{aligned}
 & a \frac{P_{i+1,j} + P_{i-1,j} - 4P_{i,j} + P_{i,j+1} + P_{i,j-1}}{(\Delta h)^2} \\
 & + (1-a) \frac{P_{i+1,j+1} + P_{i-1,j+1} - 4P_{i+1,j-1} + P_{i+1,j-1} + P_{i-1,j+1}}{(\Delta h)^2} \\
 & + \frac{w^2}{c_{i,j}^2} [bP_{i,j} + c(P_{i+1,j} + P_{i-1,j} + P_{i,j-1} + P_{i,j+1})] \\
 & + \frac{1-b-4c^2}{4} (P_{i-1,j-1} + P_{i+1,j-1} + P_{i+1,j+1} + P_{i-1,j+1}) \\
 & = -S_{i,j}
 \end{aligned} \tag{4.15}$$

De acordo com Jo et al. (1996), os coeficientes empregados na combinação linear dos operadores Laplacianos e na aproximação do termo de aceleração de massa são determinados através da minimização dos erros da velocidade de fase, os quais têm os seguintes valores:

$$a = 0.5461 \quad b = 0.6248 \quad c = 0.09381 \tag{4.16}$$

No nosso caso, tendo em conta a borda absorvente, a matriz de impedância seria semelhante à mostrada na Figura 4.9, novamente n_{xe} e n_{ze} faz referência ao modelo estendido com borda absorvente, onde o modelo é 6×6 pontos e o tamanho da borda é 2 pontos em

cada direção. Pode-se observar que a largura de banda da matriz impedância é inferior ao do operador de 4ª ordem mostrado na Figura 4.7.

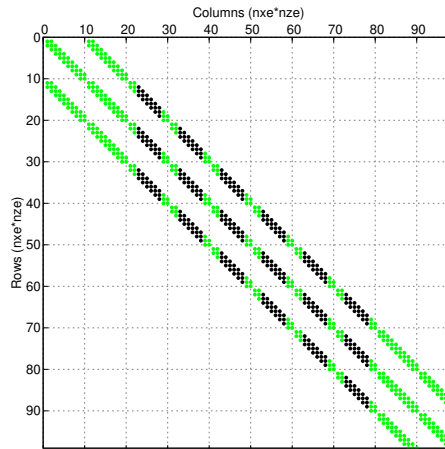


Figura 4.9: Padrão esparsa da matriz impedância do operador compacto de 9 pontos, supondo um modelo $n_{xe} = 10$ $n_{ze} = 10$

4.2.4 Modelo de vp : três camadas

O modelo de velocidade usado é apresentado na figura 4.10, ele é composto de três camadas horizontais, com velocidades de 2400 m/s , 3200 m/s e 4000 m/s , respectivamente. O modelo tem um comprimento de 1600 m e uma profundidade de 1600 m . Esta configuração é conseguida com 81 pontos separados por 20 m nos eixos x e z .

4.3 Resultados

Além dos parâmetros descritos acima, para gerar o sismograma sistemático, foram levados em consideração os seguintes parâmetros adicionais:

- $\Delta t = 0.002\text{ s}$ do sismograma no tempo
- $nt = 801$, numero de amostras no tempo
- $\Delta f = \frac{1}{\Delta t * nt}$; delta de f usada.
- $nw = \frac{4f_{\text{peak}}}{\Delta f} + 1$; numero de frequências modeladas

A fonte tipo Ricker foi inserida na posição $(20\text{ m}, 20\text{ m})$, além, 81 receptores foram distribuídos regularmente com 20 m de separação e 20 m de profundidade em um arranjo *Single-ended spread*.

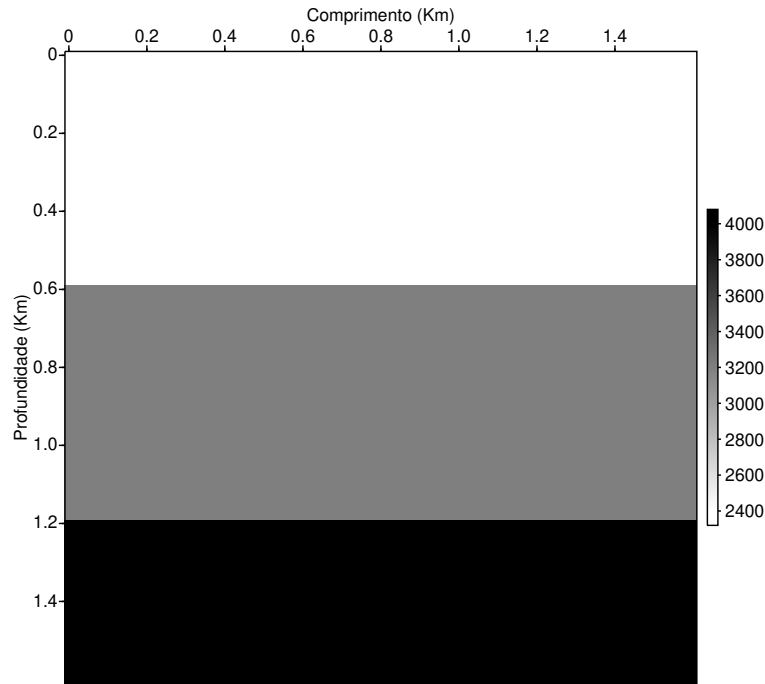


Figura 4.10: Modelo de velocidade 3 camadas, $n_z = 81, n_x = 81 \quad dx = dz = 20m$

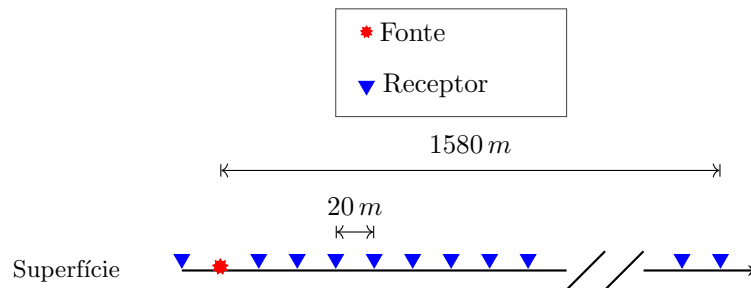


Figura 4.11: Arranjo do tipo *Single-ended spread* usado para a obter o tiro simulado

Para obter a solução do sistema linear foram utilizadas duas opções, a primeira o pacote MUMPS (*MUltifrontal Massively Parallel sparse direct Solver*), como algoritmo de referência; este pacote desenvolvido por Amestoy et al. (2001) . O MUMPS é escrito em FORTRAN e fornece subrotinas para a resolução de sistemas de equações lineares, assim como a fatoração LU de matrizes esparsas. Independente das operações, as funções podem ser aplicadas em matrizes reais e complexas, tanto em precisão simples quanto em dupla precisão. A segunda é o algoritmo recursivo particionado de Levinson descrito nos capítulos 2 e 3 desta tese, tomando especial cuidado na necessidade de reescrever a equação 2.42, para o caso de uma matriz complexa não-Hermitiana. O nível de particionamento (L) utilizado é de acordo com o tamanho do modelo estendido, de forma que o tamanho do bloco (N_c) coincida com o número de pontos do modelo de velocidade estendido.

Nas Figuras 4.12, 4.13 e 4.14 são mostrados os resultados obtidos utilizando os três esquemas de diferenças finitas: 2^a ordem, 4^a ordem e compacto de 9 pontos. Em cada caso, o resultado é mostrado usando o solucionador MUMPS (na subfigura esquerda) e o solucionador recursivo de bloco particionado de Levinson detalhado nesta tese (subfigura direita).

Em geral, os sismogramas sintéticos mostraram ser semelhantes usando os dois solucionadores. Porém, com o operador de 4^a ordem e o operador compacto de 9 pontos, o ruído que pode estar associado à dispersão numérica (linhas inclinadas paralelas à onda direta) torna-se mais evidente, já que a única diferença é o tipo de algoritmo usado para resolver o sistema linear de equações.

Esta dispersão surge devido à forma como a solução é obtida, embora a matriz de impedância seja esparsa, para cada aplicação de recursão (Eq 2.38), a solução direita (F) de ordem j (Eq. 2.39) e a solução reversa (B) de ordem j (Eq 2.40) tornam-se rapidamente densas, especialmente com um nível de particionamento grande, como o usado nesta seção ($L=121$). Isso não ocorre na fatoração LU de uma matriz esparsa, que é esparsa, apesar de possuir maior número de elementos não nulos (Amestoy et al., 2001, Brossier et al., 2010). Essa particularidade torna a recursão de bloco particionado de Levinson pouco atrativa para sistemas esparsos não Toeplitz. Para reduzir este efeito seria necessário limitar o particionamento a pequenos valores de L (2 ou 3).

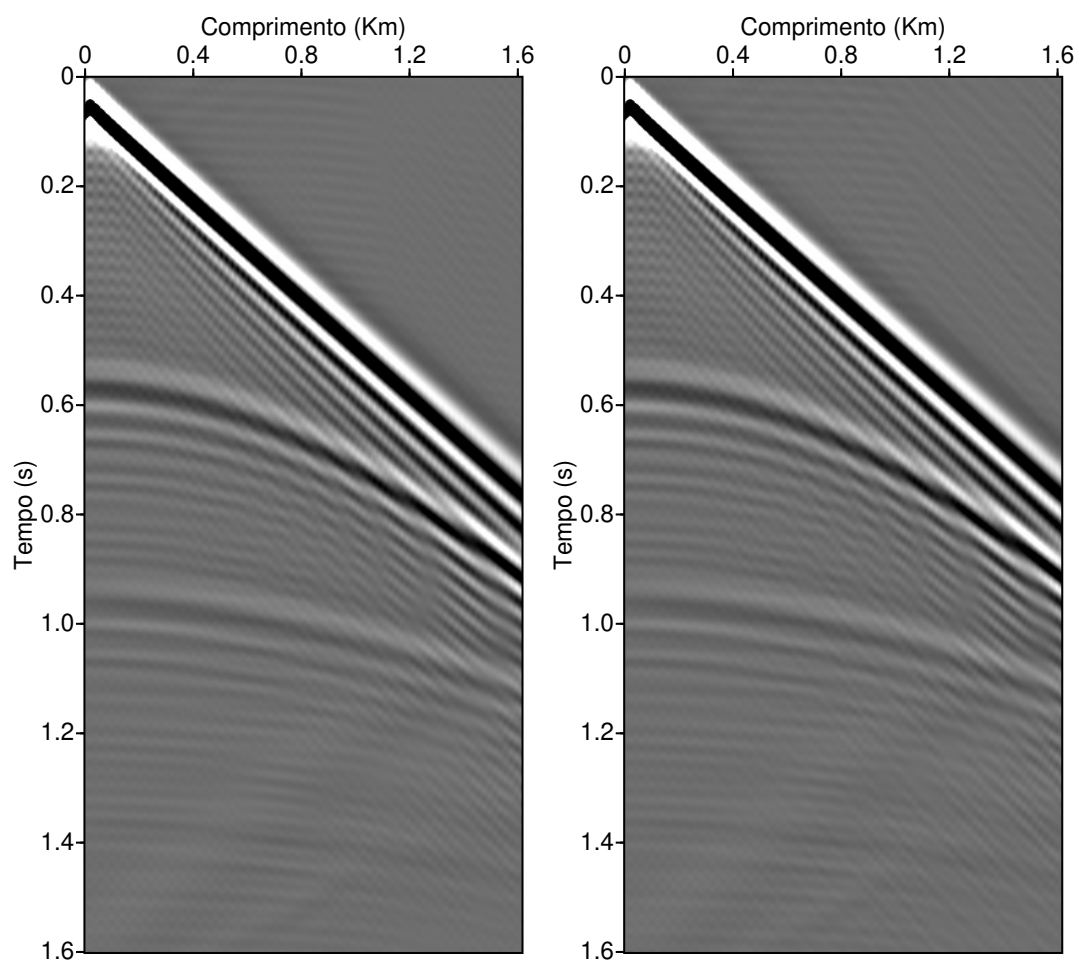


Figura 4.12: (a) *Shot* modelado referencia, (b) *shot* modelado usando $L=121$, PLS

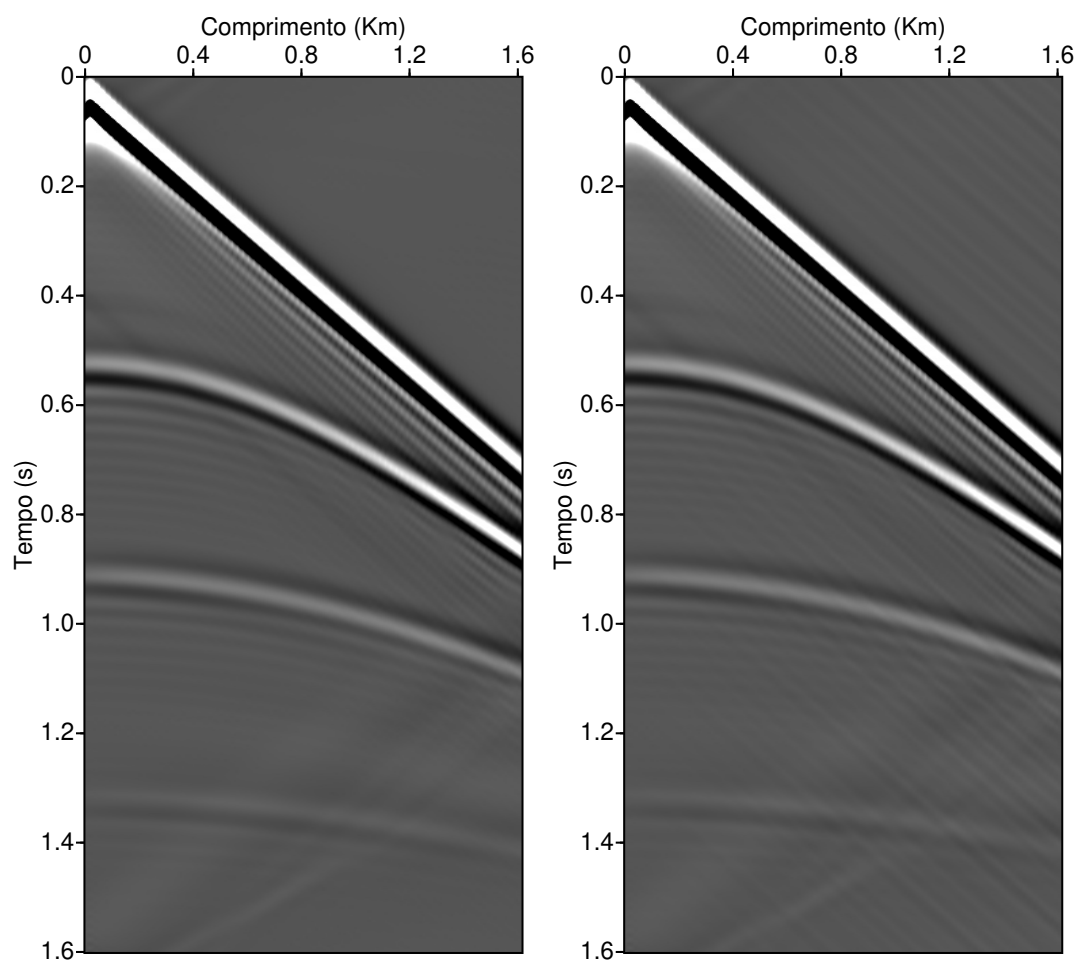


Figura 4.13: (a) *Shot* modelado referencia, (b) *shot* modelado usando $L=121$, PLS

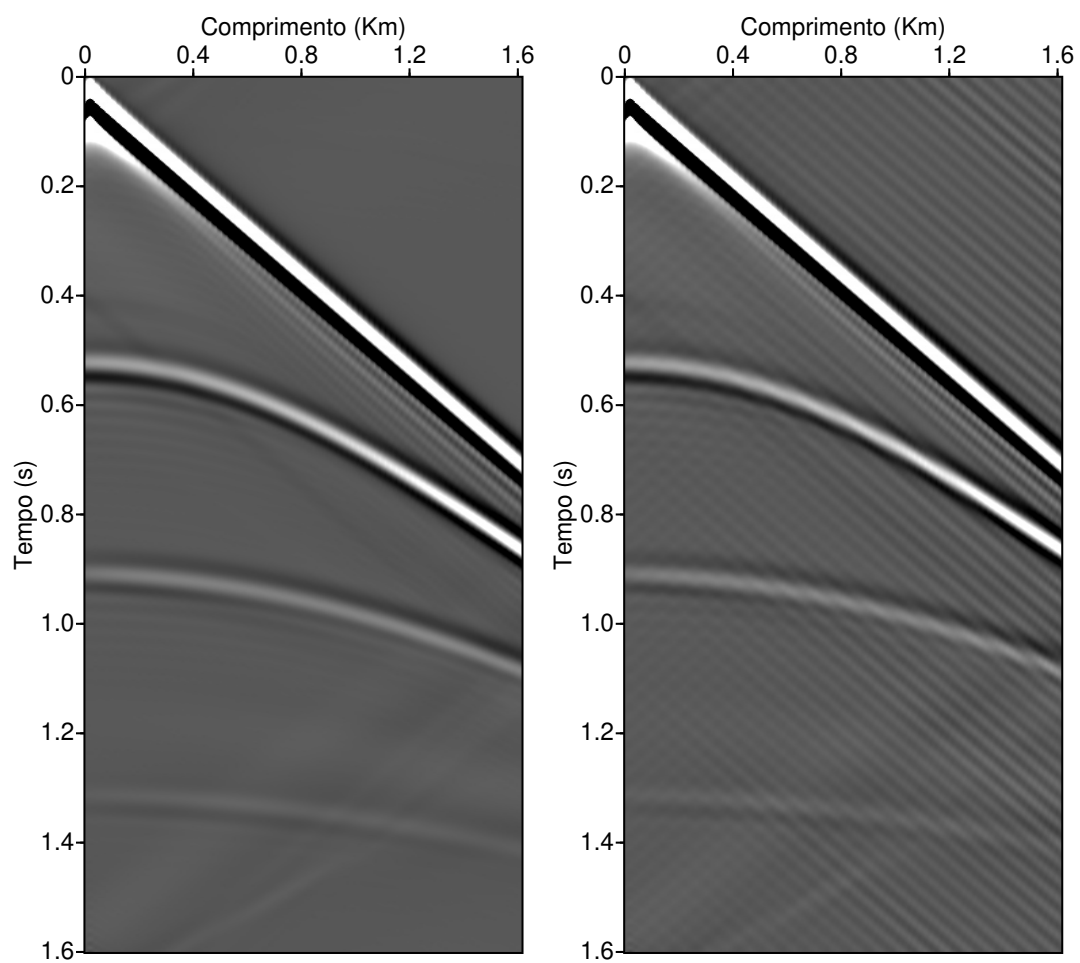


Figura 4.14: (a) *Shot* modelado referencia, (b) *shot* modelado usando $L=121$, PLS

5

Conclusões

A implementação do algoritmo originalmente proposta tem a vantagem de poder ser aplicada para problemas extremadamente grandes, os quais não podem ser resolvidos com a capacidade computacional instalada em um local específico e portanto pode fazer uso de sistemas de computação distribuído pela *web*, como por exemplo computação em nuvem, ou múltiplos centros de HPC, mantendo a privacidade do resultado devido que apenas o nó mestre (local) calcula a resposta final. Em contraste, quando o problema pode ser resolvido com a capacidade de computação instalada, na medida que aumenta a ordem da solução vão ficando livres nós computacionais, o que reduz a eficiência.

Quando o problema pode ser resolvido localmente, a implementação proposta do algoritmo de recursão de Levinson para sistemas de equações normais densos Hermitianas em bloco apresenta boa escalabilidade, embora o número de operações incremente quando o nível de partição L é aumentado. A complexidade computacional permanece na mesma ordem de grandeza ($O(n) = n^3$) e o erro no modelo estimado não apresenta variações significativas. Essa vantagem ocorre principalmente na versão OUT-OF-CORE usando GPU, quando o número de RHS é significativamente maior que o tamanho do sistema linear.

O algoritmo para obtenção da solução (slv_{BH}) é mais eficiente quando o sistema linear tem múltiplos RHS, o que para NRHS grandes compensa o menor desempenho da primeira parte (Fac_{BH}). Dependendo do tamanho do problema, existe algum valor de NRHS a partir do qual o desempenho geral é ligeiramente superior ao algoritmo de referência (Fatoração LU).

A utilização deste algoritmo na modelagem de ondas sísmicas no domínio da frequência, que envolve a solução de grandes sistemas lineares esparsos (não Toeplitz) não é recomen-

dados, pois a cada recursão, onde são calculadas a solução direta (F) de ordem j e a solução reversa (B) de ordem j , aumenta o número de elementos diferentes de zero, o que rapidamente converte essas matrizes em matrizes completamente densas. Isso aumenta o custo computacional e o consumo de memória.

Agradecimentos

Agradeço ao Governo do Brasil por considerar a educação como um direito para todos e um dever do Estado.

À Universidade Federal da Bahia (UFBA) por proporcionar ensino público, gratuito e de qualidade e por todas as oportunidades de aprendizado.

Ao meu orientador, Dr. Milton Porsani, por acreditar em mim, por sua valiosa ajuda, pela incrível paciência e me oferecer todo o apoio durante esses anos.

Ao Programa de Pós-Graduação em Geofísica pelas oportunidades e confiança. A todos os professores desta pós-graduação pelo conhecimento oferecido e orientação acadêmica. Aos funcionários, em especial Joaquim Lago e Júlio Leão.

Ao Centro de Pesquisa em Geofísica e Geologia (CPGG/UFBA) pela infraestrutura disponibilizada para que fosse possível a realização deste trabalho, que permite concretizar mais um importante passo na minha vida acadêmica.

À Coordenação de aperfeiçoamento de pessoal de nível superior (CAPES) pelo suporte financeiro através da bolsa de estudo outorgada para a realização do curso de Doutorado.

Ao o SENAI CIMATEC pelo suporte tecnológico através da possibilidade de usar o supercomputador OGUN.

À os membros da banca examinadora pelas valiosas correções que permitiram melhorar este trabalho.

À meus pais Anabeiba e Jesus por terem me presenteado com a vida e me darem o melhor de vocês. A meus irmãos pelo amor incondicional, apoio e parceria.

À Rocio por seu amor, por ser minha parceira em todos os aspectos da minha vida e minha motivação nos momentos difíceis. Obrigado pela cumplicidade, apoio e paciência. Excelente convivência durante a pandemia. Você é uma pessoa maravilhosa.

Aos colegas da pós-graduação (brasileiros, colombianos e venezuelanos) pela boa convivência e companheirismo.

Agradecimentos especiais a meus novos amigos de Brasil, México, Venezuela, República Dominicana e Colômbia, por ser um grupo de pessoas fantásticas, as quais se converteram na minha família no Brasil.

Gratidão a todas as pessoas que contribuíram para que este trabalho tenha sido um sucesso.

A todos ***Gracias Totales!***

Apêndice **A**

Fatoração de Cholesky por blocos

Tendo o seguinte sistema linear a resolver:

$$\mathbf{A}\mathbf{X} = \mathbf{B} \quad (\text{A.1})$$

Sendo \mathbf{A} uma matriz simétrica positiva definida, ela pode ser escrita como:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T \quad (\text{A.2})$$

onde $\mathbf{A} \in \mathbb{R}^{n \times n}$ e onde \mathbf{A} é simétrica positiva definida, Conseqüentemente, existe uma única matriz triangular inferior $\mathbf{L} \in \mathbb{R}^{n \times n}$ com entradas diagonais positivas tal que $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ (Golub e Van Loan, 2013).

Portanto a Eq. A.1 pode ser rescrita como:

$$\mathbf{L}\mathbf{L}^T\mathbf{X} = \mathbf{B} \quad (\text{A.3})$$

se definimos $\mathbf{Y} = \mathbf{L}^T\mathbf{X}$

Neste anexo apresentamos o pseudocódigo da decomposição Cholesky por blocos que é usada pela biblioteca LAPACK.

Para exemplificar, usaremos uma matriz por blocos 3×3 , portanto a Eq. A.2 fica:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{21}^T & \mathbf{A}_{31}^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{32}^T \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & 0 & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} & 0 \\ \mathbf{L}_{31} & \mathbf{L}_{32} & \mathbf{L}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T & \mathbf{L}_{31}^T \\ 0 & \mathbf{L}_{22}^T & \mathbf{L}_{32}^T \\ 0 & 0 & \mathbf{L}_{33}^T \end{bmatrix} \quad (\text{A.4})$$

onde $\mathbf{A}_{11}, \mathbf{A}_{21}, \mathbf{A}_{22} \in \mathbb{R}^{r \times r}$; $\mathbf{A}_{31}, \mathbf{A}_{32} \in \mathbb{R}^{n-r \times r}$ e $\mathbf{A}_{33} \in \mathbb{R}^{n-r \times n-r}$, onde r é um parâmetro de tamanho de bloco. Da Eq. A.4 podemos concluir que:

$$\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{L}_{11}^T, \quad (\text{A.5a})$$

$$\begin{bmatrix} \mathbf{A}_{21} \\ \mathbf{A}_{31} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{21} \\ \mathbf{L}_{31} \end{bmatrix} \mathbf{L}_{11}^T, \quad (\text{A.5b})$$

$$\mathbf{A}_{22} = [\mathbf{L}_{21} \quad \mathbf{L}_{22}] \begin{bmatrix} \mathbf{L}_{21}^T \\ \mathbf{L}_{22}^T \end{bmatrix}, \quad (\text{A.5c})$$

$$\mathbf{A}_{32} = [\mathbf{L}_{31} \quad \mathbf{L}_{32}] \begin{bmatrix} \mathbf{L}_{21}^T \\ \mathbf{L}_{22}^T \end{bmatrix}, \quad (\text{A.5d})$$

$$\mathbf{A}_{33} = [\mathbf{L}_{31} \quad \mathbf{L}_{32} \quad \mathbf{L}_{33}] \begin{bmatrix} \mathbf{L}_{31}^T \\ \mathbf{L}_{32}^T \\ \mathbf{L}_{33}^T \end{bmatrix}, \quad (\text{A.5e})$$

A Eq. A.5a implica a fatoração de Cholesky de \mathbf{A}_{11} para obter \mathbf{L}_{11} , já a Eq. A.5b consiste em resolver um sistema triangular inferior (\mathbf{L}_{11}^T) do lado direito com múltiplos RHS para obter $\begin{bmatrix} \mathbf{L}_{21} \\ \mathbf{L}_{31} \end{bmatrix}$. A seguir podemos definir e calcular as variáveis $\tilde{\mathbf{A}}_{22} = \mathbf{L}_{22}\mathbf{L}_{22}^T$, $\tilde{\mathbf{A}}_{32} = \mathbf{L}_{32}\mathbf{L}_{22}^T$ e $\tilde{\mathbf{A}}_{33} = \mathbf{L}_{33}\mathbf{L}_{33}^T$, para posteriormente calcular sua fatoração de Cholesky para obter \mathbf{L}_{22} , \mathbf{L}_{32} e \mathbf{L}_{33} como se observa a seguir:

$$\tilde{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T \quad \Rightarrow \quad \tilde{\mathbf{A}}_{22} = \mathbf{L}_{22}\mathbf{L}_{22}^T, \quad (\text{A.6a})$$

$$\tilde{\mathbf{A}}_{32} = \mathbf{A}_{32} - \mathbf{L}_{31}\mathbf{L}_{21}^T \quad \Rightarrow \quad \tilde{\mathbf{A}}_{32} = \mathbf{L}_{32}\mathbf{L}_{22}^T, \quad (\text{A.6b})$$

$$\tilde{\mathbf{A}}_{33} = \mathbf{A}_{33} - [\mathbf{L}_{31} \quad \mathbf{L}_{32}] \begin{bmatrix} \mathbf{L}_{31}^T \\ \mathbf{L}_{32}^T \end{bmatrix} \Rightarrow \tilde{\mathbf{A}}_{33} = \mathbf{L}_{33}\mathbf{L}_{33}^T, \quad (\text{A.6c})$$

Deve-se notar que os resultados das multiplicações $\mathbf{L}_{21}\mathbf{L}_{21}^T$ e $[\mathbf{L}_{31} \quad \mathbf{L}_{32}] \begin{bmatrix} \mathbf{L}_{31}^T \\ \mathbf{L}_{32}^T \end{bmatrix}$ assim como \mathbf{A}_{jj} é uma matriz simétrica e conseqüentemente as variáveis $\tilde{\mathbf{A}}_{jj}$ são simétricas. No algoritmo 7 apresentamos o pseudocódigo da fatoração de Cholesky por blocos para uma matriz de $P \times P$ blocos.

As sub-rotinas de LAPACK: XTRSM ($\mathbf{X}\mathbf{A}^T = \alpha\mathbf{B}$) e XSYRK ($\mathbf{C} = \beta\mathbf{C} + \alpha\mathbf{A}\mathbf{A}^T$) onde \mathbf{X} pode ser:

- **S**(real, precisão simples simples)
- **D**(real, precisão dupla)
- **C**(complexo, precisão simples)
- **Z**(complexo, precisão dupla)

Algoritmo 7 Algoritmo para fatoração de Cholesky por blocos**Require:** $\mathbf{A}(n, n)$: Sistema simétrico positivo definido de NE; P , tal que $(P - 1)r < n \leq Pr$.A parte triangular inferior de \mathbf{A} é substituída pela parte triangular inferior de \mathbf{A}

```

1: if  $r \geq n$  then
2:    $\mathbf{L} \leftarrow LL^T_{(\mathbf{A})}$  ▷ fatoração de Cholesky recursiva
3: else
4:   for  $j = 1, P$  do
5:      $\tilde{\mathbf{A}}_{jj} = \mathbf{A}_{jj} - \sum_{k=1}^{j-1} \mathbf{L}_{j,k} \mathbf{L}_{j,k}^T$  ▷ XSYRK
6:      $\mathbf{L}_{jj} \leftarrow LL^T_{(\tilde{\mathbf{A}}_{jj})}$  ▷ fatoração de Cholesky recursiva
7:     if  $(j < P)$  then
8:       
$$\begin{bmatrix} \tilde{\mathbf{A}}_{j+1,j} \\ \vdots \\ \tilde{\mathbf{A}}_{P,j} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{j+1,j} \\ \vdots \\ \mathbf{A}_{P,j} \end{bmatrix} - \begin{bmatrix} \mathbf{L}_{j+1,1} & \cdots & \mathbf{L}_{j+1,j-1} \\ \vdots & \ddots & \vdots \\ \mathbf{L}_{P,1} & \cdots & \mathbf{L}_{P,j-1} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{j,1}^T \\ \vdots \\ \mathbf{L}_{j,j-1}^T \end{bmatrix}$$

9:       
$$\begin{bmatrix} \mathbf{L}_{j+1,j} \\ \vdots \\ \mathbf{L}_{P,j} \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{L}_{j+1,j} \\ \vdots \\ \mathbf{L}_{P,j} \end{bmatrix} \mathbf{L}_{jj}^T = \begin{bmatrix} \tilde{\mathbf{A}}_{j+1,j} \\ \vdots \\ \tilde{\mathbf{A}}_{P,j} \end{bmatrix}$$
 ▷ XTRSM
10:    end if
11:  end for
12: end if
13: return  $\mathbf{L}$ 

```

elas resolvem um SL triangular e faz a atualização da multiplicação de matriz simétrica respectivamente.

Para o cálculo da fatoração Cholesky, é usada a sub-rutina XPOTRF2 de LAPACK que usa o algoritmo recursivo proposto por Gustavson (1997), esta versão permite obter uma maior eficiência por conta do bloqueio automático de variáveis que está implícito em seu uso. No artigo de Gustavson são apresentados detalhadamente os benefícios do algoritmo bloco recursivo.

A seguir apresentamos em que consiste o algoritmo recursivo para obter a fatoração Cholesky. Inicialmente, a matriz \mathbf{A} é dividida em 4 blocos:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{21}^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T \\ 0 & \mathbf{L}_{22}^T \end{bmatrix} \quad (\text{A.7})$$

onde $\mathbf{A}_{11} \in \mathbb{R}^{r \times r}$; $\mathbf{A}_{21} \in \mathbb{R}^{n-r \times r}$ e $\mathbf{A}_{22} \in \mathbb{R}^{n-r \times n-r}$, onde $r = \text{ceil}(n/2)$. Da Eq. A.7

podemos concluir que:

$$\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{L}_{11}^T, \quad (\text{A.8a})$$

$$\mathbf{A}_{21} = \mathbf{L}_{21}\mathbf{L}_{11}^T, \quad (\text{A.8b})$$

$$\mathbf{A}_{22} = \mathbf{L}_{21}\mathbf{L}_{21}^T + \mathbf{L}_{22}\mathbf{L}_{22}^T \Rightarrow \tilde{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T, \quad (\text{A.8c})$$

onde $\tilde{\mathbf{A}}_{22} = \mathbf{L}_{22}\mathbf{L}_{22}^T$, sugerindo o seguinte procedimento de 4 etapas:

- Calcule a fatoração de Cholesky (metade do tamanho) de \mathbf{A}_{11} para obter \mathbf{L}_{11}
- Resolva um sistema triangular inferior direito com NRHS para obter \mathbf{L}_{21}
- Atualize a matriz $\tilde{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T$
- Calcule a fatoração de Cholesky de $\tilde{\mathbf{A}}_{22}$ para obter \mathbf{L}_{22}

Na forma recursiva, obtemos o seguinte algoritmo (8):

Algoritmo 8 Recursivo Cholesky(\mathbf{A}, n)

Require: $\mathbf{A}(n, n)$: Sistema simétrico positivo definido de NE;

A parte triangular inferior de \mathbf{A} é substituída pela parte triangular inferior de \mathbf{L}

```

1: if  $n = 1$  then
2:    $\mathbf{A}_{11} = \sqrt{A_{11}}$ 
3: else
4:    $n_1 = n/2; n_2 = n - n_1$ 
5:    $\mathbf{L}_{11} \leftarrow$  Recursivo Cholesky( $\mathbf{A}_{11}, n_1$ )
6:    $\mathbf{L}_{21} \leftarrow \mathbf{L}_{21}\mathbf{L}_{11}^T = \mathbf{A}_{21}$  ▷ XTRSM
7:    $\tilde{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T$  ▷ XSYRK
8:    $\mathbf{L}_{22} \leftarrow$  Recursivo Cholesky( $\tilde{\mathbf{A}}_{22}, n_2$ )
9: end if
10: return  $\mathbf{L}$ 

```

Apêndice **B**

Cholesky vs Levinson

A seguir apresentamos a relação entre a fatoração de Cholesky por blocos e o método recursivo de Levinson, para um sistema de blocos 2×2 .

B.1 Sistema linear por blocos 2×2

Tendo o seguinte sistema linear a resolver:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{21}^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{X}_{2,1} \\ \mathbf{X}_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \end{bmatrix} \quad (\text{B.1})$$

B.1.1 Fatoração de Cholesky

Lembrando a Eq A.7, temos que \mathbf{A} pode ser escrito como:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{21}^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T \\ 0 & \mathbf{L}_{22}^T \end{bmatrix} \quad (\text{B.2})$$

temos que :

$$\mathbf{L}_{11}\mathbf{L}_{11}^T = \mathbf{A}_{11}, \quad (\text{B.3a})$$

$$\mathbf{L}_{21}\mathbf{L}_{11}^T = \mathbf{A}_{21}, \quad (\text{B.3b})$$

$$\tilde{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T, \quad (\text{B.3c})$$

$$\mathbf{L}_{22}\mathbf{L}_{22}^T = \tilde{\mathbf{A}}_{22}, \quad (\text{B.3d})$$

Para obter a solução \mathbf{X} temos primeiro resolver o sistema $\mathbf{L}\mathbf{Y} = \mathbf{B}$ e logo $\mathbf{L}^T\mathbf{X} = \mathbf{Y}$

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \end{bmatrix} \quad (\text{B.4})$$

temos que:

$$\mathbf{L}_{11} \mathbf{Y}_1 = \mathbf{B}_1, \quad (\text{B.5a})$$

$$\mathbf{L}_{22} \mathbf{Y}_2 = \mathbf{B}_2 - \mathbf{L}_{21} \mathbf{Y}_1, \quad (\text{B.5b})$$

das quais podemos obter \mathbf{Y}_1 e \mathbf{Y}_2 através da solução de sistemas triangulares, para logo:

$$\begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T \\ \mathbf{0} & \mathbf{L}_{22}^T \end{bmatrix} \begin{bmatrix} \mathbf{X}_{2,1} \\ \mathbf{X}_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} \quad (\text{B.6})$$

do qual podemos concluir que:

$$\mathbf{L}_{22}^T \mathbf{X}_{2,2} = \mathbf{Y}_2, \quad (\text{B.7a})$$

$$\mathbf{L}_{11}^T \mathbf{X}_{2,1} = \mathbf{Y}_1 - \mathbf{L}_{21}^T \mathbf{X}_{2,2}, \quad (\text{B.7b})$$

B.1.2 Recursão de Levinson

Reescrevendo o apresentado na seção 2.2.1.1

$$\begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix} = \begin{bmatrix} -\mathbf{B}_1 & \mathbf{A}_{11} & \mathbf{A}_{12} \\ -\mathbf{B}_2 & \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{nr} \\ \mathbf{X}_{2,1} \\ \mathbf{X}_{2,2} \end{bmatrix}, \quad (\text{B.8})$$

e usando as Eqs. 2.15 e 2.19:

$$\mathbf{A}_{11} {}^1\mathbf{F}_{11} = -\mathbf{A}_{12}, \quad (\text{B.9a})$$

$${}^1\mathbf{E}_{F1} = \mathbf{A}_{22} + \mathbf{A}_{21} {}^1\mathbf{F}_{11}, \quad (\text{B.9b})$$

e para obter a solução:

$$\mathbf{A}_{11} \mathbf{X}_{1,1} = \mathbf{B}_1, \quad (\text{B.10a})$$

$$\Delta_{X1} = -\mathbf{B}_2 + \mathbf{A}_{21} \mathbf{X}_{1,1}, \quad (\text{B.10b})$$

$${}^1\mathbf{E}_{F1} \mathbf{X}_{2,2} = -\Delta_{X1}, \quad (\text{B.10c})$$

$$\mathbf{X}_{2,1} = \mathbf{X}_{1,1} + {}^1\mathbf{F}_{11} \mathbf{X}_{2,2}, \quad (\text{B.10d})$$

B.1.3 Relação entre o procedimento Cholesky e Levinson

No início é fatorada \mathbf{A}_{11} da Eq. B.9a e multiplicada pela esquerda com \mathbf{L}_{11}^{-1} para simplificar $\mathbf{L}_{11}^{-1}\mathbf{L}_{11} = \mathbf{I}$; Após é inserida a Eq. B.3b e multiplicando com $(\mathbf{L}_{11}^T)^{-1}$ como se observa:

$$\begin{aligned}
 \mathbf{A}_{11} {}^1\mathbf{F}_{11} &= -\mathbf{A}_{21}^T \\
 \mathbf{L}_{11}\mathbf{L}_{11}^T {}^1\mathbf{F}_{11} &= -\mathbf{A}_{21}^T \\
 \mathbf{L}_{11}^{-1}\mathbf{L}_{11}\mathbf{L}_{11}^T {}^1\mathbf{F}_{11} &= -\mathbf{L}_{11}^{-1}\mathbf{A}_{21}^T \\
 (\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{11}^T {}^1\mathbf{F}_{11} &= -(\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{11}^{-1}\mathbf{L}_{11}\mathbf{L}_{21}^T \\
 \boxed{{}^1\mathbf{F}_{11} = -(\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{21}^T}
 \end{aligned} \tag{B.11}$$

Logo inserindo a B.3b e Eq. B.11 na Eq. B.9b temos que:

$$\begin{aligned}
 {}^1\mathbf{E}_{F1} &= \mathbf{A}_{22} + \mathbf{L}_{21}\mathbf{L}_{11}^T(\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{21}^T \\
 {}^1\mathbf{E}_{F1} &= \mathbf{A}_{22} + \mathbf{L}_{21}\mathbf{L}_{21}^T = \tilde{\mathbf{A}}_{22} = \mathbf{L}_{22}\mathbf{L}_{22}^T \\
 \boxed{{}^1\mathbf{E}_{F1} = \mathbf{L}_{22}\mathbf{L}_{22}^T}
 \end{aligned} \tag{B.12}$$

Para obter a solução do sistema \mathbf{X} , comparando a Eq. B.10a com B.5a temos que:

$$\begin{aligned}
 \mathbf{A}_{11}\mathbf{X}_{1,1} &= \mathbf{B}_1 \\
 \mathbf{L}_{11}\mathbf{L}_{11}^T\mathbf{X}_{1,1} &= \mathbf{B}_1 \\
 \mathbf{L}_{11}^T\mathbf{X}_{1,1} &= \mathbf{Y}_1 \\
 \boxed{\mathbf{X}_{1,1} = (\mathbf{L}_{11}^T)^{-1}\mathbf{Y}_1}
 \end{aligned} \tag{B.13}$$

A matriz Δ_{X1} é então (inserindo B.3b e B.13 e comparando com B.5b):

$$\begin{aligned}
 \Delta_{X1} &= -\mathbf{B}_2 + \mathbf{L}_{21}\mathbf{L}_{11}^T(\mathbf{L}_{11}^T)^{-1}\mathbf{Y}_1 \\
 \Delta_{X1} &= -\mathbf{B}_2 + \mathbf{L}_{21}\mathbf{Y}_1 = -\mathbf{L}_{22}\mathbf{Y}_2 \\
 \boxed{\Delta_{X1} = -\mathbf{L}_{22}\mathbf{Y}_2}
 \end{aligned} \tag{B.14}$$

sendo evidente que $\mathbf{L}_{22}^T\mathbf{X}_{2,2} = \mathbf{Y}_2$ e ${}^1\mathbf{E}_{F1}\mathbf{X}_{2,2} = -\Delta_{X1}$ são equivalentes. Similarmente, a Eq B.7b e a Eq B.10d são equivalentes:

$$\begin{aligned}
 \mathbf{X}_{2,1} &= \mathbf{X}_{1,1} + {}^1\mathbf{F}_{11}\mathbf{X}_{2,2} \\
 \boxed{\mathbf{X}_{2,1} = (\mathbf{L}_{11}^T)^{-1}\mathbf{Y}_1 - (\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{21}^T\mathbf{X}_{2,2}} \\
 \mathbf{L}_{11}^T\mathbf{X}_{2,1} &= \mathbf{Y}_1 - \mathbf{L}_{21}^T\mathbf{X}_{2,2}
 \end{aligned} \tag{B.15}$$

Apêndice **C**

Complexidade

O contagem de FLOPs (operações de ponto flutuante) das operações básicas usadas para a solução de sistemas simétricos, positivos definidos e densos é apresentado na tabela C.1 quando é usado números reais precisão dupla:

Operação	Multiplicações	Adições	Totais
$\mathbf{C} = \mathbf{C} + \mathbf{GB}$	mnk	mnk	$2mnk$
$\mathbf{S} = \mathbf{S} + \mathbf{EF}$	$kn(n+1)/2$	$kn(n+1)/2$	$kn(n+1)$
$LL_{\mathbf{A}_{n \times n}}^T$	$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n$	$\frac{1}{6}n^3 - \frac{1}{6}n$	$\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$
$\mathbf{M} \leftarrow LL_{\mathbf{A}_{n \times n}}^T \mathbf{M} = \mathbf{B}$	$NRHS(n^2 + n)$	$NRHS(n^2 - n)$	$2NRHS(n^2)$

Tabela C.1: Custo computacional das operações matriciais básicas, onde $\mathbf{C}^{m \times n}$, $\mathbf{G}^{m \times k}$, $\mathbf{B}^{k \times n}$, $\mathbf{S}^{n \times n}$ (simétrica), $\mathbf{E}^{n \times k}$, $\mathbf{F}^{k \times n}$ (onde o produto \mathbf{EF} é simétrico), $\mathbf{M}_{n \times NRHS}$ e $\mathbf{B}_{n \times NRHS}$, todas elas $\in \mathbf{R}$.

Tendo em conta a tabela C.1, o contagem de FLOPs para o algoritmo 3, no caso de uma matriz Simétrica positiva definida ($\in \mathbf{R}$), é apresentado na tabela C.2.

linha	uma vez	loop p (loop i)
10: ${}^i\Delta_{Hj-1}$	$(j-1)2N_c^3$	$\frac{1}{6}L(L-1)(L-2)2N_c^3$
13: ${}^i\mathbf{H}_{jj}$	S_m	$\frac{1}{2}(L-1)(L-2)S_m$
14: ${}^i\mathbf{E}_{Hj}$	$N_c^3 + N_c^2$	$\frac{1}{2}(L-1)(L-2)(N_c^3 + N_c^2)$
16: ${}^i\mathbf{F}_{jj}$	S_m	$\frac{1}{2}L(L-1)S_m$
17: ${}^i\mathbf{E}_{Fj}$	$N_c^3 + N_c^2$	$\frac{1}{2}L(L-1)(N_c^3 + N_c^2)$
18: ${}^i\mathcal{F}_j$	$(j-1)2N_c^3$	$\frac{1}{6}L(L-1)(L-2)2N_c^3$
20: ${}^i\mathcal{H}_j$	$(j-1)2N_c^3$	$\frac{1}{6}(L-1)(L-2)(L-3)2N_c^3$
25: $LL^*({}^1\mathbf{E}_{Fj})$	$\frac{1}{6}N_c(N_c+1)(2N_c+1)$	$\frac{1}{6}LN_c(N_c+1)(2N_c+1)$
total		$N_c^3(L^3 - 3L^2 + \frac{10}{3}L - 1) + N_c^2(L^2 - \frac{3}{2}L + 1) + \frac{1}{6}LN_c + S_m(L^2 - 2L + 1)$

Tabela C.2: Custo computacional detalhado do algoritmo para números reais, sub-rotina 1 (Algoritmo 3) tipo Levinson para solução de sistemas NE. S_m representa o custo para resolver uma família de N_c sistemas associados à mesma matriz de coeficientes.

Se para o cálculo de S_m , usamos a fatoração de Cholesky, e tendo que para a decomposição os FLOPs são $\frac{1}{6}N_c(N_c+1)(2N_c+1)$ e a solução de 2 sistemas triangulares é $2N_c^2NRHS$ onde $NRHS = N_c$ (Golub e Van Loan, 2013; Blackford e Dongarra, 1999), temos que:

$$\begin{aligned}
 S_m &= \frac{1}{6}N_c(N_c+1)(2N_c+1) + 2N_c^3 \\
 S_m &= \frac{7}{3}N_c^3 + \frac{1}{2}N_c^2 + \frac{1}{6}N_c
 \end{aligned} \tag{C.1}$$

portanto, os FLOPs totais:

$$FLOP_{slv} = \frac{1}{3}N_c^3(3L^3 - 2L^2 - 4L + 4) + \frac{1}{2}N_c^2(3L^2 - 5L + 3) + \frac{1}{6}N_c(L^2 - L + 1) \tag{C.2}$$

tendo para os casos limite: sem particionamento ($L = 1$), o valor corresponde a fatoração de Cholesky de tamanho n , o seja $\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$, já quando $N_c = 1$ e $L = n$, (onde cada “bloco” é de tamanho 1), temos que:

$$FLOP_{slv \Rightarrow L=n} = n^3 + n^2 - 4n + 3 \tag{C.3}$$

Para quando L é alterado, podemos separar o custo para a multiplicação matriz-matriz (e soma) e o custo para resolver subsistemas, portanto a Eq. C.2, é a soma entre os FLOPs do produto entre matrizes:

$$FLOP_{slv=mm} = N_c^3(L^3 - 3L^2 + 3L - 1) + N_c^2(L^2 - 2L + 1), \tag{C.4}$$

e os FLOPs da solução de subsistemas:

$$FLOP_{slv=slv} = \frac{1}{3}N_c^3 (7L^2 - 13L + 7) + \frac{1}{2}N_c^2 (L^2 - L + 1) + \frac{1}{6}N_c (L^2 - L + 1). \quad (C.5)$$

Na figura C.1 é apresentado os FLOPs (normalizados respeito a $n^3 + n^2 - 4n + 3$) para $N = 50000$, onde é possível observar, al igual que para o caso de números complexos, que quando o particionamento é aumentado o custo computacional aumenta para o valor máximo que corresponde ao custo de $L = N$, $N_c = 1$. Também é possível mostrar como à medida que L aumenta, o custo computacional para resolver subsistemas lineares ($FLOP_{slv=slv}$) diminui e o custo das operações entre matrizes ($FLOP_{slv=mm}$) aumenta (multiplicações entre matrizes e soma de matrizes).

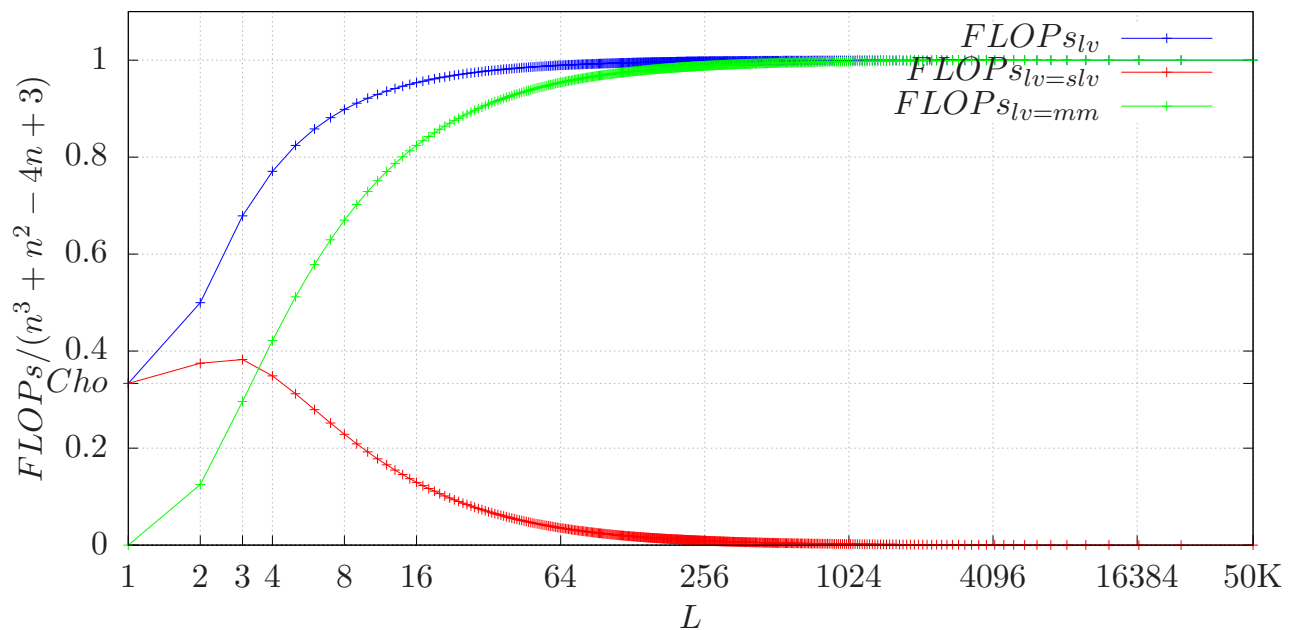


Figura C.1: Operações de ponto flutuante (FLOP) normalizado para decompor (Algoritmo 3) um sistema $N \times N$ ($N = 50000$) em função do particionamento L .

Para a obtenção da solução (algoritmo 4), os FLOPs são apresentados na tabela C.3.

linha	uma vez	loop j
1: \mathbf{M}_{11}	$2N_c^2$	$2N_c^2$
3: Δ_{Mj}	$j2N_c^2$	$L(L-1)N_c^2$
4: $\mathbf{M}_{j+1,j+1}$	$2N_c^2$	$(L-1)(2N_c^2)$
5: \mathcal{M}_{j+1}	$j2N_c^2$	$L(L-1)N_c^2$
total		$2L^2N_c^2$

Tabela C.3: Custo computacional detalhado do algoritmo, sub-rotina 2 (Algoritmo 4) tipo de Levinson para solução de sistemas NE para apenas um RHS, para múltiplos RHS multiplique por NRHS.

Tendo que o tamanho de sistema pode ser entre $(L - 1)N_c < n \leq LN_c$; simplificando, suponha que $n = LN_c$, nesse caso o total de FLOP para resolver (sub-rotina 2) um RHS é $2n^2$, isso é igual que a solução dos dois sistemas triangulares (substituição direta e reversa) na decomposição de Cholesky (Golub e Van Loan, 2013; Blackford e Dongarra, 1999). Em compensação, as operações de multiplicação matriz-matriz (soma das linhas 3 e 5):

$$FLOPs - S2_{lv=mm} = 2L^2N_c^2 - 2LN_c^2, \quad (C.6)$$

são maiores que a substituição direta e reversa (soma das linhas 1 e 4):

$$FLOPs - S2_{lv=slv} = 2LN_c^2, \quad (C.7)$$

estas últimas diminuindo, a medida que L aumenta, como se apresenta na figura C.2.

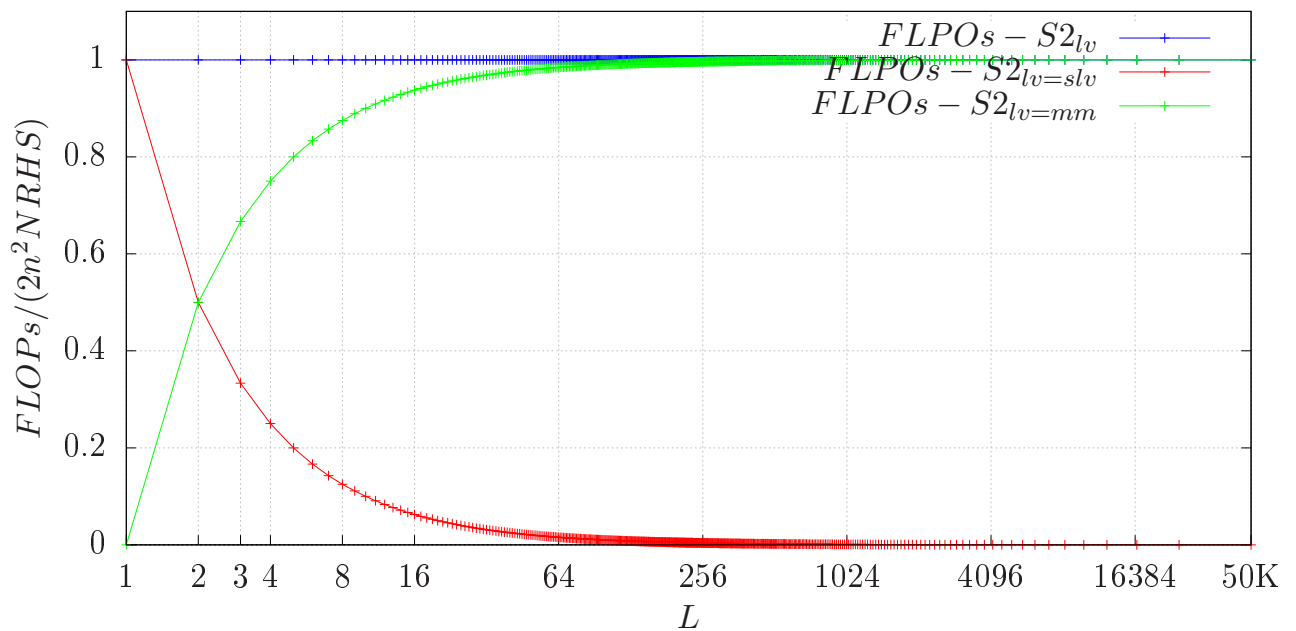


Figura C.2: Operações de ponto flutuante (FLOP) normalizado para resolver (Algoritmo 4) um sistema de N ($N = 50000$) parâmetros e $NRHS$ lados direitos em função do particionamento L .

No caso de um SL Hermitiano, onde as operações matriciais são com números complexos, temos que ter em mente que cada multiplicação contaria como 6 operações (4 multiplicações e 2 adições) e cada adição como 2 operações (2 adições), portanto, a contagem de operações das operações básicas é apresentado na Tabela C.4, e os FLOPs dos algoritmos 3 e 4 no caso de números complexos já foram apresentados nas tabelas 2.1 e 2.2.

Operação	Multiplicações	Adições	Totais
$\mathbf{C} = \mathbf{C} + \mathbf{GB}$	$4mnk$	$4mnk$	$8mnk$
$\mathbf{S} = \mathbf{S} + \mathbf{EF}$	$2kn(n+1)$	$2kn(n+1)$	$4kn(n+1)$
$LL_{\mathbf{A}_{n \times n}}^*$	$\frac{2}{3}n^3 + 2n^2 + \frac{4}{3}n$	$\frac{2}{3}n^3 + n^2 - \frac{1}{3}n$	$\frac{4}{3}n^3 + 3n^2 + \frac{5}{3}n$
$\mathbf{M} \leftarrow LL_{\mathbf{A}_{n \times n}}^T \mathbf{M} = \mathbf{B}$	$4NRHS(n^2 + n)$	$4NRHS(n^2)$	$4NRHS(2n^2 + n)$

Tabela C.4: Custo computacional das operações matriciais básicas, onde $\mathbf{C}^{m \times n}$, $\mathbf{G}^{m \times k}$, $\mathbf{B}^{k \times n}$, $\mathbf{S}^{n \times n}$ (Hermitiana), $\mathbf{E}^{n \times k}$, $\mathbf{F}^{k \times n}$ (onde o produto \mathbf{EF} é hermitiano), $\mathbf{M}_{n \times NRHS}$ e $\mathbf{B}_{n \times NRHS}$, todas elas $\in \mathbf{C}$.

Apêndice **D**

Algoritmo para a solução por blocos Cholesky usando MAGMA, Out-Of-Core

No momento, o MAGMA não possui uma implementação Out-Of-Core do XPOTRS, por isso implementamos uma para fins de comparação.

D.1 Sistema linear por blocos $j \times j$

Generalizando, tendo o seguinte sistema linear de $j \times j$ blocos para resolver:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{21}^T & \cdots & \mathbf{A}_{j1}^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{j2}^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{j1} & \mathbf{A}_{j2} & \cdots & \mathbf{A}_{jj} \end{bmatrix} \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \vdots \\ \mathbf{X}_j \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_j \end{bmatrix} \quad (\text{D.1})$$

onde \mathbf{A} pode ser escrita como:

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{L}_{j1} & \mathbf{L}_{j2} & \cdots & \mathbf{L}_{jj} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T & \cdots & \mathbf{L}_{j1}^T \\ \mathbf{0} & \mathbf{L}_{22}^T & \cdots & \mathbf{L}_{j2}^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{L}_{jj}^T \end{bmatrix} \quad (\text{D.2})$$

O procedimento inicia da mesma forma em que foi expressado no apêndice B.1.1, obtendo a solução do sistema $\mathbf{L}_{11} \mathbf{Y}_1 = \mathbf{B}_1$

Algoritmo 9 Resolve um sistema linear $\mathbf{AX} = \mathbf{B}$ com uma matriz simétrica positiva definida \mathbf{A} usando a fatoração de Cholesky $\mathbf{A} = \mathbf{LL}^T$ calculada por XPOTRF.

Require: $\mathbf{A} = \mathbf{LL}^T(N, N)$:

$\mathbf{B}(N, NRHS)$

- 1: Seção: solução $\mathbf{LY} = \mathbf{B}$, sobrescrevendo \mathbf{B} com ($= \mathbf{Y}$) .
 - 2: **for** $j = 1, L$ **do**
 - 3: **for** $i = j - 1, 1, -1$ **do**
 - 4: $\mathbf{B}_j = \mathbf{B}_j - \mathbf{L}_{j,i} \mathbf{Y}_i$
 - 5: **end for**
 - 6: $\mathbf{Y}_j \leftarrow \mathbf{L}_{j,j} \mathbf{Y}_j = \mathbf{B}_j, :: \text{XTRSM}$
 - 7: **end for**
 - 8: Seção : solução $\mathbf{L}^T \mathbf{X} = \mathbf{Y}$, sobrescrevendo $\mathbf{B}(= \mathbf{Y})$ com \mathbf{X} .
 - 9: **for** $j = L, 1, -1$ **do**
 - 10: **for** $i = j + 1, L$ **do**
 - 11: $\mathbf{B}_j = \mathbf{B}_j - \mathbf{L}_{i,j}^* \mathbf{X}_i$
 - 12: **end for**
 - 13: $\mathbf{X}_j \leftarrow \mathbf{L}_{j,j}^* \mathbf{X}_j = \mathbf{Y}_j :: \text{XTRSM}$
 - 14: **end for**
 - 15: **return** \mathbf{X}
-

D.2 Algoritmo Out-of-Core com multiples RHS

Algoritmo 10 Algoritmo Out-of-Core usando GPU, resolve um sistema linear $\mathbf{AX} = \mathbf{B}$ com uma matriz simétrica positiva definida \mathbf{A} usando a fatoração de Cholesky $\mathbf{A} = \mathbf{LL}^T$ calculada por XPOTRF.

Require: $\mathbf{A} = \mathbf{LL}^T(N, N)$:

$\mathbf{B}(N, NRHS)$

Require: memória de trabalho na GPU: uma matriz (N_b, N_b) ; dois matrizes $(N_b, maxRHS)$
: dA, dB, dC ▷ No GPU

```

1: Cal Nb,L,maxRHS(N,NRHS,Memoria da GPU,bytes)
2: for jr = 1, nrhs, maxNRHS do
3:   jb = min( maxNRHS, nrhs-jr+1 )
4:   Seção : solução  $L^*X = B$ , sobrescrevendo B com  $X(==Y)$ .
5:   for j = 1, L do
6:     Set  $\mathbf{B}_j$  in dB
7:     for i = j - 1, 1, -1 do
8:       Set  $\mathbf{L}_{j,i}$  in dA
9:       if(i < j - 1) then: Set  $\mathbf{Y}_i$  in dC
10:       $\mathbf{B}_{j,jr} = \mathbf{B}_{j,jr} - \mathbf{L}_{j,i} \mathbf{Y}_i$ 
11:    end for
12:    Set  $\mathbf{L}_{j,j}$  in dA
13:     $\mathbf{Y}_j \leftarrow \mathbf{L}_{j,j} \mathbf{Y}_j = \mathbf{B}_j$ , :: XTRSM (...dA,...dB..)
14:    Get  $\mathbf{Y}_j$ 
15:  end for
16:  Seção : solução  $L^{*T} * X = B$ , sobrescrevendo B(==Y) with X.
17:  for j = L, 1, -1 do
18:    if(j < L) then: Set  $\mathbf{Y}_j$  in dB
19:    for i = j + 1, L do
20:      Set  $\mathbf{L}_{i,j}^*$  in dA
21:      if(i > j + 1) then: Set  $\mathbf{X}_i$  in dC
22:       $\mathbf{B}_j = \mathbf{B}_j - \mathbf{L}_{i,j}^* \mathbf{X}_i$ 
23:    end for
24:    Set  $\mathbf{L}_{j,j}$  in dA
25:     $\mathbf{X}_j \leftarrow \mathbf{L}_{j,j}^* \mathbf{X}_j = \mathbf{Y}_j$ , :: XTRSM (...dA,...dB..)
26:    Get  $\mathbf{X}_j$ 
27:  end for
28: end for
29: return X

```

No algoritmo 10 o cálculo de N_b (e portanto L) e RHS_{max} é realizado com base em 90% da memória da GPU, de forma que haja um equilíbrio entre N_b (de preferência valores grandes) e o RHS máximo que você pode resolver de uma só vez (de preferência valores grandes).

Apêndice **E**

Resultados adicionais: Diferentes tamanhos

E.1 Versão memória compartilhada

Nas gráficas E.1 e E.1 são apresentados resultados adicionais do efeito do particionamento para quando $NRHS = N$ e $NRHS = 3N$ respectivamente. Em todos se pode apreciar a mesma tendência: Fac_{BH} aumenta, slv_{BH} diminui e a soma deles Tot_{BH} tende a aumentar.

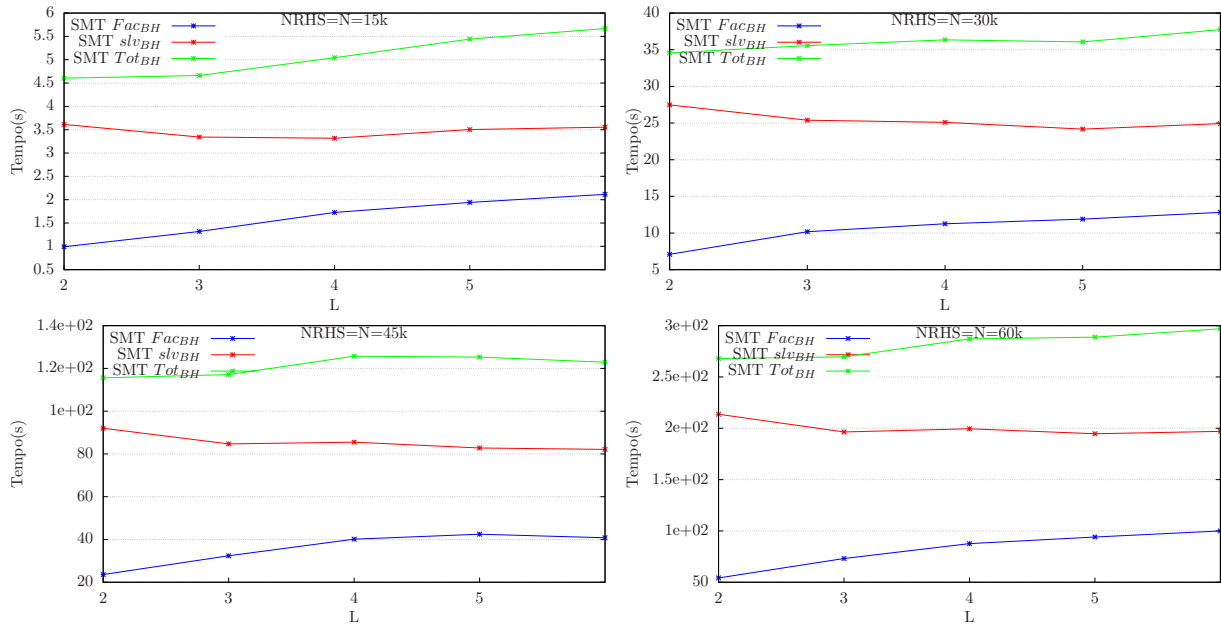


Figura E.1: Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória compartilhada vários tamanhos de N , em todos eles $NRHS = N$ e variando o particionamento, usando um CPU de 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

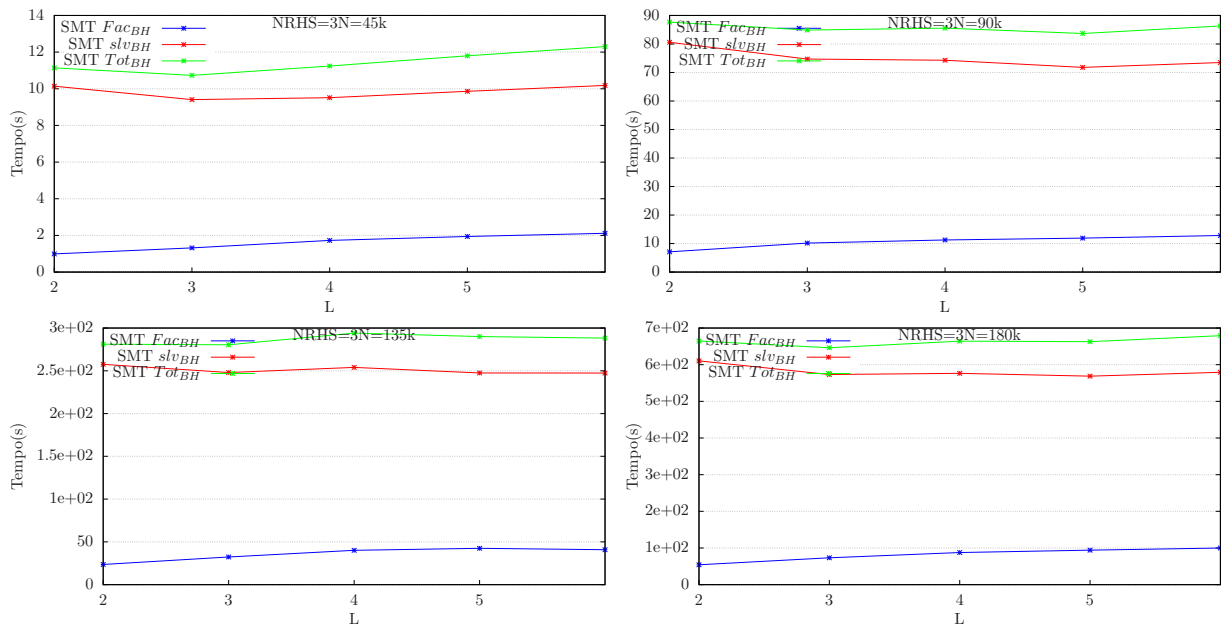


Figura E.2: Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória compartilhada vários tamanhos de N , em todos eles $NRHS = 3N$ e variando o particionamento ($L = 2, 3, 4, 5, 6$), usando um CPU de 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

Nas gráficas E.3 e E.4 são apresentados resultados adicionais quando é variado o valor de $NRHS$ ate $3N$ mantendo N constante. Em todos os casos se pode apreciar a mesma tendência: o menor desempenho da primeira parte é compensado com melhor desempenho na segunda parte, o que permite equiparar o algoritmo de referência com a implementação proposta. Para quando $NRHS$ aumenta, nosso algoritmo evidencia uma tendência a melhorar. Isto é confirmado nas figuras E.5, E.6 (onde $NRHS$ vai ate $11N$), e E.7 e E.8(onde $NRHS$ vai ate $20N$).

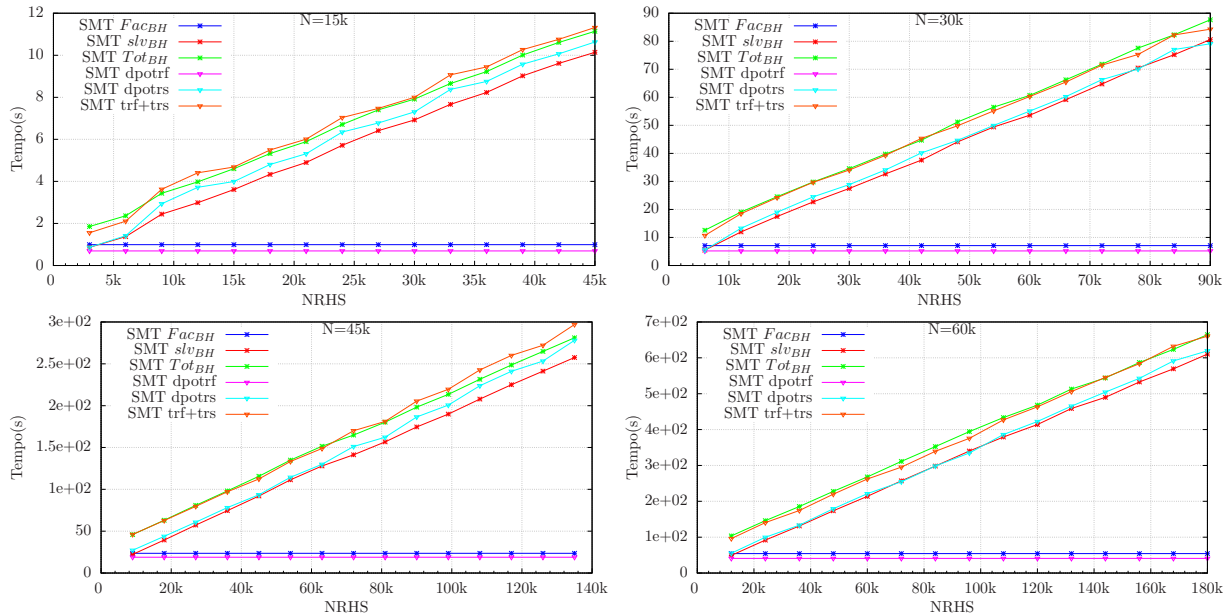


Figura E.3: Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $3N$, usando um CPU de 40 cores, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes.

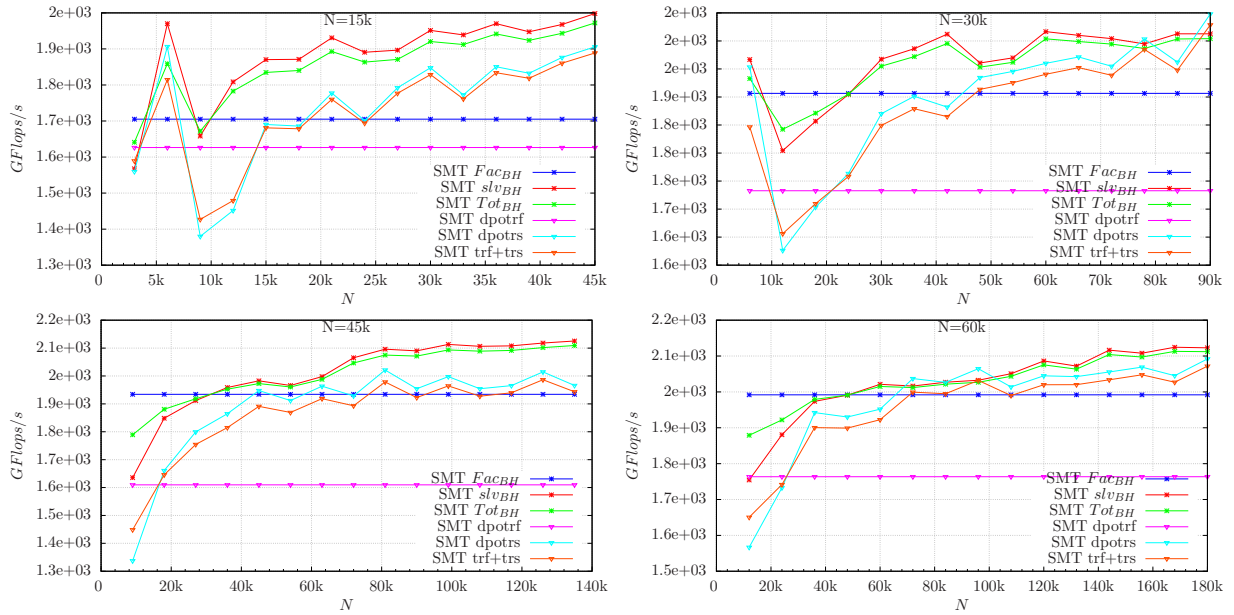


Figura E.4: Comparação de desempenho (GFLOP/s) entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $3N$, usando um CPU de 40 cores, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes.

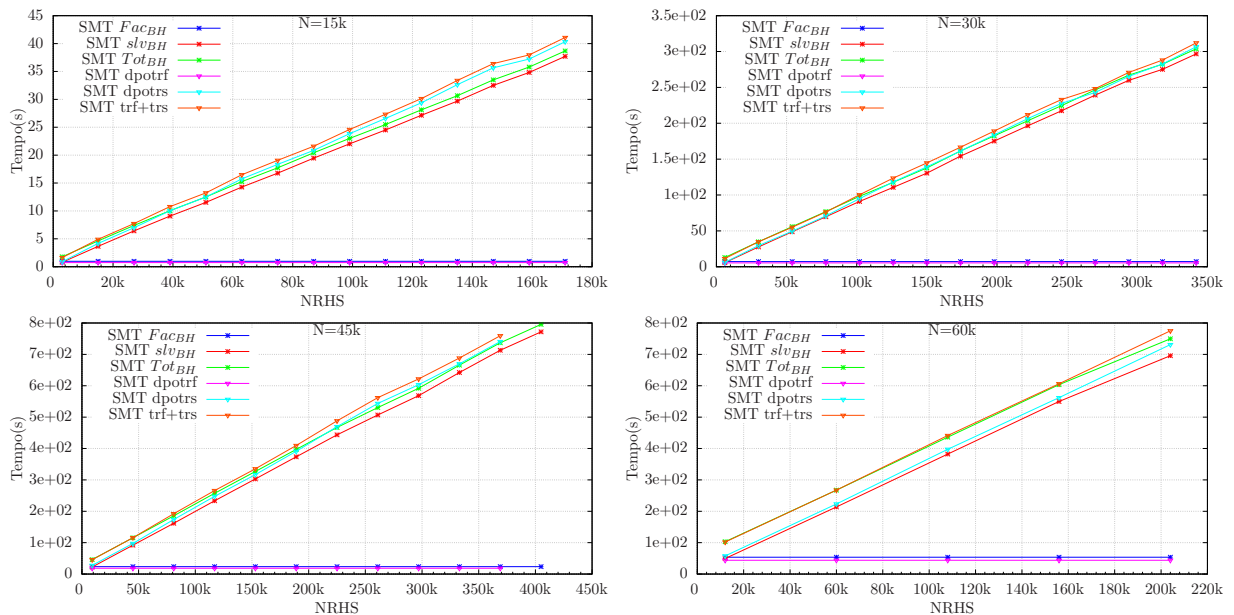


Figura E.5: Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $11N$, usando um CPU de 40 cores, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes.

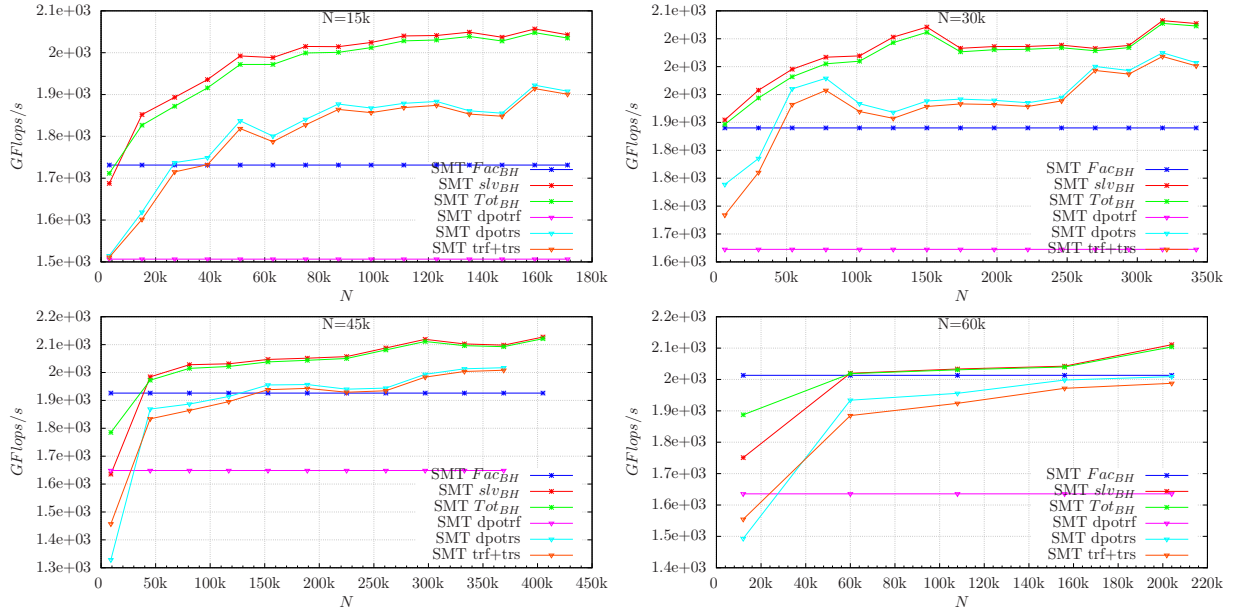


Figura E.6: Comparação de desempenho (GFLOP/s) entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k, N = 30k, N = 45k$ e $N = 60k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $11N$, usando um CPU de 40 cores, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes.

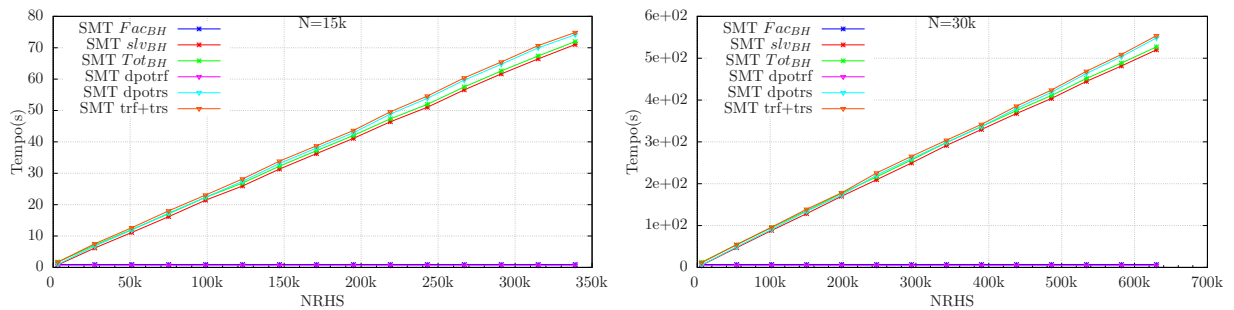


Figura E.7: Comparação entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k$ e $N = 30k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $20N$, usando um CPU de 40 cores, 80 threads (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes.

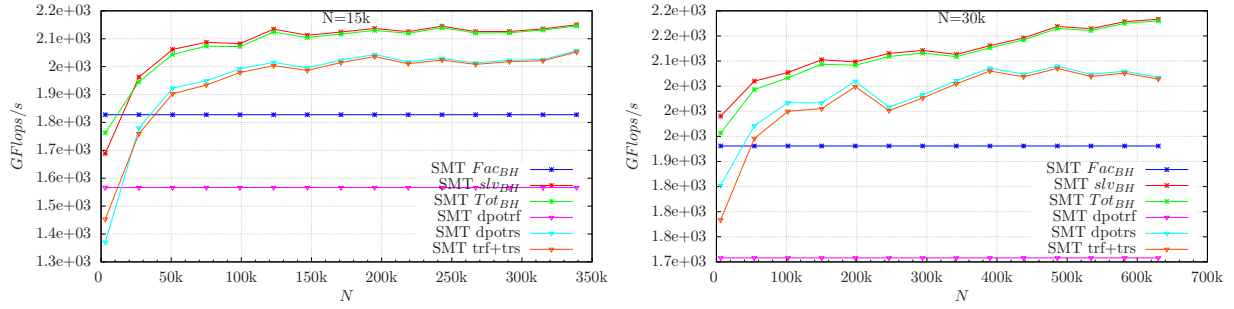


Figura E.8: Comparação de desempenho (GFLOP/s) entre o algoritmo de referência para sistema de memória compartilhada: (dpotrf + dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 15k$ e $N = 30k$ ($L = 2$) e números de tamanhos diferentes de RHS vai ate $20N$, usando um CPU de 40 cores, 80 *threads* (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes.

E.2 Versão memória distribuída

Nesta seção apresentamos resultados adicionais aos apresentados na seção 3.5. No gráfico E.9 mostramos o efeito do particionamento para diferentes tamanhos, em todos eles o mesmo efeito é evidente: em um nível maior de particionamento o tempo Fac_{BH} aumenta e tempo de solução slv_{BH} , tem um aparente declínio ou estabilização com o aumento de L .

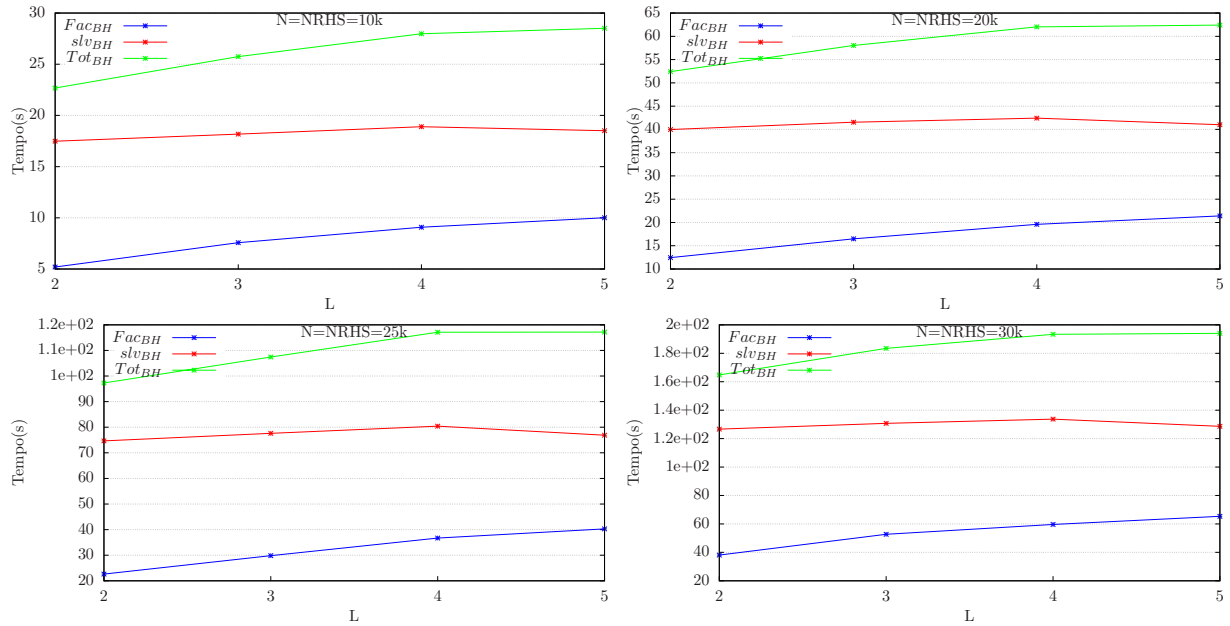


Figura E.9: Efeito do particionamento no desempenho do nosso algoritmo (Fac_{BH} e slv_{BH}) usando sistemas de memória distribuída, e diferentes dimensões ($N = NRHS = 10k$, $N = NRHS = 20k$, $N = NRHS = 25k$ e $N = NRHS = 30k$), variando o particionamento ($L = 2, 3, 4, 5$), usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

Agora, nas figuras E.10 e E.11 mostram a comparação do desempenho dos dois algoritmos para tamanhos diferentes (Sistema Marreca), onde se evidencia o mesmo comportamento, um desempenho menor de nossa implementação usando Scalapack.

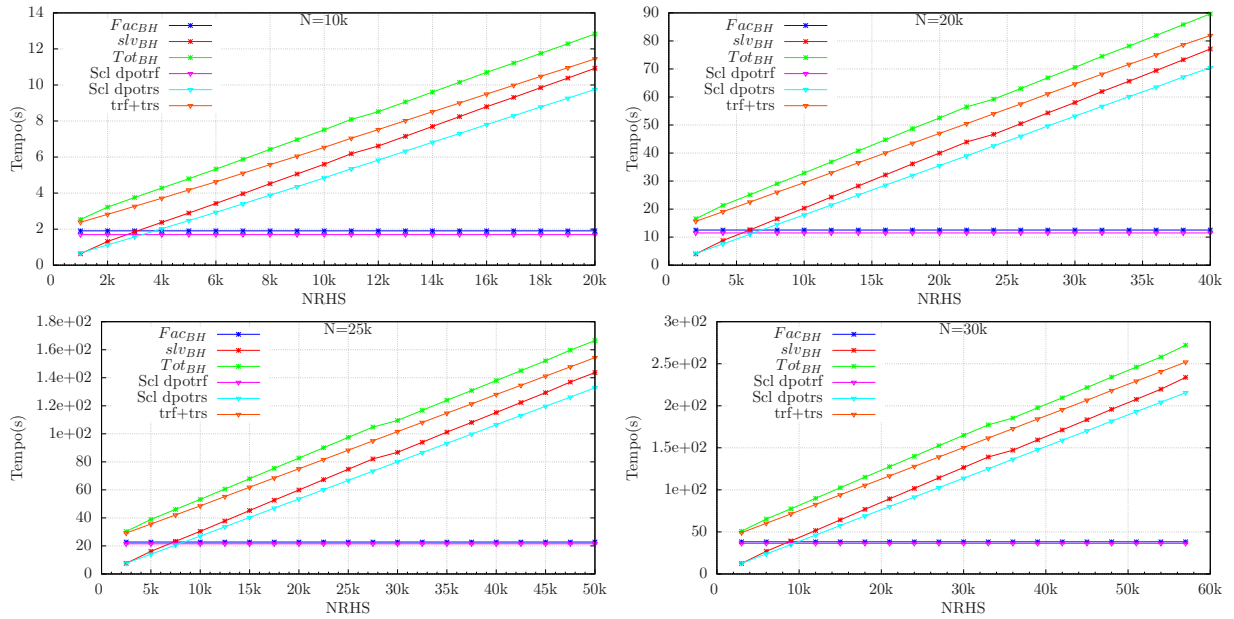


Figura E.10: Comparação entre o algoritmo de referência Scalapack (Scl dporf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para sistemas de memória distribuída, e diferente dimensões ($N = 10k$, $N = 20k$, $N = 25k$ e $N = 30k$) e números de tamanhos diferentes de RHS , usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

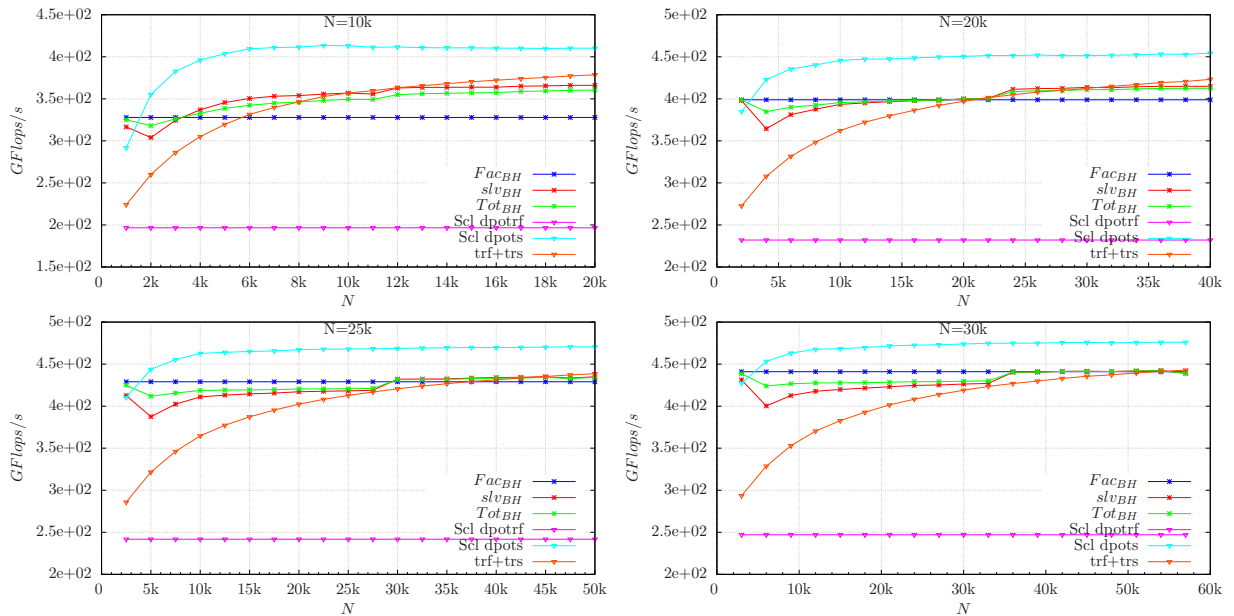


Figura E.11: Desempenho do algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dporf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 10k$ ($L = 2$ e $N_c = 5k$) e números de tamanhos diferentes de RHS , usando um CPU de 12 núcleos (Tabela 1.1). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

Por fim, a Figura E.12 apresenta o gráfico de desempenho (GFlops/s) utilizando o sistema OGUN com 300 núcleos, um complemento ao Gráfico 3.15, apresentado na seção 3.5.

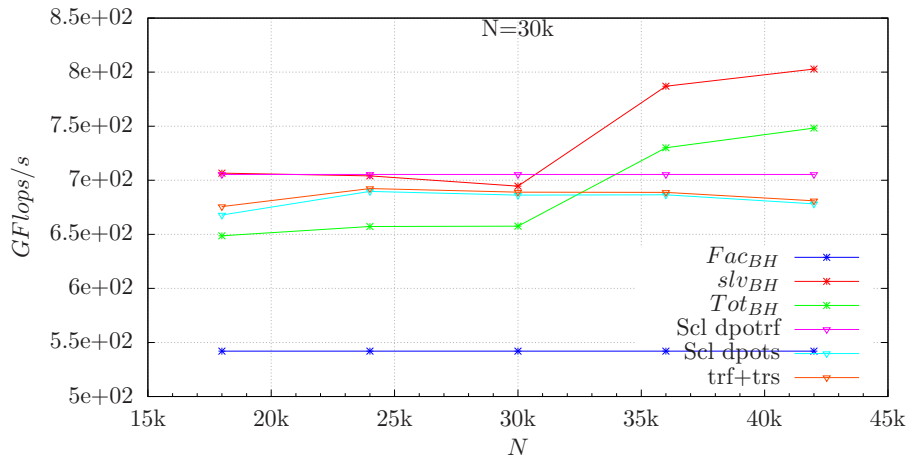


Figura E.12: Desempenho do algoritmo de referência para sistema de memória distribuída: Scalapack (Scl dporf + Scl dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 2$ e $N_c = 15k$) e números de tamanhos diferentes de RHS ($3k$ a $60k$), usando um CPU de 300 núcleos (Tabela 1.4). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

E.3 Versão OUT-OF-CORE usando GPU

E.3.1 GPU Tesla k40c: Tempo

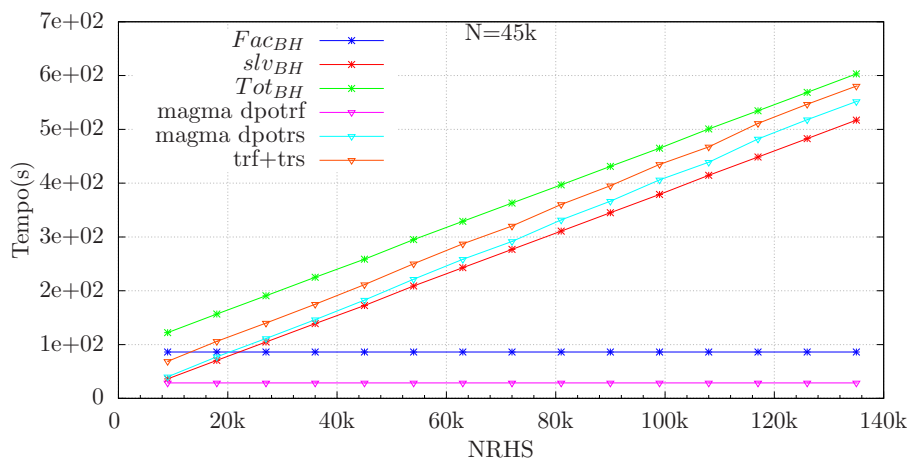


Figura E.13: Comparação entre magma (magmaf dporf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 4$ and $N_c = 11264$) e números de tamanhos diferentes de RHS ($4.5k$ to $135k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

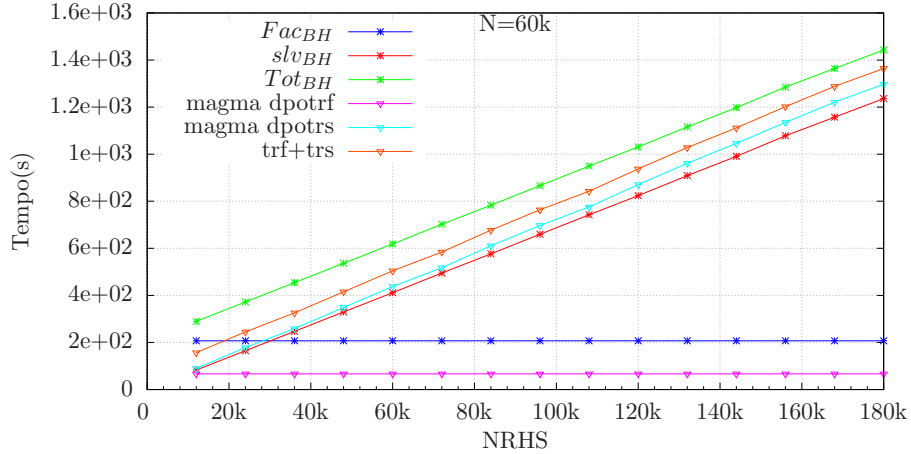


Figura E.14: Comparação entre magma_f (magma_f dporf + magma_f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 60k$ ($L = 5$ and $N_c = 12k$) e números de tamanhos diferentes de RHS ($6k$ to $180k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

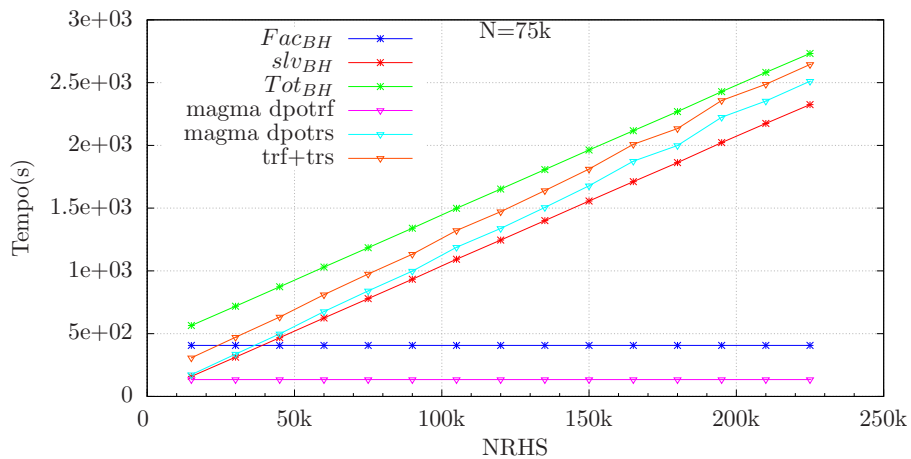


Figura E.15: Comparação entre magma_f (magma_f dporf + magma_f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 75k$ ($L = 6$ and $N_c = 12512$) e números de tamanhos diferentes de RHS ($7.5k$ to $225k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

A Tabela E.1 apresenta um resumo dos valores de L , tamanho de bloco (N_c) Numero máximo de RHS para cada um dos tamanhos de N utilizados.

N	L	N_c	RHS_{max}
45k	4	11264	53404
60k	5	12032	49525
75k	6	12544	46740
90k	7	12928	44973

Tabela E.1: Valores do particionamento (L), tamanho do bloco (N_c) e Numero máximo de RHS usando a GPU Tesla k40c (12GB) para diferentes tamanhos do sistema linear

E.3.2 GPU Tesla k40c: Desempenho

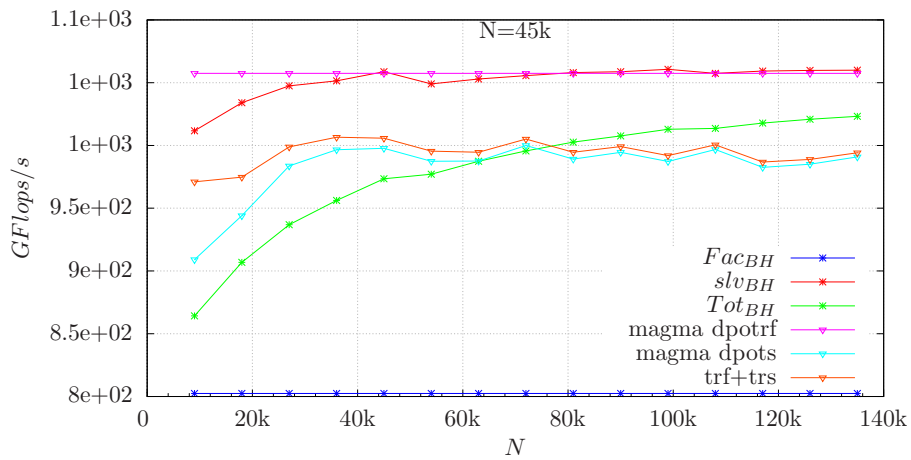


Figura E.16: Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 45k$ e diferentes valores de $NRHS$ usando a GPU tesla k40c

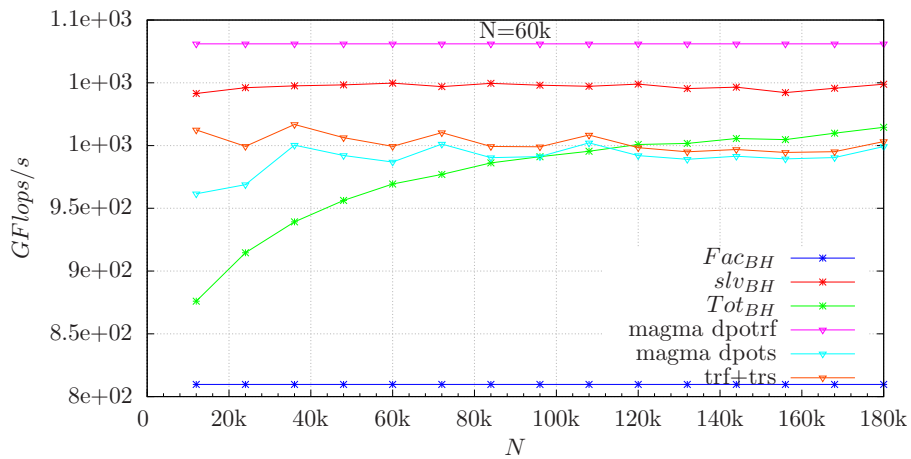


Figura E.17: Comparação de desempenho (GFLOP/s) entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 60k$ e diferentes valores de $NRHS$ usando a GPU tesla k40c

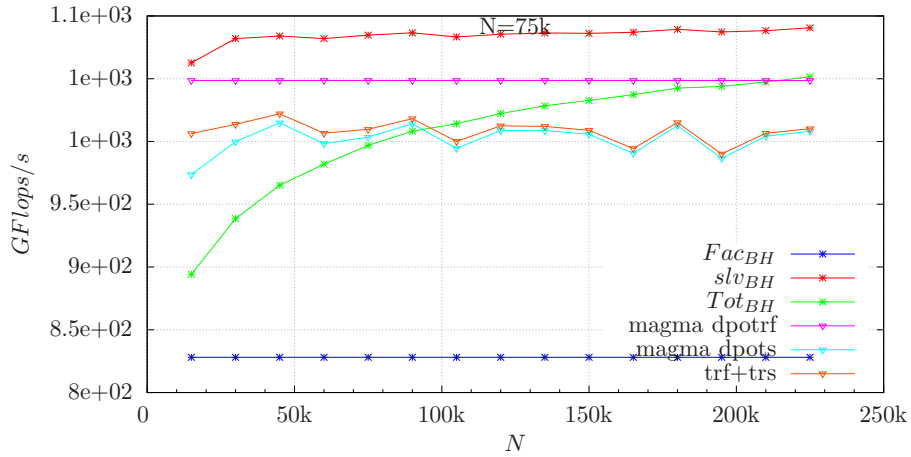


Figura E.18: Comparação de desempenho (GFLOP/s) entre magma (magma PDPOTRF + magma PDPOTRS) e nossas implementações para um sistema quando $N = 75k$ e diferentes valores de $NRHS$ usando a GPU tesla k40c

E.3.3 GPU Tesla P100: Desempenho

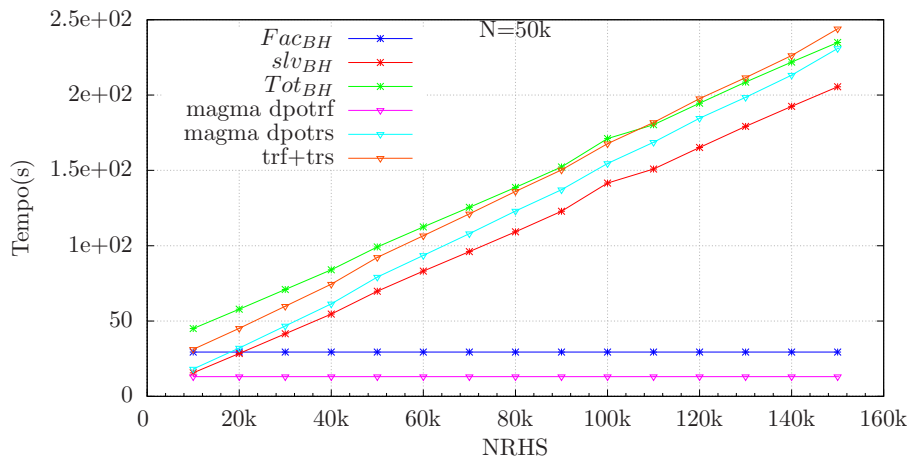


Figura E.19: Comparação entre magma (magma PDPOTRF + magma PDPOTRS) e nossas implementações para um sistema quando $N = 50k$ ($L = 3$ e $N_c = 16768$) e números de tamanhos diferentes de RHS ($10k$ a $150k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes

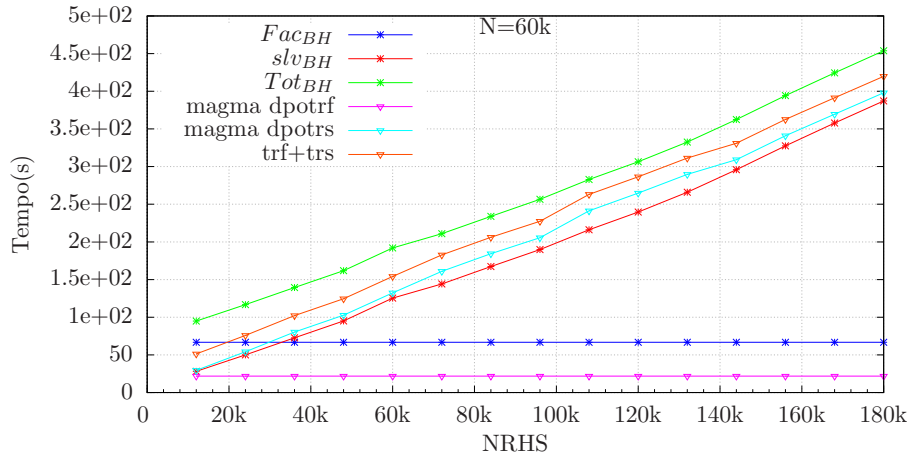


Figura E.20: Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 60k$ ($L = 4$ e $N_c = 15104$) e números de tamanhos diferentes de RHS (12k a 180k), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes

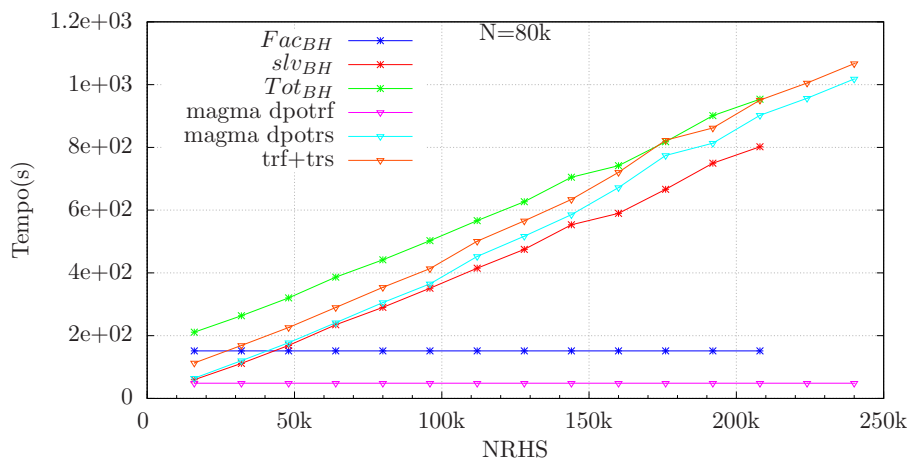


Figura E.21: Comparação entre magmaf (magmaf PDPOTRF + magmaf PDPOTRS) e nossas implementações para um sistema quando $N = 808$ ($L = 5$ e $N_c = 16000$) e números de tamanhos diferentes de RHS (16k a 240k), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes

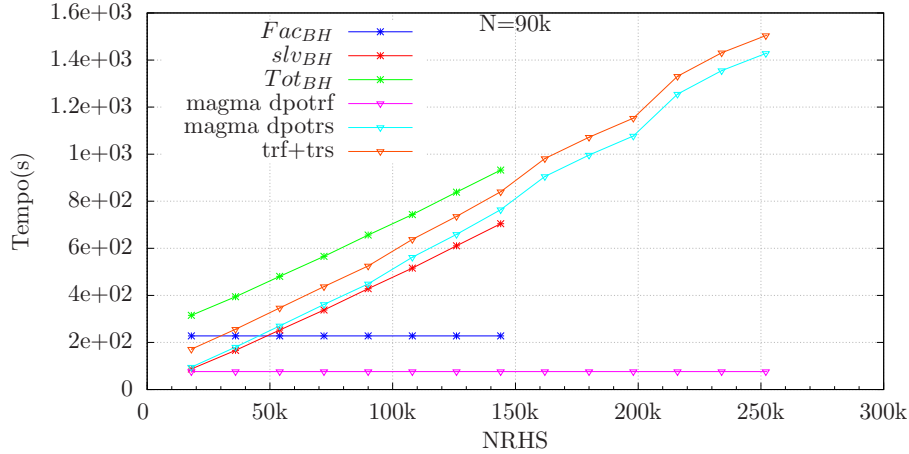


Figura E.22: Comparação entre magma_f (magma_f PDPOTRF + magma_f PDPOTRS) e nossas implementações para um sistema quando $N = 909$ ($L = 6$ e $N_c = 15104$) e números de tamanhos diferentes de RHS ($18k$ a $270k$), usando uma GPU NVIDIA Tesla P100 de 16 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, calculando a média dos três restantes

A Tabela E.2 apresenta um resumo dos valores de L , tamanho de bloco (N_c) Numero máximo de RHS para cada um dos tamanhos de N utilizados.

N	L	N_c	RHS_{max}
$50k$	3	16768	44692
$60k$	4	15104	51371
$70k$	5	14080	56169
$80k$	5	16000	47624
$90k$	6	15104	51371

Tabela E.2: Valores do particionamento (L), tamanho do bloco (N_c) e Numero máximo de RHS usando a GPU Tesla P100 (16GB) para diferentes tamanhos do sistema linear

E.3.4 Incrementando o $NRHS$ para GPU Tesla k40c: Desempenho

Assim como na figura 3.20, as figuras E.23 e E.24 mostram que a inclinação da linha slv_{BH} é menor que a da linha de magma.dpotrs, produzindo o mesmo efeito evidenciado anteriormente; as linhas de desempenho global Tot_{BH} (menor inclinação, melhor desempenho) e $trf + trs$ se cruzam em um determinado valor RHS, que depende do tamanho do sistema linear. Para $N = 45k$, o ponto de interseção está próximo de $NRHS = 100k$, para $N = 60k$ está em torno de $NRHS = 350k$. Para o caso de $N = 90k$ (Figura E.25), pode-se mostrar que o mesmo efeito de antes não ocorre.

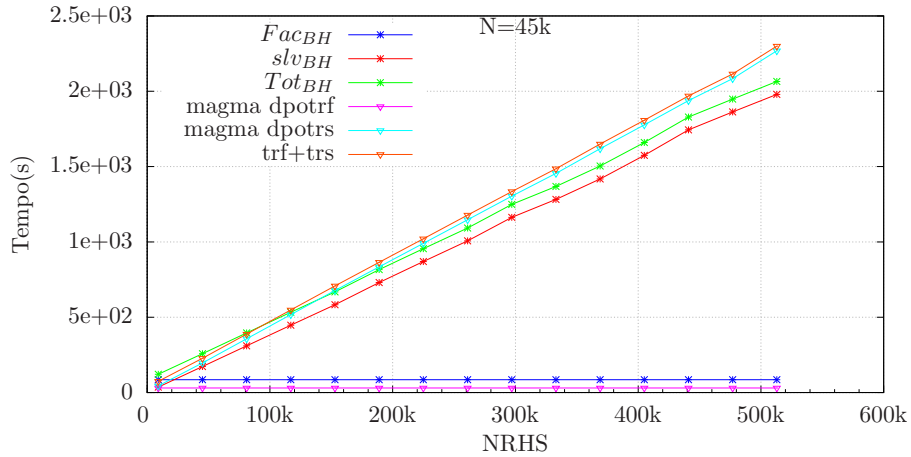


Figura E.23: Comparação entre magma_f (magma_f dpotrf + magma_f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 4$ e $N_c = 11264$) e números de tamanhos diferentes de RHS ($9k$ a $513k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

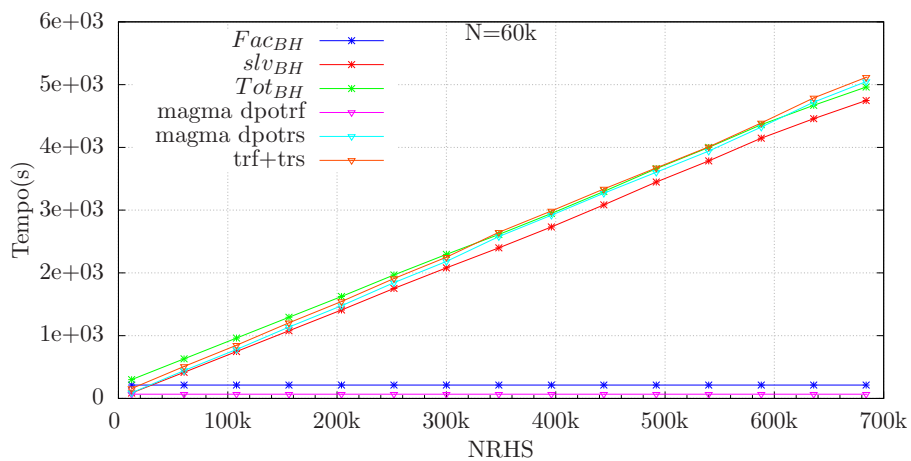


Figura E.24: Comparação entre magma_f (magma_f dpotrf + magma_f dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 60k$ ($L = 5$ e $N_c = 12032$) e números de tamanhos diferentes de RHS ($12k$ a $684k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

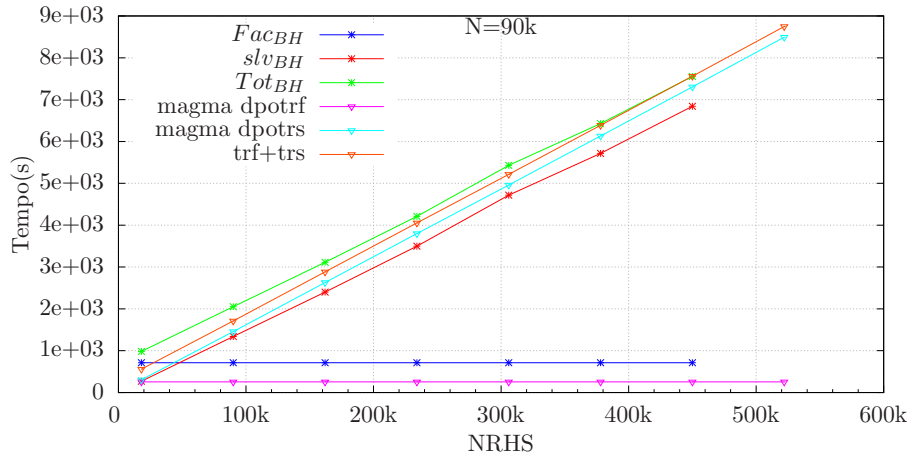


Figura E.25: Comparação entre magmaf (magmaf dpotrf + magmaf dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 90k$ ($L = 7$ e $N_c = 12928$) e números de tamanhos diferentes de RHS ($18k$ a $450k$), usando um NVIDIA Tesla k40c de 12 GB. Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

E.3.5 Limitando a memória máxima a ser usada na GPU Tesla k40c a 2GB

Nesta seção, mostramos que aumentando o particionamento (maiores valores de L) não influencia o desempenho de nossa implementação. A limitação de memória foi realizada em nossas 2 sub-rotinas, porém, não limitamos a memória para a fatoração de Cholesky (magmaf_dpotrf_m), apenas limitamos a memória da solução por blocos triangulares do sistema (substituição *Forward* e *Backward* para a solução do dois sistemas triangulares, apresentados no apêndice D, criando uma comparação injusta com nossos algoritmos. As figuras E.26, E.27, E.28, E.29 e E.30 ratificam o que foi evidenciado na figura 3.22; o desempenho do segundo algoritmo (que obtém a solução final) é melhor para todos os valores de $NRHS$ testados, e no desempenho total é semelhante, apesar de na fatoração de Cholesky o algoritmo de referência rencia (magma_dpotrs) usa toda a capacidade da GPU.

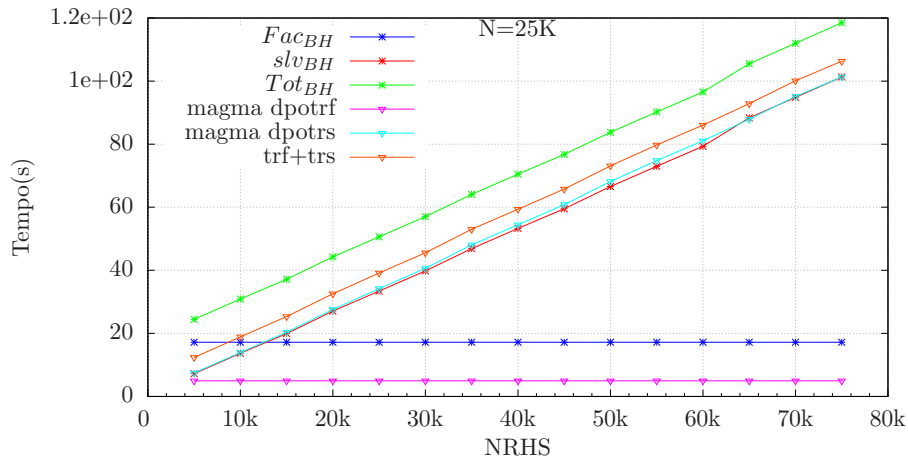


Figura E.26: Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L . Comparação entre magma (magma dpotrf + magma dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 25k$ ($L = 4$ e $N_c = 6272$) e números de tamanhos diferentes de RHS (5k a 75k). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

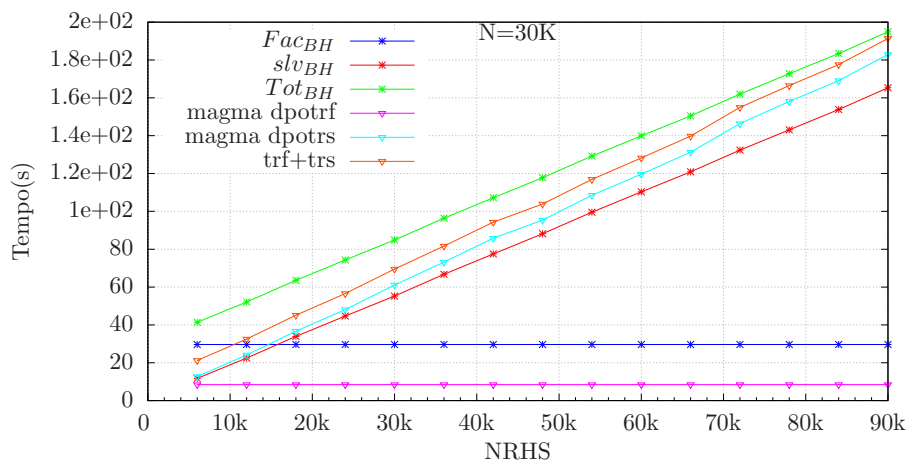


Figura E.27: Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L . Comparação entre magma (magma dpotrf + magma dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 30k$ ($L = 5$ e $N_c = 6016$) e números de tamanhos diferentes de RHS (6k a 90k). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

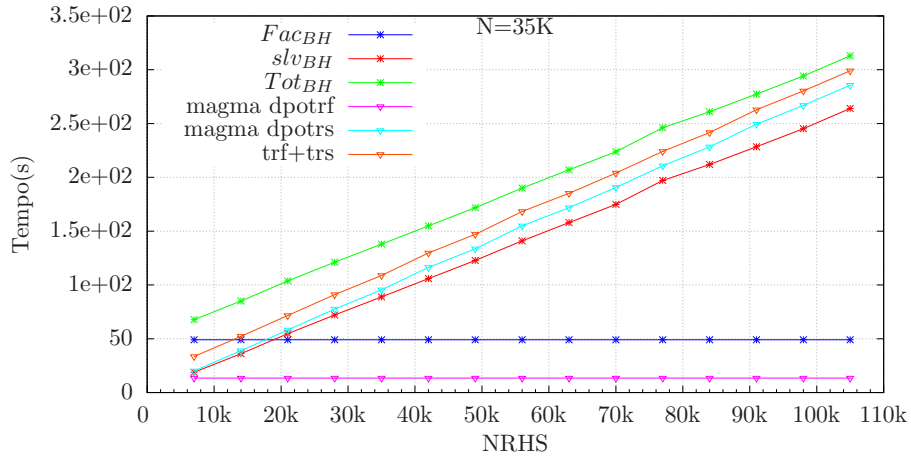


Figura E.28: Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L . Comparação entre magma (magma dporf + magma dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 35k$ ($L = 6$ e $N_c = 5888$) e números de tamanhos diferentes de RHS ($7k$ a $105k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

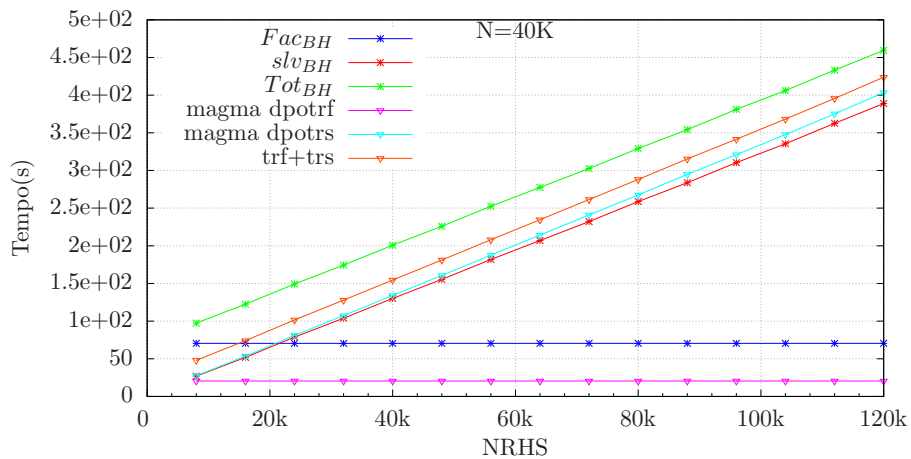


Figura E.29: Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L . Comparação entre magma (magma dporf + magma dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 40k$ ($L = 7$ e $N_c = 5760$) e números de tamanhos diferentes de RHS ($8k$ a $120k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

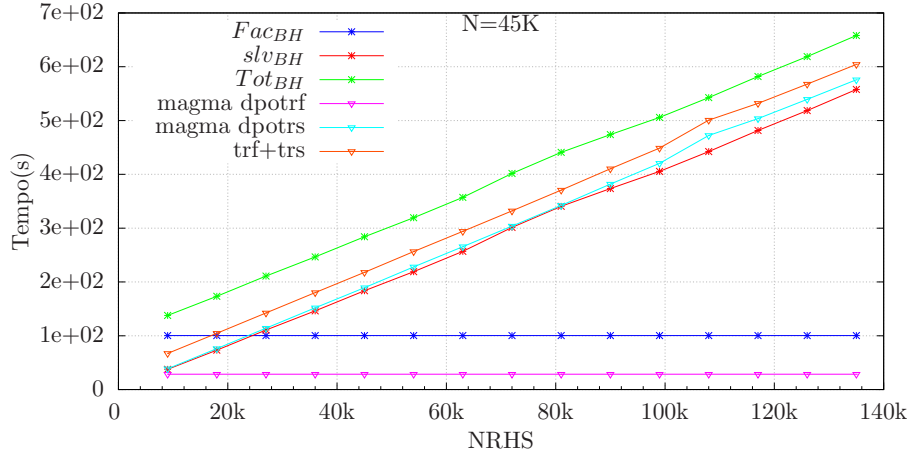


Figura E.30: Desempenho para quando se simula uma GPU de 2 GB, para aumentar o particionamento L . Comparação entre magma (magma dpotrf + magma dpotrs) e nossas implementações (Fac_{BH} e slv_{BH}) para um sistema quando $N = 45k$ ($L = 8$ e $N_c = 5632$) e números de tamanhos diferentes de RHS ($9k$ a $135k$). Foram realizadas 5 repetições e eliminados o melhor e o pior tempo, com média das três restantes

A Tabela E.3 apresenta um resumo dos valores de L e tamanho de bloco (N_c) para cada um dos tamanhos de N utilizados.

N	L	N_c	RHS_{max}
20k	4	5120	21032
25k	4	6272	16123
30k	5	6016	17071
35k	6	5888	17571
40k	7	5760	18091
45k	8	5632	18632
50k	8	6272	16123

Tabela E.3: Valores do particionamento (L), tamanho do bloco (N_c) e dimensão máxima do RHS quando é simulado uma GPU de 2GB para diferentes tamanho do sistema linear

Referências Bibliográficas

- Ajo-Franklin, J. B. (2005) Frequency-domain modeling techniques for the scalar wave equation: An introduction.
- Amestoy, P. R.; Duff, I. S.; L'Excellent, J.-Y. e Koster, J. (2001) A fully asynchronous multifrontal solver using distributed dynamic scheduling, *SIAM Journal on Matrix Analysis and Applications*, **23**(1):15–41.
- Anderson, E.; Benzoni, A.; Dongarra, J.; Moulton, S.; Ostrouchov, S.; Tourancheau, B. e van de Geijn, R. (1991) Basic linear algebra communication subprograms, In: *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pp. 287–288, IEEE Computer Society.
- Anderson, E.; Bai, Z.; Bischof, C.; Blackford, L. S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A. et al. (1999) *LAPACK Users' Guide*, SIAM.
- Arridge, S. R. (1999) Optical tomography in medical imaging, *Inverse problems*, **15**(2):R41.
- Arridge, S. R. e Hebden, J. C. (1997) Optical imaging in medicine: II modelling and reconstruction, *Physics in Medicine & Biology*, **42**(5):841.
- Aster, R. C.; Borchers, B. e Thurber, C. H. (2011) *Parameter Estimation and Inverse Problems*, vol. 90, Academic Press.
- Ben-Hadj-Ali, H.; Operto, S. e Virieux, J. (2011) An efficient frequency-domain full waveform inversion method using simultaneous encoded sources, *Geophysics*, **76**(4):R109–R124.
- Berkhout, A. J. (1977) Least-squares inverse filtering and wavelet deconvolution, *Geophysics*, **42**(07):1369–1383.
- Bertero, M. e Boccacci, P. (1998) *Introduction to Inverse Problems in Imaging*, CRC press.
- Björck, Å. (1996) *Numerical Methods for Least Squares Problems*, SIAM.
- Blackford, L. S.; Choi, J.; Cleary, A.; D'Azevedo, E.; Demmel, J.; Dhillon, I.; Dongarra, J.; Hammarling, S.; Henry, G.; Petitet, A. et al. (1997) *ScaLAPACK Users' Guide*, SIAM.

- Blackford, S. e Dongarra, J. (1999) LAPACK working note 41 installation guide for lapack, Department of Computer Science, University of Tennessee.
- BLAS, Basic Linear Algebra Subprograms (BLAS). *Disponível em:* <http://www.netlib.org/blas/> (*Acesso em:* 2021-02-18).
- Brossier, R.; Etienne, V.; Operto, S. e Virieux, J. (2010) Frequency-domain numerical modeling of visco-acoustic waves with finite-difference and finite-element discontinuous galerkin methods, *Acoustic waves*, pp. 434–p.
- Cerjan, C.; Kosloff, D.; Kosloff, R. e Reshef, M. (1985) A nonreflecting boundary condition for discrete acoustic and elastic wave equations, *Geophysics*, **50**(4):705–708.
- Chapman, B.; Jost, G. e Van Der Pas, R. (2008) Using OpenMP: Portable Shared Memory Parallel Programming, vol. 10, MIT press.
- Choi, J.; Dongarra, J. J.; Pozo, R. e Walker, D. W. (1992) ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers, In: *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 120–121, IEEE Computer Society.
- Choi, J.; Dongarra, J.; Ostrouchov, S.; Petitet, A.; Walker, D. e Whaley, R. C. (1995) A proposal for a set of parallel basic linear algebra subprograms, In: *International Workshop on Applied Parallel Computing*, pp. 107–114, Springer.
- Choi, J.; Demmel, J.; Dhillon, I.; Dongarra, J.; Ostrouchov, S.; Petitet, A.; Stanley, K.; Walker, D. e Whaley, R. C. (1996a) ScaLAPACK: A portable linear algebra library for distributed memory computers design issues and performance, *Computer Physics Communications*, **97**(1-2):1–15.
- Choi, J.; Dongarra, J. J.; Ostrouchov, L. S.; Petitet, A. P.; Walker, D. W. e Whaley, R. C. (1996b) Design and implementation of the scaLAPACK LU, QR, and cholesky factorization routines, *Scientific Programming*, **5**(3):173–184.
- Claerbout, J. F. (1976) *Fundamentals of Geophysical Data Processing*, McGraw-Hill Co., New York, 274p.
- Coulouris, G.; Dollimore, J.; Kindberg, T. e Blair, G. (2013) *Sistemas Distribuídos: Conceitos e Projeto*, Bookman Editora.
- Dablain, M. (1986) The application of high-order differencing to the scalar wave equation, *Geophysics*, **51**(1):54–66.
- Dongarra, J. (1994) Scalability issues in the design of a library for dense linear algebra, *Journal of Parallel and Distributed Computing*, **22**(3):523–537.

- Durbin, J. (1960) The fitting of time-series models, *Revue de l'Institut International de Statistique*, pp. 233–244.
- Engl, H. W.; Louis, A. K. e Rundell, W. (1996) Inverse problems in medical imaging and nondestructive testing, In: *Conference in Oberwolfach, Federal Republic of Germany*, Springer-Verlag Wien GmbH.
- Foster, I. T. (1995) *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley.
- Gardan, Y. (1985) Numerical methods for solving linear and non-linear equations, In: *Mathematics and CAD*, pp. 91–134, Springer.
- Golub, G. H. e Van Loan, C. F. (2013) *Matrix Computations*, vol. 3, JHU press.
- Goto, K. e Van De Geijn, R. (2008) High-performance implementation of the level-3 BLAS, *ACM Transactions on Mathematical Software (TOMS)*, **35**(1):1–14.
- Gu, B.; Liang, G. e Li, Z. (2013) A 21-point finite difference scheme for 2d frequency-domain elastic wave modelling, *Exploration Geophysics*, **44**(3):156–166.
- Gustavson, F. G. (1997) Recursion leads to automatic variable blocking for dense linear-algebra algorithms, *IBM Journal of Research and Development*, **41**(6):737–755.
- Hadamard, J. (1932) *Le Probleme de Cauchy et les Équations aux Dérivées Partielles Linéaires Hyperboliques*, vol. 220, Paris Russian translation).
- Jo, C.-H.; Shin, C. e Suh, J. H. (1996) An optimal 9-point, finite-difference, frequency-space, 2-d scalar wave extrapolator, *Geophysics*, **61**(2):529–537.
- Kale, V. (2019) *Parallel Computing Architectures and APIs: IoT Big Data Stream Processing*, CRC Press.
- Kaminsky, A. (2009) *Building Parallel Programs: SMPs, Clusters & Java*, Nelson Education.
- Kearey, P.; Brooks, M. e Hill, I. (2002) *An Introduction to Geophysical Exploration*, vol. 4, John Wiley & Sons.
- Kim, Y.; Min, D.-J. e Shin, C. (2011) Frequency-domain reverse-time migration with source estimation, *Geophysics*, **76**(2):S41–S49.
- LAPACK, Linear Algebra PACKage. *Disponível em: <http://www.netlib.org/lapack/>* (Acesso em: 2021-02-18).
- Levinson, N. (1946) The Wiener (root mean square) error criterion in filter design and prediction, *Studies in Applied Mathematics*, **25**(1-4):261–278.

- Liao, W.; Yong, P.; Dastour, H. e Huang, J. (2018) Efficient and accurate numerical simulation of acoustic wave propagation in a 2d heterogeneous media, *Applied Mathematics and Computation*, **321**:385–400.
- Louis, A. (1992) Medical imaging: state of the art and future development, *Inverse Problems*, **8**(5):709.
- Luo, J. e Xie, X.-B. (2017) Frequency-domain full waveform inversion with an angle-domain wavenumber filter, *Journal of Applied Geophysics*, **141**:107–118.
- Ma, X.; Li, H.; Li, G. e Li, Y. (2023) Frequency-domain reverse-time migration with attenuation compensation, *IEEE Transactions on Geoscience and Remote Sensing*.
- MAGMA, Matrix Algebra on GPU and Multicore Architectures (MAGMA). *Disponível em: <https://icl.cs.utk.edu/magma/>* (Acesso em: 2021-02-18).
- Menke, W. (2012) *Geophysical Data Analysis: Discrete Inverse Theory: MATLAB edition*, vol. 45, Academic press.
- MKL, Math Kernel Library (MKL). *Disponível em: <http://software.intel.com/en-us/intel-mkl>* (Acesso em: 2021-02-18).
- Mulder, W. e Plessix, R.-E. (2004) How to choose a subset of frequencies in frequency-domain finite-difference migration, *Geophysical Journal International*, **158**(3):801–812.
- Peacock, K. L. e Treitel, S. (1969) Predictive deconvolution - Theory and practice, *Geophysics*, **34**(02):155–169.
- Petit, A. P. e Dongarra, J. J. (1999) Algorithmic redistribution methods for block-cyclic decompositions, *IEEE Transactions on Parallel and Distributed Systems*, **10**(12):1201–1216.
- Porsani, M. J. e Ulych, T. J. (1991) Levinson-type extensions for non-Toeplitz systems, *IEEE Transactions on signal processing*, **39**(2):366–375.
- Porsani, M. J. e Ursin, B. (2007) Direct multichannel predictive deconvolution, *Geophysics*, **72**(2):H11–H27.
- Porsani, M. J.; Stoffa, P. L.; Sen, M. K. e Seif, R. K. (2010) Partitioned least-squares operator for large-scale geophysical inversion, *Geophysics*, **75**(6):R121–R128.
- Pratt, R. G. (1990) Inverse theory applied to multi-source cross-hole tomography.: Part 2: Elastic wave-equation method1, *Geophysical Prospecting*, **38**(3):311–329.
- Pratt, R. G. e Worthington, M. H. (1990) Inverse theory applied to multi-source cross-hole tomography. part 1: Acoustic wave-equation method 1, *Geophysical prospecting*, **38**(3):287–310.

- Revelo, D. (2015) Modelagem e migração de dados sísmicos no domínio do tempo através do operador exponencial matricial e no domínio da frequência a partir da solução da equação de Helmholtz, Dissert. de Mestrado, UFBA.
- Robinson, E. A. (1957) Predictive decomposition of seismic traces, *Geophysics*, **22**(04):767–778.
- Robinson, E. A. (1983) *Multichannel Time Series Analysis with Digital Computer Programs*, Goose Pond Press.
- Russell, B. H. (1988) *Introduction to Seismic Inversion Methods*, vol. 2, Society of Exploration Geophysicists Tulsa.
- Shin, C. e Sohn, H. (1998) A frequency-space 2-d scalar wave extrapolator using extended 25-point finite-difference operator, *Geophysics*, **63**(1):289–296.
- Snieder, R. e Trampert, J. (2000) Linear and nonlinear inverse problems, In: *Geomatic method for the analysis of data in the earth sciences*, pp. 93–164, Springer.
- Tanenbaum, A. S. e Austin, T. (2012) *Structured Computer Organization*, Pearson, 6^o edic..
- Tarantola, A. (2005) *Inverse Problem Theory and Methods for Model Parameter Estimation*, vol. 89, SIAM.
- Tikhonov, A. (1963) Solution of incorrectly formulated problems and the regularization method, *Soviet Meth. Dokl.*, **4**:1035–1038.
- Tikhonov, A. N. e Arsenin, V. Y. (1977) *Solutions of Ill-posed Problems*, V. H. WINSTON and SONS.
- Tikhonov, A. N.; Goncharsky, A.; Stepanov, V. e Yagola, A. G. (2013) *Numerical Methods for the Solution of Ill-posed Problems*, vol. 328, Springer Science Business Media.
- Tomov, S.; Nath, R.; Ltaief, H. e Dongarra, J. (2010) Dense linear algebra solvers for multicore with GPU accelerators, In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, IEEE.
- Treitel, S. (1970) Principles of digital multichannel filtering, *Geophysics*, **35**(5):785–811.
- Treitel, S. e Robinson, E. A. (1966) The design of high-resolution digital filters, *IEEE Trans. on Geoscience Electronics*, **GE-4**:25–38.
- Trench, W. F. (1964) An algorithm for the inversion of finite toeplitz matrices, *Journal of the Society for Industrial and Applied Mathematics*, **12**(3):515–522.
- Wang, Q.; Zhang, X.; Zhang, Y. e Yi, Q. (2013) Augem: automatically generate high performance dense linear algebra kernels on x86 cpus, In: *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analy-*

- sis, pp. 1–12, IEEE. *Disponível em:* http://xianyi.github.io/paper/agem_SC13.pdf (*Acesso em:* 2021-02-20).
- Wiggins, R. A. e Robinson, E. A. (1965) Recursive solution to the multichannel filtering problem, *J. of Geophys. Res.*, **70**:1885–1891.
- Xu, W.; Wu, B.; Zhong, Y.; Gao, J. e Liu, Q. H. (2021) Adaptive 27-point finite-difference frequency-domain method for wave simulation of 3d acoustic wave equation, *Geophysics*, **86**(6):T439–T449.
- Zhdanov, M. S. (2002) *Geophysical Inverse Theory and Regularization Problems*, vol. 36, Elsevier.
- Zohar, S. (1969) Toeplitz matrix inversion: The algorithm of WF trench, *Journal of the ACM (JACM)*, **16**(4):592–601.