PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

COMPUTAÇÃO VERIFICADA APLICADA À RESOLUÇÃO DE SISTEMAS LINEARES INTERVALARES DENSOS EM ARQUITETURAS MULTICORE

CLEBER ROBERTO MILANI

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes
Co-orientadora: Prof^a. Dr^a. Mariana Luderitz Kolberg

Porto Alegre
2010

Dados Internacionais de Catalogação na Publicação (CIP)

M637c Milani, Cleber Roberto
Computação verificada aplicada à resolução de sistemas lineares intervalares densos em arquiteturas Multicore / Cleber Roberto Milani. – Porto Alegre, 2010.
87 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.

1. Informática. 2. Matemática Computacional. 3. Sistemas Lineares. 4. Processamento Paralelo. 5. Arquitetura de Computador. I. Fernandes, Luiz Gustavo Leão.
II. Título.

CDD 004.0151

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "**Computação Verificada Aplicada à Resolução de Sistemas Lineares Intervalares Densos em Arquiteturas Multicore**", apresentada por Cleber Roberto Milani, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 25/03/10 pela Comissão Examinadora:

Prof. Dr. Luiz Gustavo Leão Fernandes – Orientador — PPGCC/PUCRS

Profa. Dra. Mariana Luderitz Kolberg (Co-orientadora) – — ULBRA

Prof. Dr. César Augusto Fonticielha De Rose - — PPGCC/PUCRS

Prof. Dr. Dalcidio Moraes Claudio - — PUCRS/FAMAT

Homologada em 17/09/2010, conforme Ata No. 181.10 pela Comissão Coordenadora.

P/

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

"A journey of a thousand miles
begins with a single step."
Lao-tzu, The Way of Lao-tzu

AGRADECIMENTOS

Primeiramente, a Deus por iluminar meu caminho e ter me permitido chegar até aqui.

A meus pais, Benoni e Inês, e irmão Jeferson por tudo, principalmente pelo apoio e esforços dispensados para que eu pudesse me dedicar durante estes anos de estudo.

Aos orientadores "Gãs" (Luiz Gustavo) e Mariana pelas discussões, idéias, auxílios burocráticos e pela amizade.

À Jajá, minha chefe e amiga durante o segundo ano de realização desta Dissertação, pela compreensão ao negociar horários e eventuais faltas ao trabalho.

A todos os colegas pelas trocas de experiências, ajudas, mas, principalmente, pela amizade e companheirismo. Em especial aos colegas Dione, Dorn e Mateus pelos diversos "galhos quebrados".

Ao LAD (Laboratório de Alto Desempenho) da PUC-RS nas figuras do professor De Rose, por disponibilizar a estrutura para execução dos testes, e do colega Antonio de Lima que colaborou muito na realização deste trabalho por sua disposição em me auxiliar, sempre de boa vontade, na utilização dos recursos do laboratório.

A todos os amigos, em especial à "parenta" Cris, à Eliz e ao Papalia que me acompanharam de perto durante a realização deste trabalho.

Meu muito obrigado a todos! Com certeza sem vocês a realização deste trabalho não seria possível. Todos são, de alguma forma, parte fundamental da jornada.

COMPUTAÇÃO VERIFICADA APLICADA À RESOLUÇÃO DE SISTEMAS LINEARES INTERVALARES DENSOS EM ARQUITETURAS MULTICORE

RESUMO

A resolução de Sistemas de Equações Lineares é um problema de grande importância em Ciência da Computação. Entretanto, os métodos tradicionais não oferecem garantia de soluções corretas e nem mesmo da existência de uma solução. Por isso, cada vez mais tem-se aplicado a Computação Verificada em tais algoritmos. Por outro lado, a Computação Verificada aumenta o custo computacional e, em alguns casos, impossibilita a resolução dos sistemas em um tempo aceitável. Uma alternativa encontrada para minimizar o custo é a utilização de Computação Paralela. Diversos trabalhos têm focado em otimizar a Computação Verificada para execução em agregados de computadores. Entretanto, dado o grande avanço dos processadores com múltiplos núcleos de processamento (*cores*), é uma necessidade premente que sejam também propostas soluções baseadas em modelos de paralelismo para memória compartilhada buscando, assim, explorar eficientemente as novas arquiteturas. Nesse contexto, o presente trabalho apresenta uma ferramenta para resolução verificada de Sistemas Lineares Densos Intervalares de Grande Porte. Além de prover verificação automática dos resultados, a ferramenta é otimizada para execução em arquiteturas *multicore*. As estratégias adotadas permitiram desenvolver uma solução escalável que, ao resolver Sistemas Intervalares de ordem 15.000x15.000 em um computador com 8 *cores*, obteve redução de 85% no tempo de execução e *speedup* de 6,70 em comparação com a solução inicial.

Palavras-chave: Computação Verificada; Sistemas Lineares Intervalares; *Multicore*.

VERIFIED COMPUTING APPLIED TO INTERVAL LINEAR SYSTEMS SOLVING ON MULTICORE COMPUTERS

ABSTRACT

Bounding the solution set of Systems of Linear Equations is a major problem in Computer Science. However, traditional methods offer no guarantee of correct solutions and not even of the existence of a solution. Hence, automatic result verification is an important additional tool in these algorithms. However, Verified Computing increases the computational cost and, in some cases, the required resolution time becomes unacceptable. The use of High Performance Computing (HPC) techniques appears as a solution. Several works have focused on optimizing Verified Computing performance for computer clusters. However, many changes have been occurring in High Performance Computing. Given the number of cores on multicore chips expected to reach tens in a few years, efficient implementations of numerical solutions using shared memory programming models is of urgent interest. In this context, this work presents a self-verified solver for Dense Interval Linear Systems optimized for parallel execution on multicores processors. The adopted strategies have resulted in a scalable solver that obtained up to 85% of reduction at execution time and a speedup of 6.70 when solving a 15,000x15,000 Interval Linear System on a eight core computer.

Keywords: Verified Computing; Interval Linear Systems; Multicore.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

ACML	*AMD Core Math Library*
ATLAS	*Automatically Tuned Linear Algebra Software*
BLACS	*Basic Linear Algebra Communication Subprograms*
BLAS	*Basic Linear Algebra Subprograms*
C-XSC	*C for eXtended Scientific Computing*
DAG	*Direct Acyclic Graph*
ECC	*Error Correction Code*
ED	Equação Diferencial
EISPACK	*Eigensystem Package*
FORTRAN	*IBM Mathematical FORmula TRANslation System*
FPGA	*Field Programmable Gate Array*
GPU	*Graphics Processing Unit*
HPC	*High Performance Computing*
HPL	*Highly Parallel Computing Benchmark*
ILP	*Instruction Level Parallelism*
LAD	Laboratório de Alto Desempenho
LAPACK	*Linear Algebra PACKage*
MAGMA	*Matrix Algebra on GPU and Multicore Architectures*
MKL	*Intel Math Kernel Library*
MLIB	*HP's Mathematical Software Library*
MPI	*Message Passing Interface*
NRHS	*Number of Right Hand Sides*
NUMA	*Non-Uniform Memory Access*
PBLAS	*Parallel Basic Linear Algebra Subprograms*
PB-BLAS	*Parallel Block Basic Linear Algebra Subprograms*
PLASMA	*Parallel Linear Algebra for Scalable Multi-Core Architectures*
POSIX	*Portable Operating System Interface*
PUC-RS	Pontifícia Universidade Católica do Rio Grande do Sul
PVM	*Parallel Virtual Machine*
ScaLAPACK	*SCAlable Linear Algebra PACKage*
SELA	Sistema de Equações Lineares Algébricas

SMP	*Symmetric Multiprocessor*
SO	Sistema Operacional
SPMD	*Single-Program-Multiple-Data*
SSE	*Intel's Streaming SIMD Extensions*
TLB	*Translation Lookahead Buffer*
TLP	*Thread Level Parallelism*

SUMÁRIO

1. INTRODUÇÃO

Diversos problemas podem ser modelados através de Equações Diferenciais (EDs) e Sistemas de Equações Lineares Algébricas (SELAs). Uma ED é definida em um conjunto infinito de pontos, ou seja, em um espaço contínuo. Porém, para que possa ser resolvida computacionalmente, faz-se necessário discretizá-la (criar um espaço discreto e finito), processo esse que implica na resolução de SELAs. Dessa forma, a resolução de SELAs assume papel de extrema importância para as mais variadas áreas do conhecimento.

Um Sistema Linear é tipicamente expresso na forma $Ax = b$, na qual A é uma matriz formada pelos coeficientes, x é o vetor das incógnitas do sistema e b é o vetor dos termos independentes. O sistema é composto por n equações com n incógnitas e, quando empregado na modelagem de problemas ou discretização de EDs, o valor de n costuma ser na ordem dos milhares. SELAs podem ser densos ou esparsos. Um Sistema Esparso é aquele em que 80% ou mais dos coeficientes das incógnitas nas equações são nulos, enquanto os densos possuem coeficientes não nulos para mais de 20% das variáveis. Por fim, sistemas cujo valor de n é elevado são ditos de grande porte [CLA00].

Dada sua complexidade, a resolução de SELAs de grande porte requer, na prática, a utilização de ferramentas computacionais e, com frequência, de Computação Paralela de Alto Desempenho. No entanto, em Ciência da Computação a garantia de que um algoritmo foi testado e está correto não implica, necessariamente, que o resultado computado por ele será correto [HAM97]. Frequentemente, um computador digital produz resultados incorretos para problemas numéricos, não devido a erros de programação ou ao uso de *hardware* não confiável, mas, simplesmente, por serem máquinas discretas e finitas que não conseguem tratar alguns dos aspectos contínuos e infinitos da Matemática. Até mesmo um número simples como $\frac{1}{10}$ pode causar grandes problemas, pois o computador não é capaz de executar cálculos exatos com o mesmo. Isso ocorre porque a fração decimal $\frac{1}{10}$ não possui representação finita exata na notação binária [HAY03]. Em tais situações, os computadores aproximam, através de frações finitas, os números reais originalmente expressos por frações decimais infinitas. Essa diferença entre o valor exato e o aproximado é dita **erro de arredondamento** [HAM97], [HAY03].

Assim, os métodos numéricos que utilizam Aritmética de Ponto Flutuante tradicional oferecem apenas uma aproximação do resultado e, uma vez que o resultado exato é desconhecido, não é possível saber quão boa é essa aproximação. Já a Computação Verificada permite ao computador determinar se o resultado encontrado é correto e/ou útil. De posse dessa informação, é possível optar por um algoritmo alternativo, repetir a computação utilizando maior exatidão ou informar ao usuário quando o resultado não é válido. Logo, uma execução de um algoritmo auto-validado fornece resultados com uma garantia que não pode ser oferecida mesmo por milhões de execuções com ponto flutuante tradicional. Adicionalmente, muitas vezes tais técnicas permitem ao computador estabelecer a existência e unicidade da solução. Portanto, a verificação automática dos resultados é de suma importância na redução do impacto dos erros de aritmética na Computação Numérica.

A Computação Verificada emprega a Aritmética Intervalar que, por sua vez, define as operações aritméticas básicas através de números intervalares (intervalos numéricos). Além de viabilizar a aplicação prática da Computação Verificada, outro grande benefício da Aritmética Intervalar é permitir trabalhar-se com os **Sistemas Lineares Intervalares**, ou seja, sistemas em que os valores contidos em A e b são intervalos ao invés de números pontuais. Tais sistemas têm adquirido cada vez mais importância científica por permitirem a representação de dados incertos, ou seja, aqueles em que os instrumentos empregados para sua obtenção apresentam variações e imperfeições ou em que a modelagem não é exata, dentre outros. Em princípio, a resolução verificada desses SELAs implicaria na necessidade de resolver um número infinito de matrizes contidas em um intervalo. Entretanto, por ser esse um problema NP-Completo, o que acontece na prática é a computação de um intervalo estreito o qual contém tais matrizes. Uma estratégia, para tal, é obter soluções aproximadas de **Sistemas Lineares Pontuais**, ou seja, aqueles cujas equações são formadas por números pontuais, com o auxílio de bibliotecas matemáticas otimizadas, tais como o LINPACK [DON79], LAPACK (*Linear Algebra PACKage*) [AND99], [DEM89], [AND90] ou ScaLAPACK (*SCAlable Linear Algebra PACKage*) [CHO96], [CHO95a] e, a partir da solução aproximada, calcular os limites que constituem o intervalo no qual a solução reside [KEA96].

Bibliotecas como LAPACK e ScaLAPACK fazem uso de Computação Paralela visando otimizar a resolução dos mais variados problemas de Álgebra Linear. Entretanto, mesmo com a utilização dessas bibliotecas de *software* otimizadas, a resolução de SELAs continua apresentando grande custo computacional quando se tratando de Sistemas de Grande Porte. O custo torna-se ainda maior ao considerar Sistemas Lineares Intervalares. Além disso, as bibliotecas otimizadas disponíveis atualmente para Álgebra Linear não oferecem suporte à resolução de Sistemas Intervalares, bem como não suportam resolução verificada mesmo de Sistemas Pontuais. As bibliotecas puramente de Computação Verificada, por outro lado, embora permitam a resolução verificada e, em alguns casos, deem suporte aos Sistemas Intervalares, acabam transformando-se em um gargalo na aplicação por requererem uma grande quantidade de operações adicionais. A ferramenta de Computação Verificada mais popular no mundo hoje é o C-XSC (*C for eXtended Scientific Computing*) [KLA93].

Visando equilibrar esse cenário, diversos trabalhos têm sido desenvolvidos combinando algoritmos de Computação Verificada com técnicas da Computação Paralela. Em geral, tais trabalhos são voltados aos agregados de computadores (*clusters*) e empregam, por exemplo, a biblioteca MPI (*Message Passing Interface*) [SNI96] em conjunto com o ScaLAPACK. Basicamente, utilizam-se as bibliotecas otimizadas para acelerar alguns trechos específicos dos algoritmos de Computação Verificada. Entretanto, nos últimos anos muitas modificações vêm ocorrendo na Computação de Alto Desempenho. Os processadores atingiram frequências de *clock* bastante próximas do limite suportado pelas atuais tecnologias, o que tem dificultado crescimento maior de sua velocidade, pois o aumento da frequência dos *clocks* tornou-se um esforço com custos muito elevados e ganhos, proporcionalmente, muito baixos [SKI98].

Diante disso, a solução encontrada pelas fabricantes tem sido empacotar diversos processadores em um único, criando os chamados processadores *multicore*. Esse processo deu início à popularização

das arquiteturas paralelas entre os usuários domésticos e pequenos laboratórios. Em complemento, sabe-se que os fenômenos naturais são inerentemente paralelos, logo, é natural expressar as computações relativas aos mesmos de forma paralela, pois, em algumas situações, a ordem de execução é importante para o melhor entendimento do problema real, mesmo que em outras ela seja irrelevante [SKI98]. Assim, a adaptação das soluções da Computação Verificada para esse novo nicho de arquiteturas é de grande interesse. Nesse contexto, o presente trabalho foi concebido visando desenvolver um *solver* (ferramenta computacional para resolução) de SELAs Intervalares Densos de Grande Porte que execute a verificação automática dos resultados e seja otimizado para execução escalável em arquiteturas com processadores *multicore*.

1.1 Trabalhos Relacionados

Existem diversos trabalhos que combinam a Computação Verificada e a Computação Paralela na resolução de SELAs para, respectivamente, gerar resultados validados e obter um tempo de execução não impeditivo. Alguns desses, como [KOL07] e [KOL08b], operam Sistemas Lineares do tipo pontual enquanto outros, como [KOL08c], resolvem Sistemas Lineares Intervalares. Duas ferramentas estado da arte para Sistemas Intervalares Densos de Grande Porte são apresentadas em [KOL08c] e em [KRA09]. Em ambos os casos os autores obtiveram resultados satisfatórios, tanto em relação à qualidade das soluções computadas quanto ao tempo de execução das ferramentas em agregados de computadores.

O trabalho desenvolvido em [KRA09] apresenta uma nova abordagem para o cálculo do **produto escalar ótimo**, algoritmo no qual se baseia o C-XSC. Os autores apresentam também uma versão da solução paralelizada para agregados de computadores. Seu resultado final é um *solver* voltado a Sistemas Lineares Intervalares Densos de Grande Porte. Já o trabalho desenvolvido em [KOL08c] emprega a Aritmética Intervalar do tipo Ponto-Médio e Raio e um algoritmo baseado no Método de *Newton*, o qual será visto no Capítulo 3, pra resolução de Sistemas Intervalares Densos. Assim como em [KRA09], os autores utilizaram a PBLAS e ScaLAPACK para implementação de uma versão paralela da ferramenta voltada a ambientes heterogêneos com troca de mensagens.

1.2 Motivação

A necessidade original de alto desempenho em Computação Numérica veio de uma série de contextos envolvendo equações diferenciais parciais, como dinâmica de fluidos, previsão do tempo, processamento de imagens entre outros. Porém, em decorrência dos grandes avanços da computação nos últimos anos, mesmo para problemas menores os usuários têm se tornado mais exigentes desejando simular modelos maiores, mais precisos e multidimensionais, além de integrar problemas que antes eram simulados separadamente [BUT07]. As EDs e os SELAs estão entre as ferramentas matemáticas mais utilizadas nesses casos e sua resolução em tempo adequado somente é viável com ferramentas da Computação Paralela.

Sabe-se hoje que muitas das estratégias criadas para otimização dos algoritmos da Álgebra Linear Densa, como no LAPACK, por exemplo, levaram a ideias e mecanismos que influenciaram positivamente outras categorias de algoritmos paralelos. Portanto, o desenvolvimento de projetos combinando problemas da Álgebra Linear (como a resolução de SELAs) com Computação Paralela é um trabalho com motivações em ambas as disciplinas. Por um lado, a Computação Paralela contribui com as mais variadas áreas do conhecimento, pois permite executar simulações em tempo viável e, por outro, essas áreas apresentam situações que levam a algoritmos genéricos o bastante a ponto de influenciar positivamente diversas outras aplicações da Computação Paralela.

A aplicação da Computação Verificada em tais algoritmos tem duas motivações principais. A primeira é controlar os erros de arredondamento, de modo a oferecer resultados corretos e confiáveis. A segunda é permitir a resolução de Sistemas Lineares Intervalares visando dar suporte a problemas que operam com dados incertos e/ou imprecisos. Por fim, a proposta de uma solução que explore o paralelismo em processadores *multicore* justifica-se não apenas pelo contexto mercadológico atual, mas também por suas tendências futuras. A grande popularização das arquiteturas *multicore* torna a ferramenta acessível para uma infinidade de públicos, diferentemente daquelas desenvolvidas para aglomerados de computadores. Ademais, mesmo os aglomerados atuais vêm sendo construídos utilizando processadores *multicore* em seus nós. Nesse contexto, a exploração do paralelismo no nível atual não é suficiente para tirar proveito de todas as características das novas máquinas, demonstrando a necessidade de aplicar técnicas específicas de programação e, ao mesmo tempo, mesclá-las com as estruturas de *softwares* já existentes.

1.3 Objetivos

O objetivo central do presente trabalho é **desenvolver uma ferramenta computacional para resolução de SELAs Intervalares Densos de Grande Porte com verificação automática dos resultados otimizada para execução em arquiteturas *multicore***. Essa ferramenta deverá ser capaz de prover resultados verificados para Sistemas Lineares em que os dados contidos nas matrizes e vetores são intervalos ao invés de números pontuais. Tal implementação deverá tirar proveito das características presentes nas arquiteturas dotadas de processadores *multicore* visando ganhos de desempenho de modo que seu tempo de execução seja satisfatório para o usuário.

Objetivos secundários aparecem como consequência da metodologia adotada, tais como: investigar a resolução de SELAs Densos de Grande Porte; realizar estudos comparativos da Aritmética Intervalar; estudar as arquiteturas de memória compartilhada; explorar a programação *multithread*; e estudar as bibliotecas mais populares para Álgebra Linear de Alto Desempenho.

1.4 Organização do Trabalho

Esta Dissertação está dividida em seis capítulos, sendo o primeiro deles a presente introdução. O restante está organizado da seguinte forma:

- **Capítulo 2**: contextualiza as arquiteturas *multicore*. Discutem-se as tendências que deverão nortear a programação de tais processadores nos próximos anos e a implementação de algoritmos para Álgebra Linear de Alto Desempenho. Por fim, algumas das bibliotecas mais utilizadas para essa finalidade são apresentadas;

- **Capítulo 3**: apresenta uma visão geral sobre a Computação Verificada, suas bases, funcionamento e aplicação na resolução de SELAs. A Aritmética Intervalar é também revisada neste capítulo;

- **Capítulo 4**: descreve a solução desenvolvida neste trabalho, ou seja, a ferramenta para resolução verificada de SELAs em *multicore*;

- **Capítulo 5**: demonstra as avaliações de exatidão e de desempenho realizadas para validar o *solver* desenvolvido e as estratégias adotadas para sua otimização;

- **Capítulo 6**: conclui o presente trabalho e apresenta propostas de trabalhos futuros.

2. ÁLGEBRA LINEAR EM ARQUITETURAS *MULTICORE*

A compreensão do inter-relacionamento entre as características de um algoritmo paralelo e aquelas da arquitetura sobre a qual ele irá executar é fundamental para a obtenção de alto desempenho. Nas três últimas décadas, diversos algoritmos e *softwares* para resolução de problemas da Álgebra Linear têm sido desenvolvidos visando alto desempenho e portabilidade. O resultado mais importante neste contexto foi o desenvolvimento da biblioteca BLAS (*Basic Linear Algebra Subprograms*) [LAW79a], que é utilizada como núcleo para desenvolvimento dos mais populares pacotes de *software* para Álgebra Linear [DON00].

Neste capítulo, serão inicialmente abordados aspectos gerais das arquiteturas dotadas de processadores *multicore*. Na sequência, discute-se a Álgebra Linear de Alto Desempenho focando sua programação em arquiteturas *multicore*. Por fim, a biblioteca BLAS e os pacotes mais populares para Álgebra Linear de Alto Desempenho são apresentados.

2.1 Arquiteturas *Multicore*

O paralelismo, atualmente, está presente em todos os níveis da Computação indo desde o paralelismo de instruções em um *pipeline*, passando pelos processadores *multicore*, colocados como solução para o problema de limitação no crescimento do *clock*, até sistemas distribuídos de larga escala como as grades computacionais. Nos últimos 20 anos, os fabricantes de microprocessadores exploraram altos graus de *Instruction Level Parallelism* (ILP). Baseando-se nisso, muitas gerações de processadores foram construídas aumentando-se cada vez mais a frequência do *clock* e com *pipelines* cada vez mais profundos. As aplicações beneficiavam-se naturalmente dessas inovações e, como consequência, para atingir maior desempenho bastava simplesmente delegar aos compiladores a exploração eficiente do ILP.

Porém, devido a vários fatores, em especial às limitações físicas, tornou-se necessário alterar o foco do paralelismo do ILP para o TLP (*Thread Level Parallelism*). Nesse, o ganho de desempenho é alcançado através da replicação das unidades de execução (*cores*) ao passo que se mantêm os *clocks* em uma faixa na qual o consumo de energia e dissipação de calor são menores e, portanto, menos problemáticos [BUT08]. Ao longo dos anos, diversas foram as tentativas de desenvolver compiladores paralelizantes. Entretanto, esses se demonstraram úteis apenas em algumas classes restritas de problemas. Assim, no contexto dos computadores *multicore*, não é possível fundamentar-se apenas nos compiladores para obter ganhos de desempenho. Faz-se necessário reescrever as aplicações de modo a tirar proveito do paralelismo de granularidade fina.

Embora a obsolescência dos paradigmas tradicionais esteja ocorrendo de maneira bastante rápida, não há, ainda, uma alternativa bem compreendida a qual possa ser utilizada como base única para os desenvolvedores. Diferentemente das premissas adotadas em modelos anteriores, os núcleos de um processador *multicore* não podem ser considerados independentemente, ou seja, arquiteturas

multicore não devem ser vistas como novas SMP (*Symmetric Multiprocessor*). Isso porque, tais núcleos compartilham recursos *intra-chip* (inclusive múltiplos *caches* e TLB (*Translation Lookahead Buffer*), o que não ocorria com os múltiplos processadores independentes [BUT08], [CHA08].

Essa situação tende a ser agravada considerando que se espera, futuramente, ter uma variedade de combinações de componentes, como a mistura de diferentes núcleos, aceleradores de *hardware* e sistemas de memória diversificados. Os autores de [BUT08], [CHA08], [CHA07a], [CHA07b] apontam alguns fatores como causas desse aumento:

- "Mais transistores e *clocks* mais lentos": tal abordagem tende a elevar o número de *cores* nos processadores e, por consequência, a quantidade de paralelismo requerido. Modelos baseados em *pipelines* profundos e estreitos tendem a perder espaço para os *designs* baseados em *pipelines* rasos e amplos. Com isso, para extrair desempenho dos *multicores*, os programadores terão de explorar um nível maior de paralelismo, ou seja, explorá-lo em nível de múltiplos *threads* (*thread-level parallelism* (TLP)). Serão necessários, para tal, mecanismos mais eficientes de comunicação entre os processadores e de sincronização para melhorar o gerenciamento dos recursos. Dessa forma, a abordagem adotada pelos *designs* superescalares, na qual a complexidade do paralelismo era mascarada em *hardware* por uma combinação de paralelismo crescente em nível de instrução (ILP) e *pipelines* profundos e estreitos, deixa de ser satisfatória. Necessita-se, portanto, que o paralelismo seja explorado em *software*.

- "Muro de memória mais espesso": a eficiência na comunicação tende a se tornar cada vez mais essencial. Os pinos que conectam processador e memória principal transformaram-se em um gargalo devido ao crescimento do número de pinos e à queda da largura de banda por pino. Assim, estima-se que a lacuna de desempenho existente entre processador e memória, a qual já é de aproximadamente mil ciclos, cresça em torno de 50% por ano. Adicionalmente, estima-se que o número de *cores* em um único *chip* duplicará a cada 18 meses. Uma vez que as limitações de espaço físico inibirão o crescimento dos recursos de *cache* na mesma proporção, a quantidade de *cache* por *core* tende a diminuir cada vez mais. Com isso, pode-se esperar aumento significativo em problemas como largura de banda da memória, latência de memória e fragmentação de *cache*.

- "Limitações dos processadores *commodity* devem aumentar a heterogeneidade e complexidade dos sistemas": sabe-se que, por motivos econômicos, os sistemas *petascale* serão construídos com processadores *commodity of-the-shelf* (de prateleira). Infelizmente, as arquiteturas de propósito geral não são capazes de atingir as características requeridas pelas aplicações de pesquisa de ponta. Consequentemente, em adição aos diferentes níveis de *multithreading* que os sistemas *multicore* deverão explorar (nível de *core*, de *socket*, de placa e nível de memória distribuída), os mesmos deverão, também, incorporar uma diversidade de elementos de processamento para uso específico, tais como GPUs (*Graphics Processing Unit*), FPGAs (*Field Programmable Gate Array*), dentre outros. Tal heterogeneidade incrementará ainda

mais complexidade de controle e programação. Uma vez que os *designs* oferecidos hoje pelos fabricantes já divergem entre si e que configurações de *hardware* heterogêneas já podem ser facilmente encontradas, não se pode esperar obter uma categoria de arquitetura comum para a qual desenvolver os modelos de programação futuros.

Embora ainda não se tenha claro quais as estratégias futuras que os fabricantes utilizarão para manter o crescimento do número de núcleos, é possível identificar algumas propriedades que os algoritmos deverão apresentar para alcançar níveis mais altos de TLP:

- Granularidade fina: os *cores* são e, provavelmente, continuarão sendo associados com memórias locais relativamente pequenas. Assim, deve-se reorganizar as operações em tarefas que operem sob pequenas porções de dados para reduzir o tráfego no barramento e aumentar a localidade de dados.

- Não sincronismo: uma vez que o grau de TLP cresce e a granularidade diminui, a presença de pontos de sincronização em uma execução paralela afeta seriamente a eficiência de um algoritmo. Deve-se, portanto, evitá-los.

2.2 Álgebra Linear de Alto Desempenho

Embora a descontinuidade introduzida pelas novas arquiteturas vá ser ubíqua, diversos autores como [BUT09] argumentam que há razões para focar-se os esforços em problemas da Álgebra Linear em geral e da Álgebra Linear Densa dada sua importância em diversas aplicações da Ciência da Computação. Além disso, argumentam que, como tais problemas já são bastante conhecidos e estudados, tem-se um embasamento que cria vantagem estratégica para os esforços de pesquisa, uma vez que já se sabe exatamente como seus algoritmos funcionam e onde podem ser alterados. Outro ponto exposto por tais autores é que as técnicas de Álgebra Linear são genéricas o suficiente para que os ganhos obtidos com seus estudos possam ser, posteriormente, aplicados a outras áreas, como ocorreu no desenvolvimento do LAPACK.

De acordo com [BUT09] a exploração eficiente do paralelismo nas operações da Álgebra Linear em arquiteturas *multicore* apresenta dois requisitos básicos: granularidade fina e operações assíncronas. De fato, as técnicas que vêm apresentando melhores resultados em tais arquiteturas são a exploração do paralelismo em nível de tarefas, redução dos TLB (*translation lookahead buffer*) *misses* [GOT08] e escalonamento dinâmico de operações com execução fora de ordem, ou seja, de maneira semelhante a um *pipeline* superescalar [CHA08], [SON09]. Adicionalmente, é um fato bem estabelecido que a técnica de execução com adiantamento/antecipação de operações (*look ahead*), possível devido à execução fora de ordem, pode ser utilizada para aumentar significativamente o desempenho na fatoração de matrizes [KUR07].

Sabe-se, também, que outro fator de grande importância na Computação Paralela é o particionamento dos dados. Determinar o esquema mais adequado para divisão dos dados em um algoritmo influencia diretamente o desempenho do programa paralelo. Nos cálculos com matrizes os dois

métodos mais utilizados são o particionamento por faixas (*striped partitioning*) e o particionamento por submatrizes (*checkerboard partitioning*). Tais métodos subdividem-se ainda de duas maneiras: distribuição por blocos e distribuição cíclica. As figuras 2.1, 2.2, 2.3 e 2.4 ilustram essas o funcionamento dessas distribuições.

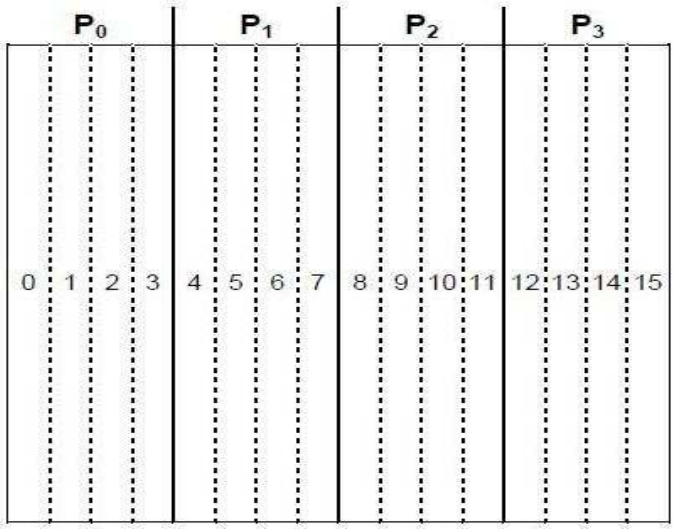

Figura 2.1: Particionamento por Blocos de Colunas (*Block-Striped Partitioning*). Adaptado de [KUM02]

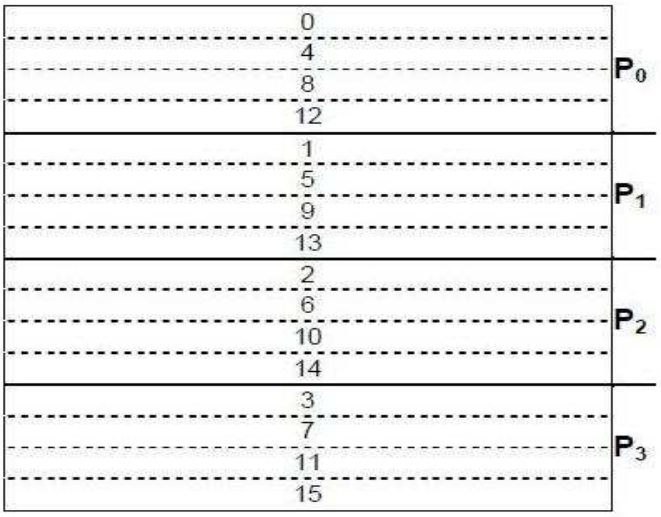

Figura 2.2: Particionamento Cíclico de Linhas (*Cyclic-Striped Partitioning*). Adaptado de [KUM02]

No desenvolvimento do LINPACK não houve preocupação com a movimentação dos dados pela memória. Já no LAPACK e ScaLAPACK organizou-se os algoritmos visando que os mesmos se beneficiassem da hierarquia de memória ao diminuir o máximo o movimento dos dados por ela. Utilizou-se, para tal, o particionamento por submatrizes com distribuição cíclica. A Figura 2.5 ilustra uma divisão típica da hierarquia de memória nos computadores modernos. Em [GOT02] os autores afirmam que um nível intermediário, o espaço de endereçamento acessível ao TLB, deve ser considerado entre os *caches* L1 e L2.

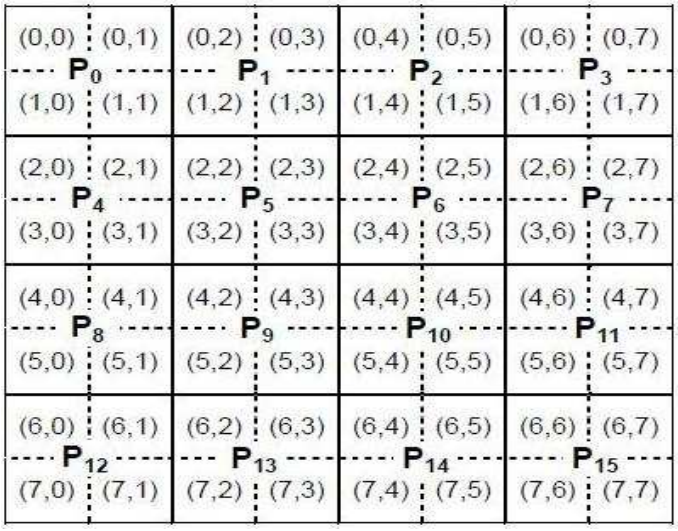

Figura 2.3: Particionamento por Blocos de Sub-Matrizes (*Block-Checkboard Partitioning*). Adaptado de [KUM02]

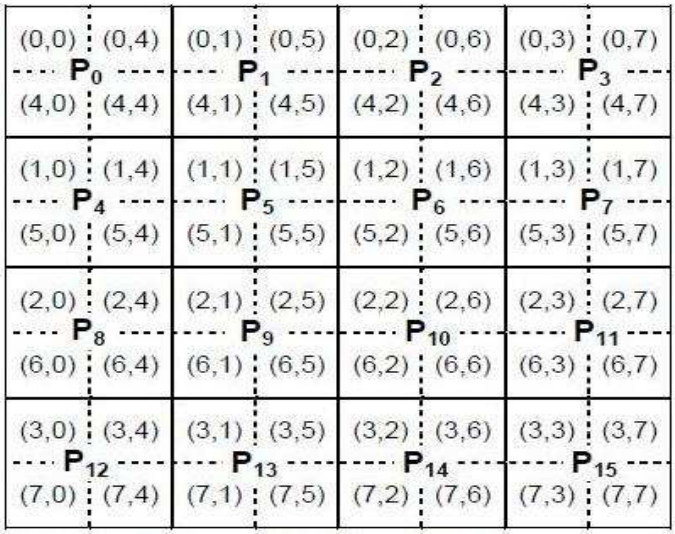

Figura 2.4: Particionamento Cíclico de Sub-Matrizes (*Cyclic-Checkboard Partitioning*). Adaptado de [KUM02]

Em arquiteturas *multicore*, no entanto, a literatura vem apontando que o particionamento por submatrizes com distribuição cíclica não é o método mais eficiente. A maior limitação para execução das computações de granularidade fina é que a BLAS geralmente tem desempenho bastante ruim para blocos pequenos de dados. Essa situação pode ser melhorada armazenando-se as matrizes em pequenos blocos ao invés de utilizar o *column major format*, formato padrão de armazenamento do FORTRAN, e, então, cada bloco contiguamente no formato *column major*. Como resultado tem-se um padrão mais regular de acesso à memória e o desempenho da BLAS é consideravelmente melhorado [BUT08].

Em [CHA07a] os autores introduzem o sistema de escalonamento dinâmico *SuperMatrix*. A principal proposta desse é que as matrizes sejam vistas como compostas por um conjunto de submatrizes, ou seja, particionadas em submatrizes por blocos. Assim, os dados são reorganizados

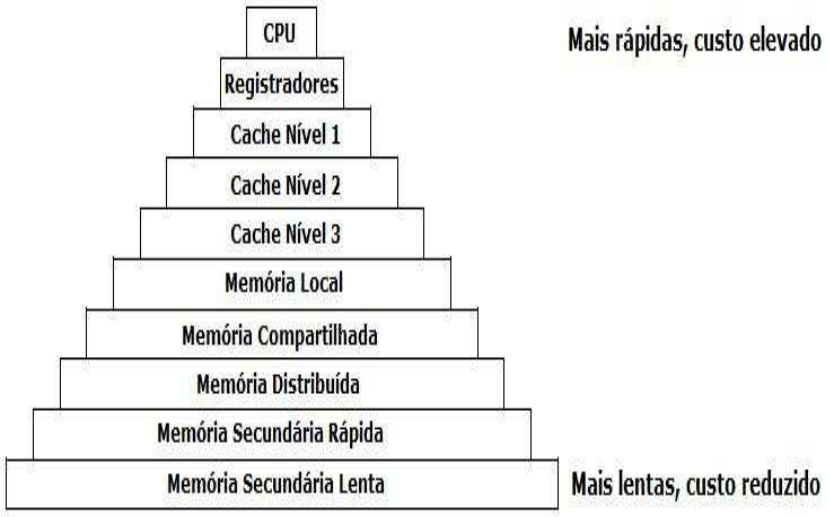

Figura 2.5: Hierarquia de Memória nos Computadores Modernos. Adaptado de [DON03]

em estruturas de modo que os blocos de submatrizes são tratados como unidades fundamentais de dado (equivalente aos escalares). As operações, por sua vez, são reorganizadas de modo a serem vistas como unidades fundamentais de computação (tarefas). De acordo com os autores, isso reduz fortemente a complexidade no gerenciamento das dependências de dados permitindo a utilização de **algoritmos por blocos** ao invés dos tradicionais **algoritmos blocados**. A continuidade do projeto *SuperMatrix*, bem como alguns exemplos de sua aplicação com resultados bastante satisfatórios, são apresentados em [CHA07b] e [CHA08]. Uma opção para facilitar a manipulação das matrizes constituídas por blocos de submatrizes é a FLASH API [LOW04].

Os autores da PLASMA, descrita no próximo capítulo, vêm seguindo as mesmas estratégias anteriormente citadas bem como a abordagem do *SuperMatrix*. Em [BUT09] os autores apoiam a ideia de que a quebra das operações em pequenas tarefas reduz o tráfego no barramento e tira melhor proveito da localidade de dados. Em relação ao escalonamento fora de ordem e o não sincronismo, os autores observam que isso permite amenizar o problema da latência no acesso à memória. Adicionalmente, é apresentada a necessidade de reformular os algoritmos utilizados pelo LAPACK e ScaLAPACK e coloca-se como alternativa a utilização de DAGs (*Direct Acyclic Graph*) onde os nodos representam tarefas e as arestas as dependências entre elas.

Um grafo acíclico dirigido é um grafo dirigido sem ciclos, isto é, para qualquer vértice v, não há nenhum caminho dirigido começando e acabando em v. Seja $G = (N, A)$ um DAG, onde $N = \{1, \ldots, N\}$ é o conjunto de nós e A é o conjunto dos arcos dirigidos, cada nó representa uma operação a ser feita pelo algoritmo e os arcos representam as dependências de dados. Em particular, um arco $(i, j) \in A$ indica que a operação correspondente ao nó j usa o resultado da operação correspondente ao nó i. As operações podem tanto ser elementares, como a adição de dois escalares, quanto operações de alto nível, como a execução de uma subrotina.

Os algoritmos propostos em [BUT09] recebem o nome de *tiled algorithms*. A divisão dos dados em pequenos blocos quadrados de colunas contíguas é defendida, ou seja, uma divisão por subma-

trizes em blocos. Por fim, os autores atentam para o balanceamento de carga e de tarefas os quais devem ser cuidadosamente implementados de modo que a escalabilidade não seja comprometida.

O trabalho apresentado em [CHA08] aplica a Computação Paralela reorganizando o código e dados de forma a tirar proveito das novas arquiteturas paralelas como NUMA (*Non-Uniform Memory Access*) dotadas de processadores *multicore*. Nesse caso o autor desenvolveu um sistema que paraleliza operações sob matrizes para arquiteturas *multicore* considerando-as como blocos hierárquicos que servem como unidades de dados sob as quais as operações, vistas como unidades de computação, são executadas. Assim, a implementação enfileira de modo transparente as operações requisitadas e verifica internamente as dependências para, então, executá-las na melhor ordem possível. Tal ideia assemelha-se aos *pipelines* de micro arquiteturas superescalares. Esse trabalho e, principalmente, o desenvolvimento da biblioteca PLASMA confirmam a tendência mundial de desenvolvimento de *softwares* para Álgebra Linear de Alto Desempenho voltados aos novos nichos de arquiteturas.

2.3 Bibliotecas e Pacotes de *Software*

Esta seção apresenta, inicialmente, a biblioteca BLAS. Após, alguns dos pacotes de *software* para Álgebra Linear construídos com base nessa são abordados.

2.3.1 *Basic Linear Algebra Subprograms* - BLAS

A BLAS (*Basic Linear Algebra Subprograms*) surgiu como resultado de um acordo para especificar um conjunto de operações básicas de Álgebra Linear que foi implementado originalmente em FORTRAN 66 e, após, FORTRAN 77. Com o passar dos anos, implementações de referência para FORTRAN 95 e C foram publicadas. A BLAS se tornou a biblioteca de Álgebra Linear mais utilizada mundialmente, pois, além de seu bom desempenho, o fato de desenvolver *softwares* em torno de um núcleo comum é visto como uma boa prática de Engenharia de *Software* [BLA02]. A BLAS original (hoje conhecida como *Level* 1 BLAS) executa operações entre escalares e vetores. Sua descrição completa está disponível em [LAW79a] e complementada por [LAW79b].

Com o surgimento das máquinas vetoriais, das máquinas com hierarquia de memória e das arquiteturas de memória compartilhada, surgiu a necessidade de *softwares* para exploração adequada de tais arquiteturas. Assim, uma extensão da BLAS original (hoje conhecida como *Level* 2 BLAS) foi proposta em [DON88b] para execução de operações entre vetores e matrizes. Em [DON88a] os autores apresentam um implementação modelo da *Level* 2 BLAS em FORTRAN 77 e também um conjunto de programas para teste. Entretanto, frequentemente a *Level* 2 BLAS não se adaptava bem aos computadores com hierarquia de memória cujo processamento era de fato paralelo. Nesse contexto, foi proposta em [DON90] a *Level* 3 BLAS para execução de operações de matrizes com matrizes. Tais operações são executadas em blocos permitindo, assim, reuso dos dados enquanto um bloco encontra-se na *cache* ou na memória local, o que evita movimentação excessiva dos dados pela hierarquia de memória [DON00], [DON90], [BLA02].

Como exemplo das operações executadas pela *Level* 1 BLAS tem-se o cálculo do produto interno de um vetor. Tais operações envolvem complexidade de dados da ordem $O(n)$ e custo computacional também da ordem de $O(n)$. Na *Level* 2 BLAS, as operações envolvem custo computacional de $O\left(n^2\right)$ com complexidade de dados também de $O\left(n^2\right)$. Um exemplo de operação realizada neste nível é o produto entre um vetor e uma matriz. Por fim, na *Level* 3 BLAS, a complexidade de dados é de $O\left(n^2\right)$ e o custo computacional da ordem de $O\left(n^3\right)$. Como exemplo, pode-se citar a multiplicação de duas matrizes [DON00]. As ordens de complexidade das operações são o fator responsável pela adoção dos nomes *Level* 1, 2 e 3 [DON90].

No contexto do presente trabalho, a *Level* 3 BLAS é a de maior interesse. Esse nível possui 81 diferentes combinações as quais se tratam tanto de rotinas em alto nível da Álgebra Linear quanto de operações elementares para construção dessas rotinas. Os algoritmos da *Level* 3 BLAS foram originalmente implementados para auxiliar o desenvolvimento de procedimentos em termos de operações em submatrizes ou blocos. Diversos trabalhos comprovaram a efetividade dos algoritmos em blocos da *Level* 3 BLAS para uma variedade de arquiteturas, nas quais o desempenho decai fortemente com o excesso de movimentação dos dados na hierarquia de memória. As *Level* 2 e *Level* 3 BLAS armazenam as matrizes na memória na forma de vetores bi-dimensionais [DON88b], [DON90].

A implementação de referência da BLAS está disponível livremente na Internet. Entretanto, a maioria dos fabricantes de computadores oferece versões proprietárias dessas rotinas otimizadas para suas arquiteturas. Essas, em alguns casos, encontram-se disponíveis gratuitamente enquanto em outros é necessário adquirir licenças para utilização. Dentre as mais populares pode-se citar: *AMD Core Math Library* - ACML - implementação *multithread* da AMD disponibilizada gratuitamente e otimizada para execução em processadores *Opteron*; *Apple Velocity Engine* - versão da *Apple* embarcada nos processadores G4 e G5 que expande a arquitetura *PowerPC* através da adição de uma unidade de execução vetorial de 128 *bits* que opera paralelamente dados inteiros e de ponto flutuante. No momento em que os computadores *Apple* passaram a ser construídos com processadores *Intel*, tal tecnologia foi incorporada pelo recurso *Intel's Streaming SIMD Extensions* (SSE); *HP's Mathematical Software Library* - MLIB - implementação da HP disponível para os sistemas HP-UX, desde servidores com um único processador até os mais robustos com múltiplos processadores como *Superdome*, otimizada para os processadores HP PA-RISC 2.0 e *Intel Itanium* 2; *Intel Math Kernel Library* - MKL - implementação da *Intel* otimizada para arquiteturas dotadas de processadores *Xeon*, *Core* i7, *Itanium*, *Pentium* e família *Core* em geral. Possui implementações sequenciais das rotinas, altamente otimizadas apenas em nível de instruções, e versões *multithread*.

Além disso, existem ainda disponíveis gratuitamente na Internet diversas implementações alternativas da BLAS construídas sob diferentes abordagens tais como a ATLAS (*Automatically Tuned Linear Algebra Software*) [WHA04], [WHA98] e a Goto BLAS [GOT02]. Tanto a Goto BLAS quanto a ATLAS e a grande maioria das implementações proprietárias da BLAS emprega recursos de *multithread*. Cabe, ainda, observar a existência de uma versão paralela da BLAS para agregados de computadores, a PBLAS (*Parallel Basic Linear Algebra Subprograms*) [CHO95b]. Surgida inicialmente como PB-BLAS (*Parallel Block Basic Linear Algebra Subprograms*) [CHO94], a PBLAS está

disponível para qualquer sistema com suporte a MPI (*Message Passing Interface*) [SNI96] ou PVM (*Parallel Virtual Machine*) [GEI94]. A PBLAS foi construída utilizando como base a versão sequencial da BLAS e a biblioteca BLACS (*Basic Linear Algebra Communication Subprograms*) [DON97], uma biblioteca desenvolvida para facilitar o desenvolvimento de programas de Álgebra Linear em ambientes com troca de mensagens. Os trabalhos relacionados apresentados no início desta Dissertação foram implementados com a PBLAS. Entretanto, como o presente trabalho é focado em processadores *multicore*, somente versões sequenciais e *multithread* da BLAS tradicional foram empregadas.

2.3.2 LINPACK, LAPACK, ScaLAPACK e PLASMA

A seguir, são apresentados os pacotes de *software* mais populares para Álgebra Linear de Alto Desempenho. Tais pacotes são implementados com emprego da BLAS e foram desenvolvidos ao longo das últimas décadas de modo a acompanhar as constantes evoluções das novas arquiteturas de *hardware*.

LINPACK

O LINPACK é um pacote criado para os supercomputadores utilizados nos anos 70 e começo dos anos 80. Consiste em uma coleção de programas para resolução de SELAs. Suas rotinas são escritas em FORTRAN e resolvem os sistemas através da abordagem decomposicional da Álgebra Linear, ou seja, dada uma matriz A, decompõe-se a mesma em um produto de outras matrizes mais simples e bem estruturadas, as quais podem ser facilmente manipuladas para resolver o problema original. Foi desenvolvido de modo a executar as operações de ponto flutuante através de chamadas as rotinas da *Level* 1 BLAS [DON03]. Os algoritmos são orientados a colunas, ou seja, as matrizes são sempre referenciadas por colunas e não por linhas, objetivando aumentar a eficiência ao preservar a localidade de dados. Isso ocorre porque o FORTRAN armazena as matrizes por colunas. Assim, ao acessar uma coluna de uma matriz, as referências à memória serão sequenciais [DON79].

Atualmente, o LINPACK é mais conhecido enquanto *benchmark* do que biblioteca. O *Benchmark* LINPACK foi originalmente concebido para fornecer aos usuários da biblioteca LINPACK informações sobre os tempos de execução necessários para resolução dos SELAs. A primeira aparição do LINPACK como *benchmark* foi em 1979. Com o passar dos anos foi recebendo incrementos e, nos dias de hoje, é composto por três *benchmarks*. O mais importante desses é o *Highly Parallel Computing Benchmark* (HPL), pois serve como medida para elaboração da lista dos 500 computadores mais rápidos do mundo [TOP10]. O método usado no *benchmark* é decomposição LU com pivotamento parcial, a matriz é do tipo densa, composta por elementos inteiros distribuídos aleatoriamente entre -1 e 1. A resolução do Sistema de Equações requer $O(n^3)$ operações de ponto flutuante, mais especificamente $2/3n^3 + 2n^2 + O(n)$ adições e multiplicações de ponto flutuante [DON79].

Dentre as técnicas para melhora do desempenho na resolução dos SELAs, duas se destacam no LINPACK: desenrolamento dos laços e reuso dos dados. Observa-se que, frequentemente, o maior

volume de computação de um programa está localizado em menos de 3% do código fonte. Essa porcentagem do código, também chamada de código crítico, consiste, em geral, em um ou alguns poucos laços de repetição imersos, ou seja, em um nível maior de aninhamento. O desenrolamento do laço consiste em replicar seu conteúdo, fazendo os devidos ajustes, o que aumenta o desempenho porque causa uma diminuição direta dos *overheads* inerentes ao *loop*. Nas máquinas com instruções vetoriais, no entanto, essa técnica tem o efeito oposto [DON79].

Em relação ao reuso de dados, sabe-se que a cada passo do processo de fatoração do LINPACK são feitas operações vetoriais para modificar uma submatriz inteira dos dados. Essa atualização faz com que um bloco de dados seja lido, atualizado e escrito novamente na memória central. O número de operações de ponto flutuante é $2/3n^3$ e o número de referências a dados é, em ambos os casos (leitura e escrita), de $2/3n^3$. Assim, para cada par adição/multiplicação é feita a leitura e escrita dos elementos, levando a um baixo reuso dos dados. Mesmo quando as operações são vetoriais, existe um gargalo significante na movimentação dos dados, o que resulta em desempenho ruim nas máquinas modernas. Em máquinas vetoriais, isso se traduz em duas operações de vetor e três referências vetoriais á memória. Nos computadores superescalares isso resulta em uma grande movimentação e atualização dos dados. Esse contexto faz com que o LINPACK tenha desempenho reduzido em computadores de alto desempenho nos quais o custo do movimento dos dados é semelhante ao das operações de ponto flutuante. Uma possível solução é reestruturar os algoritmos de modo que explorem a hierarquia de memória das arquiteturas, o que pode ser feito, por exemplo, armazenando os dados o maior tempo possível nas memórias de nível mais próximo do processador, ou seja, aumentando o reuso dos dados [DON79]. Tais otimizações, presentes nos níveis 2 e 3 da BLAS, foram posteriormente adotados por outros pacotes como o LAPACK.

LAPACK – *Linear Algebra PACKage*

O LAPACK foi desenvolvido no final dos anos 80 visando permitir as já amplamente utilizadas bibliotecas EISPACK (*Eigensystem Package*) [SMI76] e LINPACK rodarem eficientemente em computadores paralelos de memória compartilhada e vetoriais. Em tais máquinas, esses pacotes são ineficientes porque seus padrões de acesso à memória negligenciam a hierarquia de memória, o que torna o custo do acesso aos dados seja bastante alto. De modo a contornar esse problema, os algoritmos no LAPACK foram reorganizados para utilizar as operações de matrizes em bloco, tais como multiplicação de matrizes, nos laços mais internos. Essas operações em bloco podem ser otimizadas para cada arquitetura de modo a tirar proveito da hierarquia de memória e prover uma forma de se atingir alta eficiência nas diversas máquinas modernas [DON90], [AND99], [AND90], [DEM89].

Escrito em FORTRAN 77, o LAPACK provê subrotinas para resolução de Sistemas de Equações Lineares, problemas de Auto-Valores, dentre outros. As fatorações de matrizes disponíveis são LU, *Cholesky*, QR, SVD, *Schur* e *Schur* Generalizada. São suportadas matrizes densas e do tipo banda com elementos reais ou complexos, porém, matrizes esparsas não são. Em relação ao EISPACK e LINPACK obteve-se melhorias em quatro aspectos principais: velocidade, exatidão, robustez e funcionalidades [DON00]. Considerando-se seu nicho de arquiteturas, o LAPACK é, ainda hoje, o

programa estado da arte para resolução de problemas de equações densas e do tipo banda, além de outros tipos de operações da Álgebra Linear [BLA02].

Enquanto o LINPACK e EISPACK são baseados nas operações vetoriais da BLAS (nível 1), o LAPACK explora o nível 3, tendo, inclusive, influenciado posteriormente o desenvolvimento desse nível. Tal influência deve-se ao fato de que algumas operações da BLAS somente passaram a ser utilizadas com maior frequência e foram, portanto, implementadas como rotinas separadas após a implementação do LAPACK. Exemplos dessas são copiar uma matriz (*GE_COPY*) e calcular a norma de uma matriz (*GE_NORM*), dentre outras [BLA02]. Devido à granularidade grossa do nível 3 das operações da BLAS, seu uso provê alta eficiência em muitos computadores de alto desempenho, principalmente naqueles em que implementações otimizadas são oferecidas pelo fabricante da máquina. Alguns anos mais tarde, com o advento dos agregados de computadores, surgiu o ScaLAPACK [GUN01], uma versão paralela do LAPACK para máquinas de memória distribuída.

ScaLAPACK – *Scalable Linear Algebra PACKage*

O desenvolvimento do ScaLAPACK foi iniciado em 1991 e sua primeira publicação ocorreu no final de 1994. Surgiu com objetivo de estender o LAPACK para execução escalável nas arquiteturas paralelas de memória distribuída, uma vez que nessas a hierarquia de memória inclui, além da hierarquia de registradores, *cache* e memória local de cada processador, a memória externa dos outros processadores. O ScaLAPACK suporta matrizes densas e do tipo banda. Sua implementação baseia-se no paradigma de programação SPMD *Single-Program-Multiple-Data* empregando, para tal, troca explícita de mensagens em redes com suporte a PVM e/ou MPI. São utilizados os níveis 1, 2 e 3 da versão paralela da BLAS, PBLAS, e a BLACS, anteriormente apresentadas.

Assim como no LAPACK, as rotinas do ScaLAPACK empregam algoritmos que operam em blocos. Tais algoritmos assumem que as matrizes são formadas por decomposição cíclica em blocos bi-dimensionais [DON00]. Suas rotinas foram mantidas, sempre que possível, compatíveis com suas equivalentes no LAPACK. Dessa forma, em códigos alto nível as chamadas LAPACK e ScaLAPACK são bastante semelhantes, facilitando os esforços de implementação por parte dos usuários [CHO96].

Para atingir escalabilidade, além do particionamento em blocos de tamanhos ajustáveis dos algoritmos, o ScaLAPACK possui diversos algoritmos equivalentes para um mesmo cálculo. Com isso, escolhe-se em tempo de execução o melhor algoritmo para uma dada entrada ou arquitetura. Por fim, sabe-se que o modo como os dados são distribuídos pelos processos tem grande impacto no balanceamento de carga e nos custos da comunicação. A distribuição cíclica em blocos provê um mecanismo simples para distribuição de dados com algoritmos particionados em blocos em arquiteturas de memória distribuída sendo, por isso, utilizada pelo ScaLAPACK [CHO95a], [CHO96].

PLASMA – *Parallel Linear Algebra for Scalable MultiCore Architectures*

A abordagem clássica da Álgebra Linear de Alto Desempenho adotada pelos pacotes anteriores consiste em explorar o paralelismo oferecido pelas versões otimizadas da BLAS e PBLAS. No entanto,

tem-se observado que limitar o uso da memória compartilhada ao paralelismo *fork-join*, como ocorre com OpenMP, ou à utilização de versões *multithread* da BLAS não é suficiente para tratar todas as questões de desempenho em computadores *multicore*. Diversos são os relatos na literatura, como por exemplo [BUT07], de que, mesmo vinculando-se o LAPACK a versões *multithread* altamente otimizadas da BLAS, suas rotinas não exploram de maneira adequada os processadores *multicore*. Isso ocorre porque os diversos núcleos dos processadores *multicore* não podem ser considerados como processadores independentes, dado que compartilham barramento e também recursos dentro do mesmo *chip*.

Diante disso e do contexto apresentado no início deste capítulo, surgiu o projeto PLASMA (*Parallel Linear Algebra for Scalable MultiCore Architectures*) o qual é apresentado em [BUT07]. A PLASMA consiste em uma nova biblioteca em desenvolvimento pelos mesmos idealizadores do LINPACK, LAPACK e ScaLAPACK, e deverá ser a próxima geração dos pacotes de *software* para Álgebra Linear de Alto Desempenho. O desenvolvimento da PLASMA tem se dado de acordo com os princípios apontados na Seção 2.2. Para tal, a BLAS é utilizada apenas na implementação otimizada das operações com fluxo único de execução, também conhecidas como núcleos (*kernels*). Com isso, as otimizações em nível de instrução, ou seja, aquelas dependentes de máquina, são exploradas pela BLAS ao passo que o paralelismo é explorado em um nível algorítmico acima do nível da BLAS. Por essa razão, a PLASMA deve ser vinculada a versões sequenciais da BLAS ao invés de versões *multithread* como ocorria com seus antecessores. A Figura 2.6 ilustra as diferenças no nível de paralelismo.

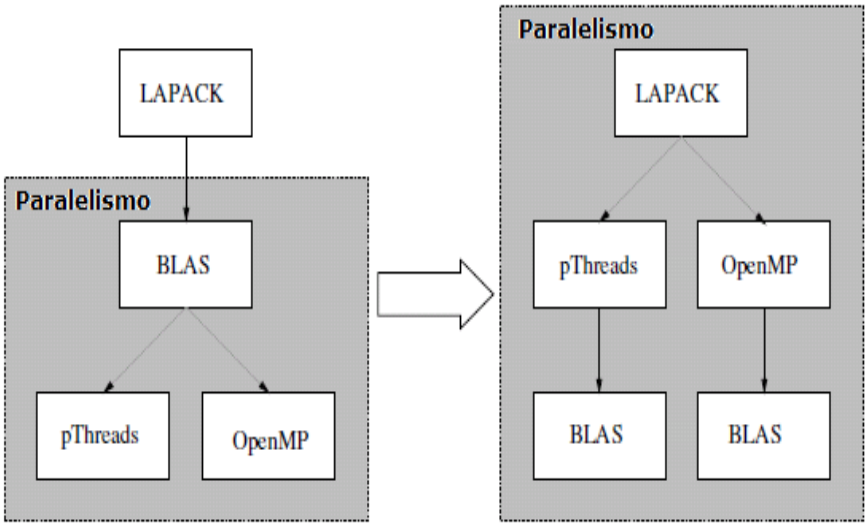

Figura 2.6: Paralelismo em Nível de BLAS X Algoritmos Paralelos. Adaptado de [BUT09]

Entretanto, não há, ainda, disponível uma versão completa final da PLASMA a qual possa substituir por completo o LAPACK e/ou ScaLAPACK. Em janeiro de 2009 os códigos-fonte de uma versão inicial com algumas rotinas da PLASMA foram disponibilizados. Em 04 de julho de 2009 uma segunda versão, mais completa e com mais rotinas, foi publicada contendo além dos códigos-fonte um instalador e documentações da biblioteca. No momento da conclusão do presente

trabalho, a versão mais atual disponível é 2.1.0 de 15 de novembro de 2009. Entretanto, embora a PLASMA tenha sido proposta para ser sucessora do LAPACK e ScaLAPACK, a migração desses para a PLASMA não será de maneira transparente para o usuário, como ocorreu na migração do LAPACK para ScaLAPACK, em que ambos apresentavam as mesmas rotinas e essas eram chamadas de maneira similar.

2.4 Considerações Finais

Este capítulo apresentou uma contextualização geral do tipo de arquitetura e das ferramentas de *software* para exploração do paralelismo empregadas neste trabalho. Cabe observar que o paralelismo tem conquistado espaço na Ciência da Computação como solução para os mais diversos problemas. Porém, as arquiteturas para tal estão em constante evolução e, portanto, nenhuma dessas pode ter a expectativa de se manter como solução ideal por muito tempo. Dessa forma, a escolha da arquitetura a ser utilizada no desenvolvimento de uma solução computacional deve priorizar atender os requisitos particulares da aplicação em questão.

No presente trabalho optou-se por trabalhar com computadores *multicore* pelos motivos já expostos, em especial a facilidade de acesso às mesmas. As ferramentas abordadas neste capítulo são aquelas que, ao longo do projeto, foram identificadas como mais adequadas para a aplicação em questão.

3. COMPUTAÇÃO VERIFICADA

A Computação Verificada garante o rigor matemático das operações fornecendo como resposta um intervalo o qual contém, com certeza, o resultado exato [KOL08c]. Os métodos que realizam a verificação automática dos resultados são chamados **métodos auto-validados** (*self validated*) e apresentam, basicamente, três objetivos: produzir resultados rigorosos; resolver o problema sem que o custo computacional seja demasiadamente maior do que o dos métodos puramente numéricos; incluir na computação provas da existência da solução e, quando for o caso, prova da unicidade da mesma [KOL08c], [RUM01]. A verificação automática dos resultados baseia-se em dois requisitos principais: utilização de algoritmos corretos e aplicação da teoria de Aritmética Intervalar em conjunto com arredondamentos direcionados [HAY03], [KEA96], [RUM83]. Este capítulo apresenta, inicialmente, a Aritmética Intervalar Real (a Aritmética Intervalar para números complexos não é tratada neste trabalho). Na sequência, a Computação Verificada é detalhada e, por fim, discute-se sua aplicação na resolução de SELAs.

3.1 Aritmética Intervalar

Aritmética Intervalar é uma aritmética definida para intervalos, ao invés de números reais. Não se trata de uma ideia nova, sua primeira aparição foi registrada em 1924 e foi reinventada diversas vezes desde então, nunca foi o foco principal da Computação Numérica, mas, também, nunca foi totalmente abandonada [HAY03], [KEA96]. A Aritmética Intervalar permite ao computador trabalhar com o contínuo e uma de suas principais funções é controlar o erro. Embora não elimine os efeitos dos erros de arredondamento, limita-os quando utilizada em conjunto com arredondamentos direcionados. Em uma computação típica, ou seja, não intervalar, o resultado é um número que representa um ponto no eixo dos números reais e está a alguma distância desconhecida da resposta exata. Na Computação Intervalar, por outro lado, o resultado é um par de números que formam um intervalo que contém o resultado exato. Tal intervalo recebe o nome de ***enclosure***, ou, **inclusão** [KOL08c], [HAM97]. Dessa forma, mesmo que a resposta exata continue desconhecida, ao menos é possível estimar o quão desconhecida ela é [HAY03], [KOL08c].

A avaliação intervalar de uma expressão aritmética costuma apresentar o dobro do custo computacional que a avaliação da mesma expressão com Aritmética de Ponto Flutuante convencional. Entretanto, uma única avaliação intervalar produz uma garantia do resultado que não é produzida mesmo com milhões de avaliações convencionais. Isso porque, é impossível avaliar todos os pontos no eixo dos números reais, dado que entre dois números reais existem, sempre, infinitos outros números e, além disso, muitos pontos não são finitamente representáveis [HAM97], [HAY03].

Um intervalo real é um subconjunto fechado, limitado e não vazio dos números reais, representado conforme a Equação 3.1, na qual $\underline{x}$ e $\overline{x}$ denotam, respectivamente, o **limite inferior** e **superior** do intervalo $[x]$. Tais limites são também chamados de **Ínfimo** e **Supremo**, respectivamente. Por

essa razão, essa forma de representar intervalos é denominada **Representação Ínfimo-Supremo** ou ***Infimum-Supremum***. Um intervalo real cobre toda a faixa de valores reais entre suas extremidades e, nos casos em que $\underline{x} = \overline{x}$, o intervalo é chamado **intervalo pontual** podendo ser escrito apenas como x ao invés de $[x]$. De maneira análoga, números exatos podem também ser representados como intervalos pontuais. Por exemplo, o número 2 pode ser escrito na forma $[2,2]$ [HAM97], [HAY03]. A Equação 3.2 apresenta outra representação possível para intervalos, chamada de **Representação por Ponto-Médio e Raio**, ou, ***Midpoint-Radius*** [RUM99].

$$[x] := [\underline{x}, \overline{x}] := \{x \in \Re / \underline{x} \leq x \leq \overline{x}\}, \tag{3.1}$$

$$\langle a, \alpha \rangle := \{x \in \Re / \, |x - a| \leq \alpha\} \;\; para \;\; a \in \Re, 0 \leq \alpha \in \Re \tag{3.2}$$

As representações em (3.1) e (3.2) são equivalentes para operações teóricas em $\Re$, nas quais não há necessidade de arredondamento dos resultados. Entretanto, a situação muda ao considerar cálculos executados em um computador digital, pois, nesse caso, os números reais precisam ser aproximados por números de ponto flutuante do **Sistema de Ponto Flutuante** [RUM99]. Essa aproximação é feita por um mapeamento especial chamado **arredondamento** (*rounding*) definido por $\bigcirc : \Re \rightarrow \mathbf{F}$. Uma operação em ponto flutuante é dita de **máxima exatidão** quando seu resultado arredondado difere do resultado exato em, no máximo, uma unidade na última casa decimal. A máxima exatidão é um comportamento padrão nos computadores quando considerando operações de ponto flutuante individuais. Porém, após uma série de operações consecutivas, o resultado pode ser completamente errado. Uma vez que os computadores executam milhões de operações em ponto flutuante por segundo, atenção especial deve ser dada à confiabilidade dos resultados [HAM97], [HAY03].

Além dos arredondamentos executados em cada operação de ponto flutuante, arredondamentos adicionais são necessários nos dados de entrada e nas constantes. Isso porque, as pessoas costumam pensar e trabalhar com notação decimal, logo, os programas utilizam essa mesma notação em suas interfaces criando a necessidade de mapear os dados decimais fornecidos pelo usuário para o Sistema de Ponto Flutuante Binário do computador. Tais arredondamentos costumam ocorrer em tempo de execução e, como em geral os números decimais não possuem representação binária exata, um pequeno erro, chamado **erro de conversão**, é gerado a cada mapeamento.

A operação de arredondamento deve satisfazer duas condições [HAM97]:

$$\begin{aligned} \bigcirc x = x \quad & \text{para todo} \quad x \in \Re, \quad \text{e} \\ x \leq y \Rightarrow \bigcirc x \leq y \quad & \text{para todo} \quad x, y \in \Re \end{aligned} \tag{3.3}$$

A primeira condição garante que os números não são alterados por um arredondamento. Já a segunda significa que o arredondamento é **monotônico**, ou seja, a ordem dos elementos é mantida ao arredondá-los. Existem três tipos de arredondamentos:

$$\begin{aligned} \square &: \text{ Arredondamento para o } \textbf{elemento mais próximo} \text{ do sistema} \\ \nabla &: \text{ Arredondamento } \textbf{direcionado a } -\infty \text{ ou } \textbf{direcionado para baixo} \\ \Delta &: \text{ Arredondamento } \textbf{direcionado a } +\infty \text{ ou } \textbf{direcionado para cima} \end{aligned} \tag{3.4}$$

Para preservar a garantia de que o valor correto sempre se encontra no intervalo, seus pontos extremos devem ser arredondados "para fora", ou seja, em $\underline{x}$ aplica-se arredondamento direcionado para baixo e em $\overline{x}$ arredondamento direcionado para cima [HAY03]. Diz-se que um arredondamento é **anti-simétrico** quando possui a seguinte propriedade:

$$\bigcirc(-x) = -\bigcirc x \quad \text{para todo} \quad x \in \Re \tag{3.5}$$

Portanto, o arredondamento para o elemento mais próximo $\square$ é anti-simétrico, porém, os arredondamentos direcionados ∇ e Δ não o são. Ao invés disso, $\nabla(-x) = -\Delta x$ e $\Delta(-x) = -\nabla x$. Os arredondamentos direcionados satisfazem as demais condições $\nabla x \leq x$, e $x \leq \Delta x$ para todo $x \in \Re$ [HAM97].

Dadas as propriedades dos arredondamentos, é importante considerar como as operações básicas $\circ \in \{+, -, ., /\}$ da aritmética são implementadas quando trabalhando no Sistema de Ponto Flutuante. No caso da representação Ínfimo-Supremo deve-se ter cuidado em alguns casos, especialmente nas multiplicações, pois, dependendo do sinal dos operandos, diferentes definições da operação são utilizados. Já no caso da aritmética representada por Ponto-Médio e Raio isso não ocorre, todas as operações são definidas diretamente sem necessidade de distinção [ALE00]. Por outro lado, a definição padrão da Aritmética de Ponto-Médio e Raio pode resultar em intervalos mais amplos do que o desejado nas multiplicações e divisões, efeito esse que recebe o nome de ***overestimation***. Considerando esse contexto, cada representação da Aritmética Intervalar possui algumas vantagens e desvantagens. As próximas seções descrevem brevemente as definições das operações básicas para ambas as representações.

3.1.1 Representação Ínfimo-Supremo

Na representação Ínfimo-Supremo, a execução de operações aritméticas básicas com intervalos é bastante próxima das operações com números pontuais tradicionais. De fato, uma única fórmula estende a definição das quatro operações básicas da Aritmética Real para suas versões intervalares. Sejam as operações $\circ \in \{+, -, ., /\}$ definidas em $\Re$, suas correspondentes intervalares são dadas por [HAY03]:

$$\begin{aligned} [\underline{x}, \overline{x}] \circ \left[\underline{y}, \overline{y}\right] = [&\min\left(\underline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \underline{y}, \overline{x} \circ \overline{y}\right) \\ &\max\left(\underline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \underline{y}, \overline{x} \circ \overline{y}\right)] \end{aligned} \tag{3.6}$$

Ou seja, calcula-se as quatro possíveis combinações para os limites superior e inferior e, na sequência, escolhe-se o menor resultado como limite inferior e o maior resultado como limite superior. Dessa forma, todas as combinações possíveis para $x \circ y$ estarão contidas no intervalo. Porém, embora

essa fórmula seja uma definição conveniente para operações intervalares, em geral não é a melhor opção, pois requer sempre a computação das quatro combinações de extremos sendo que em diversas situações é possível executar operações menos custosas [HAY03].Por exemplo, a multiplicação de dois intervalos $[x]$ e $[y]$ nos casos em que simultaneamente $0 \notin [x]$ e $y \notin [y]$ pode ser reduzida para a multiplicação de dois números reais [ALE98]. Aplicando as propriedades da monotonicidade às operações reais elementares, define-se as seguintes fórmulas mais convenientes [HAM97], [KEA96]:

$$\begin{array}{rl} [x]+[y]= & \left[\underline{x}+\underline{y}, \overline{x}+\overline{y}\right] \\ [x]-[y]= & \left[\underline{x}-\overline{y}, \overline{x}-\underline{y}\right] \\ [x]\,.\,[y]= & [\min\left\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\right\}, \max\left\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\right\}] \\ [x]\,/\,[y]= & [x]\,.\left[1/\overline{y}, 1/\underline{y}\right],\ 0 \notin [y] \end{array} \tag{3.7}$$

Cabe observar que a premissa de que as operações intervalares podem ser executadas com base apenas nos pontos finais do intervalo é válida somente para funções monotônicas, ou seja, aquelas que não alteram sua direção ao longo do domínio. Em outras funções, como por exemplo $sinh$, deve-se avaliar internamente os intervalos para obtenção dos mínimos e máximos. Diversos axiomas matemáticos falham quando considerados em um Sistema de Ponto Flutuante, mesmo em operações tradicionais. Por exemplo, a identidade $x = \sqrt{x^2}$ não é uma operação de resultado confiável. Entretanto, é possível construir sob o Sistema de Ponto Flutuante Padrão IEEE uma Aritmética Intervalar que jamais resulte em estado de erro, embora possa, em alguns casos, retornar valores tais como $[-\infty, +\infty]$, os quais não possuem utilidade prática [HAY03].

3.1.2 Representação Ponto-Médio e Raio

A seguir são apresentadas as operações intervalares na representação de Ponto-Médio e Raio conforme definidas em [RUM99]. De modo a facilitar a visualização, as operações em ponto flutuante tradicional são denotadas por $\circ \in \{+, -, ., /\}$ e suas correspondentes intervalares por $\odot \in \{\oplus, \ominus, \odot, \oslash\}$. Segue-se a mesma convenção para operações sob vetores e matrizes intervalares. Assim, seja $I\Re$ o conjunto dos reais intervalares e sejam os intervalos $X = \langle x, \alpha \rangle \in I\Re$ e $Y = \langle y, \beta \rangle \in I\Re$, as operações aritméticas básicas são definidas por:

$$\begin{array}{rl} X \oplus Y := & \langle x+y, \alpha+\beta \rangle \\ X \ominus Y := & \langle x-y, \alpha+\beta \rangle \\ X \odot Y := & \langle x.y, |x|\,\beta + \alpha\,|y| + \alpha\beta \rangle \\ 1 \oslash Y := & \langle y/D, \beta/D \rangle \ \ onde\ \ D := y^2 - \beta^2 e 0 \notin Y \\ X \oslash Y := & X \odot (1 \oslash Y) \ \ para\ \ 0 \notin Y \end{array} \tag{3.8}$$

As operações intervalares sempre satisfazem a propriedade fundamental da isotonicidade, ou seja, se X está contido em outro intervalo X' e Y contido em Y', então a combinação de X e Y está contida no intervalo computado pela combinação dos intervalos maiores X' e Y'. Qualquer representação ou implementação de uma Aritmética Intervalar deve, portanto, obedecer a isotonici-

dade [HAM97]. Conforme visto anteriormente, a representação por Ponto-Médio e Raio pode, em algumas situações, causar efeitos de *overestimation*. Tal efeito em uma operação intervalar é dado por [RUM99]:

$$\rho := \frac{rad(A \odot B)}{rad(A \circ B)} \ para \ \circ \in \{+, -, ., /\}. \tag{3.9}$$

Onde ρ consiste na medida de *overestimation*. Entretanto, Rump [RUM99] provou que o efeito de *overestimation* (3.9) nas operações aritméticas básicas utilizando Aritmética de Ponto-Médio e Raio, bem como em operações sob vetores e matrizes, é limitado ao fator de $1,5$. Para intervalos de raio pequeno esse fator se reduz a 1. Por fim, Rump [RUM99] observa que, além da implementação das operações da Aritmética de Ponto-Médio e Raio ser bastante simples nos computadores atuais, essa permite construir rotinas muito mais rápidas do que a abordagem tradicional de Ínfimo-Supremo, principalmente nos computadores paralelos.

3.2 Computação Verificada e Resolução de SELAs

A Computação Verificada garante o rigor matemático das operações. Sua implementação requer, além do emprego de algoritmos corretos, que as operações sejam executadas com máxima exatidão e construídas utilizando Aritmética Intervalar, a qual, por sua vez, deve ser computacionalmente implementada com o emprego de arredondamentos direcionados. Cabe observar, porém, que a Computação Verificada não elimina a necessidade de verificação dos algoritmos, compiladores, sistemas operacionais, dentre outros, uma vez que o passo de verificação pode resultar em falsos positivos devido a erros de programação ou ambiente de execução não confiável [HAM97].

O passo inicial para realizar Computação Verificada é substituir as operações em $\Re$ por suas equivalentes intervalares e, então, executá-las utilizando Aritmética Intervalar. Dessa forma, produz-se resultados intervalares. Porém, em geral, o diâmetro dos intervalos, ou seja, a distância entre $\underline{x}$ e $\overline{x}$, é tão grande que o resultado torna-se inútil. Logo, métodos mais sofisticados fazem-se necessários. Tais métodos consistem em uma combinação dos benefícios da Aritmética Intervalar com um mecanismo de refinamento dos intervalos encontrados, de forma que seu diâmetro seja o menor possível contendo o resultado exato [HAM97].

O refinamento iterativo é um processo que, a partir de uma solução inicial, repete o cálculo visando diminuir o erro da mesma até que o tamanho do intervalo seja menor do que a exatidão desejada. Assim, a inclusão verificada é dada pelo intervalo somado ao erro. Tais métodos, em geral, obtêm êxito em encontrar uma inclusão suficientemente útil da solução e, caso contrário, notificam o usuário de que a computação falhou. Na Aritmética de Ponto Flutuante convencional, por outro lado, a computação pode falhar sem que qualquer sinal seja dado ao usuário ou, ainda, pode ser dado um resultado incorreto [KOL08c].

Muitos dos algoritmos para verificação numérica são baseados na aplicação dos teoremas de ponto fixo clássicos da Matemática. Por exemplo, a convergência é garantida pelo Teorema de Ponto Fixo de Brouwer [KEA96]. Seja $X = [x] \in I\Re^n$ um vetor intervalar em ponto flutuante que

satisfaz as condições do Teorema de Ponto Fixo de Brouwer. Supondo que é possível encontrar uma f tal que $f([x]) \subseteq [x]$, então $[x]$ é comprovadamente uma inclusão de pelo menos um ponto fixo x^* de f. A afirmação permanece válida substituindo-se f por sua avaliação intervalar em ponto flutuante $f_\diamond$, pois $f_\diamond([x]) \subseteq [x]$ implica $f([x]) \subseteq [x]$ uma vez que $f([x])$ é subconjunto de $f_\diamond([x])$. O Teorema de Ponto Fixo de Brouwer combinado com a Aritmética Intervalar permite à Computação Numérica provar a existência e unicidade das soluções para Sistemas Lineares e Não-Lineares [KEA96], [HAY03], [RUM83].

Versões intervalares de diversos métodos clássicos para resolução de SELAs estão disponíveis, porém, elas costumam diferir bastante de suas versões tradicionais. Por exemplo, em geral o sistema $Ax = b$ precisa ser pré-condicionado com uma matriz pontual para que os algoritmos sejam eficientes. Uma possibilidade é calcular a solução aproximada para um sistema pontual utilizando-se bibliotecas de *software* otimizadas e, então, empregar essa solução na obtenção dos limites para formar um intervalo no qual se sabe que a solução exata está contida [KEA96]. Outra grande diferença é a habilidade da Computação Verificada de encontrar soluções mesmo para problemas bastante mal condicionados.

O **condicionamento** de uma matriz regular quadrada A é definido pela Equação 3.10. Tal número aponta o quanto modificações na entrada dos dados influenciam na qualidade do resultado final da computação e permite determinar o quão confiável é a solução do SELA. É um número do qual interessa a ordem de grandeza e, além disso, está relacionado com a exatidão de máquina usada. Se $cond(A) = 10^5$ mas a exatidão decimal da máquina for de 17 casas, então não há problema em se perder os últimos cinco dígitos. Porém, se a exatidão decimal for de apenas 6 casas decimais, então o resultado será confiável apenas até a primeira casa decimal. Tipicamente, algoritmos de ponto flutuante convencionais podem encontrar uma solução para problemas com condicionamento em torno de 10^8. Por outro lado, um algoritmo em Computação Verificada terá, em geral, sucesso em encontrar uma solução mesmo para problemas com condicionamento em torno de $2.5.10^{15}$ [CLA00], [HAM97].

$$cond(a) := ||A||_\infty . \left|\left|A^{-1}\right|\right|_\infty \tag{3.10}$$

Na próxima subseção é apresentado o Método Verificado Baseado no Método de *Newton*, o qual foi escolhido para desenvolvimento do presente trabalho. Sua descrição completa, bem como uma implementação sequencial do mesmo utilizando C-XSC, pode ser encontrada em [HAM97].

3.2.1 Método Verificado Baseado no Método de *Newton*

Seja $Ax = b$ um Sistema de Equações, encontrar a solução do mesmo é equivalente a encontrar a raiz de uma função $f(x) = Ax - b$. Assim, o Método de *Newton* gera a seguinte iteração de ponto fixo, onde $x^{(0)}$ é algum valor inicial arbitrário qualquer [HAM97]:

$$x^{(k+1)} = x^{(k)} - A^{-1}\left(Ax^{(k)} - b\right), \quad k = 0, 1, \ldots \tag{3.11}$$

Em geral, a inversa exata de A não é conhecida. Assim, ao invés de (3.11), a chamada *Newton-like Iteration*[1] é utilizada, onde $R \approx A^{-1}$ é uma inversa aproximada de A:

$$x^{(k+1)} = x^{(k)} - R\left(Ax^{(k)} - b\right), \quad k = 0, 1, \ldots, \tag{3.12}$$

Substituindo a iteração real $x^{(k)}$ por vetores intervalares $[x]^{(k)} \in I\Re^n$, se existe um índice k com $[x]^{(k+1)} \subset [x]^{(k)}$, então, pelo Teorema de Ponto Fixo de Brouwer, a Equação 3.12 tem pelo menos um ponto fixo $x \in [x]^{(k)}$. Supondo que R é regular, então esse ponto fixo é também uma solução para $Ax = b$. Cabe observar que a relação de inclusão para vetores e matrizes intervalares é definida por composição, ou seja, $[x] \overset{\circ}{\subset} [y] \Leftrightarrow [x]_i \overset{\circ}{\subset} [y]_i$, i $= 1, \ldots, n$ para $[x], [y] \in I\Re^n$. A relação de subconjunto é dada por $[x] \subset [y] \Leftrightarrow ([x] \subseteq [y] \wedge [x] \neq [y])$.

Entretanto, considerando o diâmetro de $[x]^{(k+1)}$, obtém-se:

$$d\left([x]^{(k+1)}\right) = d\left([x]^{(k)}\right) + d\left(R\left(A\,[x]^{(k+1)} - b\right)\right) \geq d\left([x]^{(k)}\right) \tag{3.13}$$

Assim, em geral, a relação de subconjunto não é satisfeita. Por essa razão, o lado direito de (3.12) é modificado para a equação a seguir, onde I denota a matriz identidade nxn [HAM97]:

$$x^{(k+1)} = Rb + (I - RA)\, x^{(k)}, \quad k = 0, 1, \ldots, \tag{3.14}$$

Sabe-se que se existe um índice k com $[x]^{(k+1)} \overset{\circ}{\subset} [x]^{(k)}$, então as matrizes R e A são regulares e existe uma solução única x do sistema $Ax = b$ com $x \in [x]^{(k+1)}$. Esse resultado permanece válido para qualquer matriz R. Entretanto, é um fato empírico que quanto mais R se aproxima da inversa de A, mais rapidamente o método converge. Adicionalmente, é um princípio numérico bem estabelecido que uma solução aproximada $\tilde{x}$ de $Ax = b$ pode ser melhorada resolvendo-se o sistema $Ay = d$, onde $d = b - A\tilde{x}$ é o resíduo de $A\tilde{x}$. Uma vez que $y = A^{-1}(b - A\tilde{x}) = x - \tilde{x}$, a solução exata de $Ax = b$ é dada por $x = \tilde{x} + y$. Aplicando a Equação 3.14 ao sistema residual, obtém-se o *Esquema de Iteração Residual* [HAM97]:

$$y^{(k+1)} = \underbrace{R(b - A\tilde{x})}_{=:z} + \underbrace{(I - RA\tilde{x})}_{=:C} y^{(k)}, \quad k = 0, 1, \ldots \tag{3.15}$$

A equação residual $Ay = d$ tem uma solução única $y \in [y]^{(k+1)} \overset{\circ}{\subset} [y]^{(k)}$ para a iteração intervalar correspondente. Além disso, uma vez que $y = x - \tilde{x} \in [y]^{(k+1)}$, uma solução verificada da solução única de $Ax = b$ é dada por $\tilde{x} + [y]^{(k+1)}$. Esses resultados permanecem válidos ao substituir as expressões exatas z e C em (3.15) por extensões intervalares. Entretanto, para diminuir os efeitos de *overestimation*, recomenda-se fortemente avaliar $b - A\tilde{x}$ e $I - RA$ sem arredondamentos intermediários.

[1] Não há uma tradução exata no português para o nome do método. Pode-se entender algo próximo de Iteração Baseada no Método de *Newton*

3.3 Considerações Finais

Neste capítulo foram apresentadas as bases necessárias para a construção de um *solver* com verificação automática dos resultados. Constatou-se que, para a resolução de SELAs de acordo com a proposta deste trabalho, o Método Verificado Baseado no Método de *Newton* é o mais adequado por ser capaz de resolver Sistemas Densos permitindo o emprego da Aritmética Intervalar de Ponto-Médio e Raio. Tal aritmética é de grande importância nesse contexto por possibilitar a utilização das bibliotecas de *software* otimizadas para Álgebra Linear vistas no capítulo anterior. A implementação empregando essas bibliotecas traz grandes benefícios não apenas em termos de desempenho mas também facilita o desenvolvimento. Observou-se também que atenção especial deve ser dada às direções dos arredondamentos quando trabalhando com Computação Verificada.

4. RESOLUÇÃO AUTO-VALIDADA DE SELAS EM *MULTICORE*

O presente trabalho foi desenvolvido com o objetivo de construir uma ferramenta computacional para resolução de SELAs Intervalares Densos de Grande Porte com verificação automática dos resultados otimizada para execução em arquiteturas *multicore*. A obtenção de tal resultado envolve a aplicação dos conceitos, ferramentas e estratégias discutidos nos capítulos anteriores. Neste capítulo, é descrito o processo executado para implementação do *solver* proposto.

Primeiramente, é apresentada uma solução inicial a qual foi desenvolvida visando validar a abordagem proposta e, consequentemente, obter um *software* capaz de executar nos computadores foco do trabalho, embora sem ganhos de desempenho. Após, são descritas as avaliações realizadas em tal versão de modo a validar seus resultados numéricos e obter os custos computacionais dos passos de execução. Por fim, o desenvolvimento da versão otimizada é apresentado.

4.1 Solução Proposta

O *solver* proposto no presente trabalho foi construído com base no Método de *Newton* descrito no Capítulo 3. A implementação de tal método utilizando a Aritmética Intervalar de Ponto-Médio e Raio é dada pelo Algoritmo 4.1, o qual é apresentado em [KOL08c] e trata-se de uma adaptação do algoritmo originalmente proposto por [HAM97]. Como resultado, o algoritmo gera um vetor em que cada elemento é um intervalo o qual contém o resultado correto. Tais intervalos são de máxima exatidão.

Algoritmo 4.1: Resolução Auto-Validada de um Sistema Linear Intervalar Quadrado.

1: $R \approx mid([A])^{-1}$ {Computa uma inversa aproximada}
2: $\tilde{x} \approx R.mid([b])$ {Calcula uma inclusão da solução}
3: $[z] \supseteq R([b] - [A]\,\tilde{x})$ {Computa uma inclusão do resíduo}
4: $[C] \supseteq (I - R[A])$ {Computa uma inclusão da matriz de iteração}
5: $[w] := [z], k := 0$ {Inicializa o vetor intervalar de máquina}
6: **while not** $([w] \subseteq int[y]$ ou $k > 10)$ **do**
7: $\quad [y] := [w]$
8: $\quad [w] := [z] + [C][y]$
9: $\quad k++$
10: **end while**
11: **if** $[w] \subseteq int[y]$ **then**
12: $\quad \sum([A], [b]) \subseteq \tilde{x} + [w]$ {O conjunto solução $(\sum)$ está contido na solução encontrada pelo método}
13: **else**
14: $\quad$ Verificação falhou
15: **end if**

Empregou-se, para a implementação inicial do Algoritmo 4.1, a linguagem C++ e a biblioteca

Intel MKL 10.2.1.017 como versão otimizada das bibliotecas BLAS e LAPACK. É importante observar que, devido às operações da Aritmética Intervalar, o algoritmo tem um consumo de memória bastante elevado. Somando-se a entrada com os vetores e matrizes auxiliares necessários, tem-se a utilização de um total de 8 matrizes nxn e 20 vetores de ordem n. Visando obter maior desempenho, a inversa aproximada R e a solução aproximada x são calculadas utilizando-se apenas a matriz de ponto-médio e operações de ponto flutuante tradicionais. Posteriormente, na computação do resíduo, utiliza-se a Aritmética Intervalar em conjunto com a matriz intervalar $[A]$ original e o vetor intervalar $[b]$ para garantir a exatidão do resultado [KOL08c].

No Passo 1 do algoritmo, ou seja, no cálculo de R, as seguintes rotinas do LAPACK foram empregadas:

- *dgetrf*: Computa, em dupla exatidão (*double*), uma fatoração LU da matriz utilizando pivotamento parcial com troca de linhas. A fatoração tem o formato $A = P.L.U$, onde P é a matriz de permutação, L uma matriz triangular inferior com elementos diagonais unitários e U triangular superior;

- *dlange*: Calcula a norma da matriz A a partir da fatoração LU resultante de *dgetrf*;

- *dgecon*: Computa, a partir da saída de *dlange* e da A original, o número de condicionamento do sistema;

- *dgetri*: Calcula, com dupla exatidão, uma matriz inversa aproximada de A. O cálculo é feito a partir do resultado da fatoração LU dado por *dgetrf* e composto pelos seguintes passos: Inversão triangular de U por *forward substitution* (substituição para frente); Resolução do sistema triangular $X : XL = U^{-1}$ (*backward substituion*); Permutação reversa de colunas, $A^{-1} = XP$.

O cálculo do condicionamento do sistema foi inserido como rotina intermediária no Passo 1 por depender da saída da fatoração LU, a qual, por sua vez, é sobrescrita no cálculo da inversa aproximada. No Passo 2 do algoritmo, ou seja, no cálculo da solução x aproximada, utilizou-se a seguinte rotina da BLAS:

- *dgemv*: Calcula, em dupla exatidão, a multiplicação entre a matriz R e o vetor b.

Os passos 3 e 4 computam, respectivamente, a inclusão do resíduo e a inclusão da matriz de iteração, ou seja, as inclusões necessárias para iniciar a iteração de verificação. Dado que $[A]$ e $[b]$, assim como $[C]$ e $[z]$, são intervalares, é necessário a utilização dos algoritmos da Aritmética de Ponto-Médio e Raio apresentados em [RUM99]. Seja I^+F o conjunto dos números intervalares no Sistema de Ponto Flutuante, sejam os intervalos de Ponto-Médio e Raio $A = \langle \tilde{a}, \alpha \rangle \in I^+F$ e $B = \left\langle \tilde{b}, \beta \right\rangle \in I^+F$, as operações em Ponto-Médio e Raio de adição e subtração $C := A \circ B \in I^+F$, com $\circ \in \{+, -\}$ e $C = \langle \tilde{c}, \gamma \rangle$ são definidas para o padrão IEEE 754 [ANS85] pelo Algoritmo 4.2. De maneira análoga, a multiplicação é definida pelo Algoritmo 4.3. Os símbolos $\square$, ∇ e Δ indicam,

respectivamente, os modos de arredondamento para o elemento mais próximo, direcionado para baixo e direcionado para cima.

Algoritmo 4.2: Adição e subtração de Ponto-Médio e Raio no padrão IEEE 754.

1: $\tilde{c} = \square\left(\tilde{a} \circ \tilde{b}\right)$
2: $\tilde{\gamma} = \Delta\left(\epsilon' \left|\tilde{c}\right| + \tilde{\alpha} + \tilde{\beta}\right)$

Algoritmo 4.3: Multiplicação de Ponto-Médio e Raio no padrão IEEE 754.

1: $\tilde{c} = \square\left(\tilde{a}.\tilde{b}\right)$
2: $\tilde{\gamma} = \Delta\left(\eta + \epsilon' \left|\tilde{c}\right| + \left(\left|\tilde{a}\right| + \tilde{\alpha}\right)\tilde{\beta} + \tilde{\alpha}\tilde{\beta}\right)$

Conforme visto anteriormente, as grandes vantagens da Aritmética de Ponto-Médio e Raio são permitir cálculos com operações de ponto flutuante puras e não necessitar de alterações intermediárias no modo de arredondamento, diferentemente da abordagem Ínfimo-Supremo. Portanto, embora $[C]$ e $[z]$ sejam intervalares, o cálculo dos mesmos foi implementado empregando-se, respectivamente, as rotinas *dgemv* e *dgemm* da BLAS com arredondamentos direcionados. A rotina *dgemv* implementa, em exatidão dupla, a multiplicação de matriz por vetor, enquanto *dgemm* implementa, também em dupla exatidão, a multiplicação de matriz por matriz.

De modo a formar os limites superior e inferior de $[C]$, *dgemm* é executada duas vezes, uma com arredondamentos direcionados para cima e outra com arredondamentos direcionados para baixo. Esse controle é feito no C++ através das funções disponibilizadas pela biblioteca *fenv.h*. No presente trabalho empregou-se a rotina *fesetround*, a qual recebe como parâmetro a direção desejada do arredondamento e altera o mesmo no processador no qual o processo está executando. Os macros que representam os tipos de arredondamento possíveis para essa função são: *FE_DOWNWARD, FE_TONEAREST, FE_TOWARDZERO, FE_UPWARD* os quais correspondem a, respectivamente, arredondamento direcionado para baixo, para o elemento mais próximo, direcionado a zero e direcionado para cima.

Um erro de arredondamento é gerado na avaliação do ponto-médio. Esse erro pode ser compensado com a utilização de uma unidade de erro relativa. Em [RUM99] o autor representa a unidade de erro relativa como ϵ, define $\epsilon' = \frac{1}{2}\epsilon$, e define o menor número representável de ponto flutuante positivo não normalizado por η. No padrão de exatidão dupla IEEE 754, $epsilon = 2^{-52}$ e $\eta = 2^{-1074}$. Assim, a avaliação do ponto-médio ($\tilde{c}$) e do raio ($\tilde{\gamma}$) de C é implementada através do Algoritmo 4.4.

Por fim, os passos de 5 a 15 implementam o Iteração Baseada no Método de *Newton* para encontrar a inclusão da solução. Nestes passos a Aritmética de Ponto-Médio e Raio é aplicada em conjunto com os arredondamentos direcionados. Na multiplicação $[C].[y]$ (Passo 8) utilizou-se novamente a rotina *dgemv* da BLAS. O laço *while* verifica se o novo resultado $[w]$ está contido no interior do resultado prévio $[y]$. Cabe lembrar que, conforme visto na Seção 3.2.1, a relação de

Algoritmo 4.4: Multiplicação de Matrizes em Ponto-Médio e Raio no padrão IEEE 754.

1: $\tilde{c}_1 = \nabla(R.mid(A))$
2: $\tilde{c}_2 = \Delta(R.mid(A))$
3: $\tilde{c} = \Delta(\tilde{c}_1 + 0.5(\tilde{c}_2 - \tilde{c}_1))$
4: $\tilde{\gamma} = \Delta(\tilde{c} - \tilde{c}_1) + |R|.rad(A)$

inclusão para vetores e matrizes intervalares é definida por composição. Assim, a mesma é avaliada elemento a elemento dos vetores.

O laço *while* executa 10 iterações em busca da relação de inclusão. Isso porque, de acordo com [KOL08c], é um fato empírico que se a mesma não for satisfeita após poucas iterações, é porque de fato ela não ocorre. Uma vez satisfeita a relação, tem-se a inclusão e o laço termina trazendo como resposta a resolução verificada do SELA. Se após as 10 iterações não houver sucesso, então o algoritmo termina sua execução indicado para o usuário que não obteve êxito.

4.2 Avaliação de Exatidão e Levantamento da Complexidade de Tempo

Dois diferentes tipos de testes foram executados para avaliar a implementação inicial: de desempenho (tempo de execução) e de exatidão. Os testes de desempenho tiveram dois objetivos. O primeiro deles é verificar a viabilidade da solução proposta, ou seja, se as estratégias empregadas são capazes de produzir uma ferramenta que resolva Sistemas Intervalares de Grande Porte em tempo satisfatório nas arquiteturas *multicore*. O segundo objetivo é verificar quais os trechos do algoritmo que possuem maior custo computacional. Embora em [KOL08b] os autores apontem os trechos do algoritmo com maior complexidade, a implementação deste projeto difere daquela em relação às ferramentas de *software*, paradigma de programação e arquitetura utilizadas, logo, faz-se necessário reavaliar os custos. Já os testes de exatidão fazem-se necessários para garantir que os algoritmos e ferramentas utilizadas obtêm, de fato, resultados válidos e úteis.

Em um primeiro momento, o ambiente para execução dos testes foi um computador portátil equipado com processador *Intel Core 2 Duo* T6400 @ 2.00 GHz com 2 MB de *cache*, 3 GB de memória RAM DDR2 operando em *Dual Channel* a 667 MHz. O computador executava o sistema operacional *Ubuntu Linux* 9.04, versão 32 *bits*, *kernel* 2.6.28-13-*generic*. Empregou-se a biblioteca *Intel* MKL 10.2.1.017 para chamadas às versões otimizadas das bibliotecas LAPACK e BLAS e o compilador utilizado foi o GCC. Posteriormente, avaliou-se novamente a implementação inicial na arquitetura paralela de maior porte utilizada durante os testes da solução final, conforme será visto no próximo capítulo.

4.2.1 Avaliação de Exatidão

Visando verificar a exatidão dos resultados, gerou-se um sistema baseado na fórmula de *Boothroyd/Dekker* (Equação 4.1) de ordem 10 para o qual a solução exata é conhecida. Tal matriz é um exemplo de matriz mal condicionada sendo que, para $n = 10$, apresenta condicionamento de

$1,09.10^{+15}$. De modo a tornar os dados de entrada intervalares, preencheu-se a matriz relativa ao raio de A e o vetor correspondente ao raio de b com o valor $0,1.10^{-10}$. O sistema foi resolvido com a implementação inicial do presente trabalho e, na sequência, comparou-se os resultados com a solução exata e com os resultados apresentados por [KOL08a]. Tais valores são descritos na Tabela 4.1. De modo a facilitar a visualização, os resultados são apresentados na notação Ínfimo-Supremo, embora o algoritmo utilize a Aritmética de Ponto-Médio e Raio.

$$A_{ij} = \binom{n+i-1}{i-1} \times \binom{n-1}{n-j} \times \frac{n}{i+j-1}, b_i = i, \forall (i,j) = 1,2,\ldots,N \tag{4.1}$$

Tabela 4.1: Resultados numéricos do *solver* inicial na resolução de um Sistema de *Boothroyd/Dekker* de ordem 10.

		Implementação Desenvolvida		**Resultados de [KOL08a]**	
X	Res. Exato	Ínfimo	Supremo	Ínfimo	Supremo
0	0,0	-0,0000119	0,0000108	-0,0000023	0,0000034
1	1,0	0,9998992	1,0001113	0,9999672	1,0000221
2	-2,0	-2,0005827	-1,9994736	-2,0001182	-1,9998255
3	3,0	2,9979758	3,0022444	2,9993194	3,0004611
4	-4,0	-4,0070747	-3,9936270	-4,0014680	-3,9978331
5	5,0	4,9826141	5,0193190	4,9940374	5,0040392
6	-6.0	-6,0406053	-5,9574730	-6,0099531	-5,9853083
7	7,0	6,9045776	7,1061843	6,9668534	7,0224518
8	-8,0	-8,2221804	-7,8004473	-8,0471964	-7,9303270
9	9,0	8,6064275	9,4384110	8,8620181	9,0934651

Percebe-se que os resultados das duas versões apresentam pequenas diferenças nos diâmetros dos intervalos, embora ambas contenham o resultado exato. Acredita-se que tais variações nos diâmetros se devam às diferentes estruturas de desenvolvimento e teste utilizadas. Em especial, a implementação de [KOL08a] utilizou como entrada dados pontuais, foi compilada com *Intel* icc 10.0 e executada em sistema operacional e processadores de 64 *bits*, enquanto o presente trabalho foi avaliado com entradas intervalares, em 32 *bits* e compilador GCC. Além disso, diferentes paradigmas de programação, distribuições do sistema operacional *Linux* e versões da biblioteca MKL foram empregados.

Ainda no contexto de avaliação de exatidão dos resultados, resolveu-se um sistema de ordem 5.000 com valores de A e b gerados aleatoriamente entre 0 e 1. Os raios foram preenchidos com valor $0,1.10^{-10}$ e o número de condicionamento do sistema é igual a $6,12.10^{1}$. Nesse caso, verificou-se o comportamento do *solver* proposto ao resolver problemas bem condicionados. A Tabela 4.2 apresenta os dez primeiros valores resultantes dessa avaliação. Gerou-se diversos outros sistemas de maneira análoga e os testes levaram, em todos os casos, a resultados de comportamento semelhantes ao descrito a seguir.

Conforme esperado, obteve-se resultados bastante satisfatórios para os sistemas bem condicionados. No exemplo da Tabela 4.2, diversos valores apresentaram diâmetro 0 e alguns casos 1.10^{-7}

Tabela 4.2: Resultados numéricos do *solver* inicial na resolução de um sistema aleatório de ordem 5.000.

X	Ínfimo	Supremo
0	-0,0164408	-0,0164408
1	0,1145639	0,1145639
2	0,7913921	0,7913922
3	0,7498999	0,7498999
4	0,9246397	0,9246397
5	-0,7683214	-0,7683214
6	0,0486910	0,0486910
7	0,3840869	0,3840869
8	-0,5795236	-0,5795235
9	0,4854377	0,4854377

para uma exatidão de 7 casas decimais. Por outro lado, para o exemplo mal-condicionado os resultados são menos exatos, obteve-se um diâmetro médio de $1,6009078.10^{-1}$. Em [KOL08a] os autores obtiveram, também, resultados menos satisfatórios para esse exemplo, o diâmetro médio daquela solução foi de $4,436911.10^{-2}$. Cabe observar, que tais soluções são consideradas inclusões, ou seja, são resultados verificados. Portanto, pode-se considerar como atendido o principal requisito em relação ao *solver* proposto: obteve-se resultados verificados com o emprego das ferramentas e arquiteturas disponíveis.

4.2.2 Avaliação da Complexidade de Tempo

Em relação aos testes de desempenho, o procedimento executado foi gerar matrizes A e vetores b de tamanhos variados e preenchê-los com números aleatórios entre 0 e 1. Contabilizou-se os tempos de execução para cada um dos passos do Algoritmo 4.1 sendo que a contabilização do Passo 1 foi subdividida em 4 submedições, o Passo 3 em duas e os passos de 6 a 15 unidos em uma única medida. Executou-se o algoritmo 10 vezes, removeu-se os tempos extremos inferior e superior de cada passo e obteve-se os valores finais através da média dos restantes. Utilizou-se, para tal, as funções de tempo próprias do sistema operacional *Linux*, as quais estão definidas no arquivo de cabeçalho *time.h*.

A Tabela 4.3 apresenta os tempos médios, em segundos, contabilizados para resolução de um sistema de ordem 5.000 formado por valores gerados aleatoriamente. Os tempos descritos fazem referência às seguintes operações:

1. Fatoração LU da matriz A (Passo 1.1);
2. Cálculo da norma de A (Passo 1.2);
3. Cálculo do número de condicionamento (Passo 1.3);
4. Computação de R, matriz inversa aproximada de A (Passo 1.4);

5. Cálculo da solução aproximada de x (Passo 2);
6. Cálculo do resíduo de x (Passo 3.1);
7. Cálculo da inclusão do resíduo de x, ou seja, cálculo do z (Passo 3.2);
8. Computação da inclusão da matriz de iteração (pré-condicionamento), ou seja, cálculo de $[C]$ (Passo 4);
9. Inicialização do vetor intervalar de máquina (Passo 5);
10. Refinamento e teste de inclusão, passo final da verificação (passos 6 à 15);
11. Execução completa (incluindo operações de entrada e saída).

Tabela 4.3: Tempos médios, em segundos, consumidos pelo *solver* inicial para resolução de um sistema aleatório de ordem 5.000.

Etapa	Tempo médio (segundos)
1	12,484145
2	0,069251
3	0,436299
4	131,662466
5	0,535467
6	0,324615
7	1,625113
8	109,452704
9	0,000117
10	4,635784
11	262,710470

Observando-se a Tabela 4.3 é possível perceber claramente que os passos 1 e 4 apresentam a maior complexidade de tempo do algoritmo. O Passo 1 consumiu um total de 144,65s e o Passo 4 109,45s correspondendo, respectivamente, a 55% e 42% do tempo total de execução. No Passo 1 aproximadamente 9% do custo corresponde a fatoração LU da matriz e o restante à rotina de inversão da matriz propriamente dita.

Contabilizou-se, também, o tempo médio gasto pela instrução que altera o tipo de arredondamento do processador. Tal operação consome em média $0,0000025$ segundos. Considerando o algoritmo otimizado para multiplicação de matrizes com Aritmética Ínfimo-Supremo proposto por [KNU94], o qual requer n^2 alterações no modo de arredondamento, ter-se-ia um custo adicional de $62,5$ segundos (aproximadamente 24%) relativo às alterações no arredondamento ao resolver-se um sistema com $n = 5000$. Logo, comprova-se que, de fato, a utilização da Aritmética Intervalar de Ponto-Médio e Raio traz grandes ganhos não apenas por permitir a utilização de bibliotecas otimizadas, mas, também, pela economia gerada ao eliminar as constantes alterações no modo de arredondamento.

4.3 Otimização da Solução Proposta

Conforme constatado nas avaliações apresentadas na Seção 4.2.2, os passos 1 (inversão da matriz $[A]$ e 4 do algoritmo (cálculo da inclusão $[C]$) consomem, juntos, 97% do tempo total de execução da ferramenta. Dessa forma, optou-se por focar a otimização do *solver* nesses dois pontos. Observa-se novamente que, devido às operações da Aritmética Intervalar, o algoritmo tem um consumo de memória bastante elevado dada a quantidade de vetores e matrizes auxiliares necessários. Dessa forma, na solução otimizada todos os vetores e matrizes são dinamicamente criados e liberados. As próximas seções descrevem a paralelização do *solver* desenvolvido.

4.3.1 Inversão da Matriz A

Inicialmente, efetuou-se uma busca na literatura por algoritmos que permitissem uma paralelização eficiente do cálculo da matriz inversa. Entretanto, não foram encontradas muitas referências de métodos paralelos destinados à inversão de matrizes genéricas. Os trabalhos disponíveis relatam, basicamente, algoritmos para inversão em paralelo das chamadas Matrizes Simétricas Positivo-Definidas (*Symmetric Positive-Definite Matrices*). Tais técnicas fazem uso das estruturas especiais dessas aplicando, por exemplo, a fatoração de *Cholesky*. Dado que o presente projeto é destinado a SELAs Densos, necessita-se trabalhar com matrizes quadradas genéricas não sendo, portanto, possível basear-se em tais técnicas.

De fato, há consenso na literatura de que a inversão de matrizes é um procedimento altamente custoso computacionalmente. Em muitas situações os problemas são remodelados para evitar a necessidade de calcular matrizes inversas, porém, nem sempre isso é possível [QUI00]. Há relatos na literatura de um algoritmo capaz de inverter uma matriz quadrada em $O\left(log^2 n\right)$ operações. Entretanto, tal algoritmo é apenas de interesse teórico, uma vez que o mesmo necessita de um número excessivo de processadores: n^4. Desconhece-se se há algum algoritmo capaz de produzir uma inversa com complexidade menor do que $O\left(log^2 n\right)$, mesmo no caso de matrizes triangulares. A biblioteca C-XSC utiliza o método da Eliminação de Gauss-Jordan (EGJ) para o cálculo da inversa. Porém, seu tempo para execução verificada é bastante elevado.

Por outro lado, embora a resolução verificada de SELAs com o Método Verificado Baseado no Método de *Newton* seja uma das situações em que é necessário o cálculo da matriz inversa, o mesmo requer apenas uma inversa aproximada, ou seja, não há problema em perder-se parte da exatidão. Cabe lembrar, entretanto, que quanto mais R se aproxima da inversa de A, mais rapidamente o método converge. Em [KOL07] os autores obtiveram ótimos resultados ao otimizar o *solver* apresentado em [KOL06]. Para tal, reduziram o tempo gasto pelo C-XSC na inversão de matrizes simplesmente eliminando os trechos de código da biblioteca responsáveis pela alta exatidão. Partindo desse resultado, trabalhos seguintes, como [KOL08c], obtiveram ganhos adicionais eliminando por completo C-XSC e delegando a responsabilidade de calcular a inversa para bibliotecas numéricas otimizadas, como à rotina *pdgetri* do ScaLAPACK.

Contudo, existem obstáculos à ideia de adotar-se a mesma estratégia de [KOL08c] em processadores *multicore*. Delegar a inversão da matriz A à rotina *dgetri* do LAPACK, por exemplo, não traz ganhos satisfatórios, conforme será visto no Capítulo 5. Nesse contexto, optou-se por empregar a biblioteca PLASMA para essa finalidade. Com isso, é possível se beneficiar das diversas otimizações, apresentadas no Capítulo 2, que a biblioteca disponibiliza para processadores *multicore*. Entretanto, a biblioteca PLASMA se encontra em desenvolvimento e a implementação disponível até o momento da conclusão do projeto aqui descrito (versão 2.1.0 de 15 de novembro de 2009) não oferece rotinas para o cálculo da matriz inversa.

Por outro lado, tem-se já disponíveis as rotinas para fatoração LU, QR e *Cholesky* de matrizes. Adicionalmente, a rotina *PLASMA_dgesv* permite calcular a solução de um SELA no formato $AX = B$ em que A é uma matriz nxn e X e B são matrizes nx$nrhs$ (*number of right hand sides*), ou seja, número de colunas da matriz B). A decomposição LU com pivotamento parcial e trocas de linhas é executada como passo intermediário na fatoração de A. Na sequência, a forma fatorada de A é utilizada para resolver o SELA.

De acordo com as propriedades Álgebra Linear, tem-se que a multiplicação de uma matriz A por sua inversa R resulta na matriz identidade I. Acrescentando-se a isso o fato de que a rotina *PLASMA_dgesv* permite a B, ao lado direito da equação, assumir a estrutura de uma matriz, empregou-se tal rotina como solução neste projeto. Para tal, a rotina é parametrizada atribuindo-se à matriz A os elementos da matriz a qual se deseja inverter e para B a identidade I de A. Dessa forma, a resolução X do sistema dada por *PLASMA_dgesv* coincide com a inversa aproximada R de A.

Conforme visto no Capítulo 2, para que a PLASMA obtenha bom desempenho é necessário vinculá-la a implementações otimizadas, porém sequenciais, da BLAS. Entretanto, neste trabalho utilizou-se a MKL *multithread* devido às paralelizações que essa oferece para os demais passos do algoritmo nos quais a PLASMA não é utilizada. Para solucionar tal conflito, introduziu-se no algoritmo chamadas à função *mkl_set_num_threads*. A *mkl_set_num_threads* recebe como parâmetro um inteiro que representa o número máximo de *threads* que podem ser disparadas pelo ambiente *multithreading* OpenMP da MKL. Assim, faz-se uma chamada *mkl_set_num_threads(1)* como passo anterior à execução da *PLASMA_dgesv*. Terminada a rotina da PLASMA, executa-se novamente *mkl_set_num_threads* passando como parâmetro o número de *cores* disponíveis. Com essa abordagem, obtém-se os benefícios da biblioteca PLASMA no escopo da Álgebra Linear e as otimizações, em nível de instrução, da MKL nos trechos mais internos das operações.

4.3.2 Cálculo da Matriz de Inclusão $[C]$

A otimização do Passo 4, ou seja, do cálculo da matriz de inclusão $[C]$, deu-se através da paralelização do cálculo dos limites inferior e superior de $[C]$, além do emprego da função *dgemm* da MKL para implementação da multiplicação de matrizes. Para tal, aplicou-se programação *multithread*, com *threads* POSIX (*Portable Operating System Interface*), baseada no conceito de afinidade de processador (*thread affinity*).

Afinidade de processador é uma ferramenta que permite definir o(s) processador(es) em que um *thread* ou processo deve ser executado. Embora não seja possível impedir as interrupções causadas pelo escalonador do sistema operacional na execução dos *threads*, ao utilizar *thread affinity* é possível garantir que um *thread* não será atribuído a um processador diferente daquele desejado após uma interrupção. Tal controle é de fundamental importância para certificar-se de que um *thread* não será executado em um processador configurado com modo de arredondamento diferente do esperado. Dessa forma, garante-se que as propriedades das operações da Aritmética Intervalar serão respeitadas e, portanto, os resultados confiáveis. Além disso, a afinidade de processador permite obter ganhos de desempenho, pois viabiliza, por exemplo, definir que *threads* que colaboram entre si em uma determinada operação sejam escalonados de forma mais eficiente.

Nesse contexto, a estratégia adotada é criar *threads* POSIX dividindo igualmente dentre o número de *cores* disponíveis o cálculo dos limites superior e inferior de $[C]$. Para isso, os *threads* são definidos com afinidade de processador e, então, atribuídos estaticamente aos *cores* configurados com um mesmo modo de arredondamento. Criou-se um algoritmo para tal divisão, o qual é descrito pelo Algoritmo 4.5.

Algoritmo 4.5: Função para distribuição balanceada, dentre os *threads*, dos *cores* disponíveis.

```
1:  int procsDispo[numCores]
2:  if numCores > 1 then
3:     if numCores MOD 2 == 1 then
4:        int procsUpWar_VT[(numCores/2)+1]
5:     else
6:        int procsUpWar_VT[numCores/2]
7:     end if
8:     int procsDownW_VT[numCores/2]
9:     for i=0, j=0; i < numCores; i++ do
10:       procsDispo[i]=i
11:       CPU_SET(i, &procGERAL_SET)
12:       if numCores MOD 2 == 1 then
13:          procsDownW_VT[j] = i
14:          CPU_SET(i, &procDownWa_SET)
15:          j++
16:       else
17:          procsUpWar_VT[j] = i
18:          CPU_SET(i, &procUpWard_SET)
19:       end if
20:    end for
21: else
22:    int procsUpWar_VT[numCores]
23:    int procsDownW_VT[numCores]
24:    CPU_SET(0, &procGERAL_SET)
25:    CPU_SET(0, &procDownWa_SET)
26:    CPU_SET(0, &procUpWard_SET)
27: end if
```

O Algoritmo 4.5 tem como objetivo efetuar o balanceamento dos recursos, assim, tem-se $n/2$ *threads* calculando cada limite, todos escalonados em paralelo. Utiliza-se, para tal, três variáveis do tipo "*cpu_set_t*", disponíveis no cabeçalho *sched.h*, e três vetores de inteiros sendo, em ambos os casos, uma variável para definição dos processadores relativos ao cálculo do limite superior, outra para o cálculo do limite inferior e uma terceira que define o conjunto completo de processadores disponíveis. A definição do conjunto completo visa facilitar as alterações nos arredondamentos dos processadores e o escalonamento daquelas operações em que se tem todos os *cores* cooperando e utilizando um mesmo modo de arredondamento, como, por exemplo, na inversão da matriz A.

As variáveis do tipo "*cpu_set_t*" definem os conjuntos de máscaras de processadores. Esses são passados como parâmetros para a função *sched_setaffinity* que, por sua vez, é responsável pela configuração da afinidade de processador. Já os vetores de inteiros recebem os identificadores dos processadores e são utilizados pela função que modifica o arredondamento de vários processadores simultaneamente, a qual será descrita na sequência. Verificou-se, através de análise dos arquivos de configuração do *Linux*, que a identificação dos *cores* na arquitetura utilizada (Seção 5.1) segue o seguinte padrão: os *cores* do primeiro processador físico recebem números pares sequenciais iniciando em 0, enquanto os *cores* do segundo processador recebem números ímpares iniciando por 1.

Portanto, para a divisão dos recursos, o Algoritmo 4.5 verifica, inicialmente, o número de processadores a ser utilizado. Caso houver apenas um *core* disponível, então os três vetores e conjuntos de máscaras recebem apenas o identificador 0. Havendo mais de um núcleo disponível e, sendo a quantidade par, os vetores *procsUpWar_VT* e *procsDownW_VT*, assim como os conjuntos "*cpu_set_t*", recebem $n/2$ posições. Os núcleos com identificadores ímpares são atribuídos para o cálculo do limite inferior, enquanto os pares destinam-se ao limite superior. Dessa forma, todos os *threads* que calculam um limite são escalonados nos *cores* de um mesmo processador físico. Com isso, além de aumentar a localidade de dados na memória *cache*, facilita-se a comunicação e sincronização. Em caso de número ímpar de *cores* disponíveis, o limite superior é calculado com $n/2+1$ núcleos.

Iniciado o Passo 4 do algoritmo, a primeira etapa é configurar os modos de arredondamento dos processadores para execução dos cálculos da matriz de inclusão. Visando facilitar esse processo nas execuções com múltiplos *cores*, criou-se uma função que recebe três parâmetros: a direção para a qual o arredondamento deve ser configurado, um vetor de inteiros, preenchido pelo Algoritmo 4.5, que contém os identificadores de todos os processadores cujos modos de arredondamento devem ser alterados para aquela direção e o número de processadores que compõem o vetor. De posse dessas informações, a função executa um laço *for* modificando, para cada processador indicado no vetor, o modo de arredondamento. Utiliza-se, para isso, um conjunto *cpu_set_t* temporário o qual, a cada iteração do laço, é reiniciado e recebe exclusivamente o processador indicado pela posição atual do vetor. Na sequência, a afinidade de processador da função é alterada por uma chamada à *sched_setaffinity* de modo que essa execute somente no processador indicado pelo *cpu_set_t*. Uma vez atribuída ao processador i, a função invoca a rotina *fesetround* para efetuar o chaveamento do modo de arredondamento. O Algoritmo 4.6 descreve a rotina desenvolvida.

Alterados os arredondamentos, cria-se os *threads* para execução dos cálculos. Cada *thread* tem

Algoritmo 4.6: Rotina para alteração do modo arredondamento de um conjunto de processadores.

```
1: int setaArredondamento(int direcao, int *arrayProcs, int numProcs)
2: cpu_set_t processadorAtual
3: for i=0; i<numProcs; i++ do
4:    CPU_ZERO(&processadorAtual)
5:    CPU_SET(arrayProcs[i], &processadorAtual)
6:    if (sched_setaffinity(tid(),sizeof(processadorAtual),&processadorAtual) == -1) then
7:       perror("Erro ao setar arredondamento do processador")
8:       exit(1)
9:    else
10:      fesetround(direcao)
11:   end if
12: end for
```

como instrução inicial a definição de sua afinidade de processador. Assim, uma vez iniciados, os *threads* alteram sua afinidade e permanecem bloqueados até que sua execução seja liberada pelo fluxo principal do programa. Utilizou-se, para a sincronização, três semáforos: um para controle do fluxo de cálculo do limite superior, outro do limite inferior e o terceiro para sincronização com o fluxo principal. Dessa forma, após liberar a execução dos *threads* que calculam os limites, o programa principal permanece bloqueado por um semáforo aguardando o término dos mesmos. Uma vez formados os limites, os *threads* enviam sinal ao semáforo para desbloqueio do fluxo principal.

Desenvolveu-se também uma estratégia alternativa para esse passo do *solver* a qual consiste em utilizar os n processadores para a formação de cada limite. Nesse caso, primeiro seta-se o arredondamento de todos os n processadores para *DOWNWARD* e calcula-se o limite inferior utilizando os n processadores. Na sequência, seta-se o arredondamento dos n processadores para *UPWARD* e calcula-se o limite superior utilizando, novamente, os n processadores. Assim como na abordagem até então discutida, essa emprega a rotina *dgemm* da MKL para a multiplicação de R e A, ou seja, ambas se beneficiam das otimizações relativas à arquitetura do processador.

Avaliou-se, ainda, a estratégia de dividir manualmente as matrizes R e A em um número elevado de pequenas submatrizes, conforme ocorre nos *tiled algorithms*, e computá-las em diversos *threads* em paralelos utilizando a função *dgemm* sequencial da BLAS. Porém, não se obteve ganhos nesse aspecto, pois o *overhead* introduzido pela quebra, sincronização e reconstrução das matrizes eliminou os ganhos oriundos da operação com blocos de menor magnitude levando a tempos de execução iguais ou superiores aos obtidos com a *dgemm multithreaded*. Adicionalmente, a complexidade de programação introduzida é bastante elevada. Cabe observar, porém, que esse é um resultado esperado, dado que a MKL é altamente otimizada para o processador utilizado.

4.4 Considerações Finais

Este capítulo apresentou o desenvolvimento da solução proposta pelo presente trabalho. Primeiramente, foi descrita uma versão inicial, a qual teve três importantes contribuições: comprovou a

viabilidade técnica em executar a proposta do trabalho empregando as ferramentas e *hardware* disponíveis; Possibilitou o levantamento do custo computacional do algoritmo de modo que um plano de otimização pudesse ser adequadamente traçado; Permitiu a validação dos resultados numéricos.

Na sequência, abordou-se a paralelização da solução proposta explicitando o emprego dos conceitos discutidos até então. Constatou-se que, dentre os dois trechos do algoritmo com maior potencial para otimização de desempenho, um deles pode ser tratado sem grande complexidade aplicando-se uma identidade matemática à função *PLASMA_dgesv*. Com isso, simula-se um procedimento de inversão de matriz, dado que não há tal rotina disponível na versão atual da biblioteca PLASMA. Já no segundo trecho, duas estratégias com pequenas diferenças foram adotadas. O próximo capítulo descreve em detalhes os testes executados no *solver* resultante desse processo de otimização.

5. AVALIAÇÃO DA SOLUÇÃO OTIMIZADA

A avaliação dos resultados deste projeto deu-se de duas formas: através da análise de desempenho da ferramenta, executando-a em diferentes números de processadores, e de testes de exatidão com os resultados calculados. Uma vez que o objetivo principal do trabalho é reduzir o tempo de computação, os testes de desempenho visaram medir os valores de *speedup* obtidos pela solução otimizada. Em relação aos testes de exatidão, embora já realizados anteriormente com a versão inicial do *solver*, fez-se necessário repeti-los para garantir que a exatidão dos resultados numéricos não tenha sido comprometida pelas otimizações matemáticas e modificações na implementação do algoritmo.

Este capítulo apresenta, inicialmente, o ambiente no qual os testes foram executados. Na sequência, descreve-se a avaliação de exatidão e compara-se a mesma com a versão original do *solver*. Por fim, os testes de desempenho são apresentados e discutidos.

5.1 Ambiente de Execução dos Testes

As avaliações da solução final deste projeto, descritas nas próximas seções, foram realizadas em um computador *Dell PowerEdge* R610 disponibilizado pelo LAD (Laboratório de Alto Desempenho) da PUC-RS. As configurações de *software* e versões das bibliotecas utilizadas são as seguintes:

- Sistema Operacional *Linux*, *kernel* versão 2.6.28-11-*server*, distribuição *Ubuntu* 9.04;
- Compilador gcc versão 4.3.3;
- *Intel* MKL 10.2.2.025 como implementação otimizada da BLAS e LAPACK;
- Biblioteca PLASMA 2.1.0;

Em relação ao *hardware*, a configuração disponível é a seguinte:

- 2 processadores *Quad-core Intel(R) Xeon(R)* CPU E5520 @ 2.27GHz;
- Memórias *cache* L1 de 128KB, L2 de 1MB e L3 de 8MB compartilhada. Todos os níveis possuem ECC (*Error Correction Code*) e são associativos, sendo os níveis L1 e L2 associativos por grupos de 8 vias e o L3 associativo por grupo de 16 vias;
- 16 GB de memória RAM DDR3 @ 1066 MHz.

Cabe observar que, durante a execução dos testes, a tecnologia de *Hyperthreading* dos processadores encontrava-se habilitada podendo, portanto, gerar interferência nos resultados de desempenho. Entretanto, uma vez que o *solver* desenvolvido emprega afinidade de processador, elimina-se a possibilidade de escalonamento dos *threads* nos processadores virtuais. De modo a minimizar

tal influência, incluiu-se também na versão inicial da ferramenta chamadas à rotina para controle de afinidade de processador. A Figura 5.1 ilustra o diagrama de blocos para um sistema com 2 processadores *Quad-core Intel(R) Xeon(R)* CPU E5520.

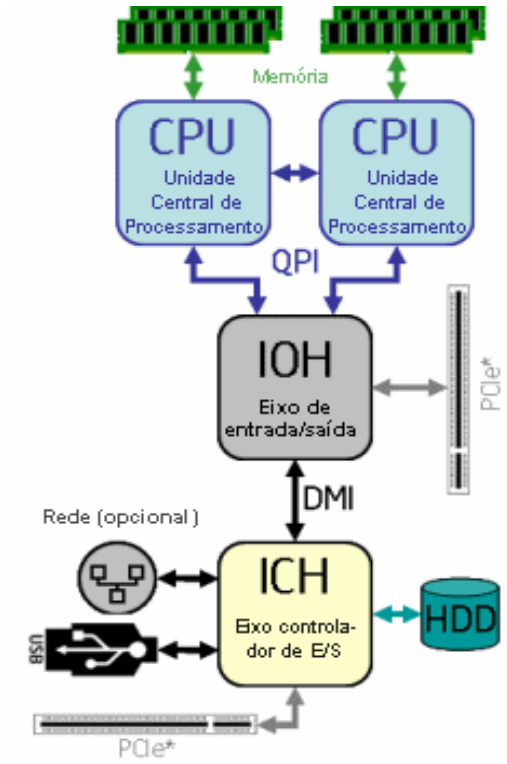

Figura 5.1: Diagrama de Blocos 2x *Quad-core Intel(R) Xeon(R)* CPU E5520. Adaptado de [INT10]

5.2 Avaliações de Exatidão

Visando verificar a exatidão dos resultados e certificar-se do não comprometimento da mesma devido às modificações introduzidas durante a paralelização do algoritmo, executou-se um processo análogo àquele realizado para avaliar a solução inicial. Gerou-se sistemas baseados em um formato clássico de matriz e também constituídos por números aleatórios entre 0 e 1. Na sequência, resolveu-se tais sistemas com as versões original e otimizada da ferramenta para, então, comparar os resultados dessas.

Na avaliação da solução paralela, variou-se o número de *cores* em que a mesma executa. Com isso, é possível certificar-se de que, no ambiente paralelo, as operações de sincronização entre os *threads* bem como as trocas de contexto dos processadores não afetam a exatidão, ou seja, o algoritmo não está sujeito a condições de corrida. Todos os testes foram repetidos 10 vezes para cada situação e obtiveram, em todos os casos, resultados numéricos idênticos.

O sistema clássico utilizado é formado pela matriz de ordem 10 dada pela fórmula de *Booth-royd/Dekker* (Equação 4.1), conforme definido no Capítulo 4. Cabe lembrar que tal matriz é um exemplo de matriz mal condicionada, seu condicionamento para $n = 10$ é de $1,09.10^{+15}$. De modo

a tornar os dados de entrada intervalares, preencheu-se a matriz relativa ao raio de A e o vetor correspondente ao raio de b com o valor $0,1.10^{-10}$.

A Tabela 5.1 apresenta a solução exata do sistema e os resultados numéricos obtidos pelo *solver* utilizando: (a) Implementação inicial; (b) Implementação otimizada executando em 1 núcleo; (C) Implementação otimizada executando em 8 núcleos. Executou-se tal avaliação variando o número de *cores* entre 1 e 8. Porém, uma vez que os resultados obtidos com quantidades intermediárias de *cores* foram exatamente iguais aos obtidos pela execução com 8 núcleos, omitiu-se os mesmos na Tabela 5.1. Com objetivo de facilitar a visualização, os resultados são apresentados na notação Ínfimo-Supremo e os valores arredondados para 5 casas decimais, pois o *solver* efetua todos os cálculos com variáveis de exatidão dupla (tipo *double* do C++).

Tabela 5.1: Resultados numéricos do *solver* otimizado na resolução de um Sistema de *Boothroyd/Dekker* de ordem 10.

		Inicial		**Paralela - 1 núcleo**		**Paralela - 8 núcleos**	
X	Res. Exato	Ínfimo	Supremo	Ínfimo	Supremo	Ínfimo	Supremo
0	0,0	-0.00001	0.00001	-0,00027	0,00027	-0,00027	0,00027
1	1,0	0.99988	1.00010	0,99935	1,00065	0,99935	1,00065
2	-2,0	-2.00050	-1.99939	-2,00100	-1,99898	-2,00100	-1,99898
3	3,0	2.99767	3.00194	2,99750	3,00242	2,99750	3,00242
4	-4,0	-4.00610	-3.99264	-4,00684	-3,99290	-4,00683	-3,99290
5	5,0	4.97990	5.01662	4,98082	5,01842	4,98083	5,01841
6	-6,0	-6.04061	-5.95079	-6,04506	-5,95301	-6,04506	-5,95301
7	7,0	6.88950	7.09104	6,89454	7,10101	6,89461	7,10094
8	-8,0	-8.19026	-7.76874	-8,21120	-7,77928	-8,21119	-7,77928
9	9,0	8.54359	9.37504	8,56438	9,41647	8,56470	9,41614

Verifica-se, através dos resultados da Tabela 5.1, que o requisito de exatidão da ferramenta é atendido em todas as avaliações realizadas. No caso da solução paralela executando em 1 núcleo, o menor diâmetro é de $0,00054$ para uma exatidão de 5 casas decimais e o maior de $0,85209$. A maior amplitude ocorre na posição 9 de x, as demais apresentam diâmetros bastante menores e, em todos os casos, contêm a solução exata. Comparando-se a execução do algoritmo paralelo em 8 *cores* com a execução em 1 *core* percebe-se uma sensível diferença nos resultados. Os valores obtidos foram semelhantes para as posições 0, 1, 2, 3 e 6. Entretanto, para as posições 4, 5, 7, 8 e 9 houve uma pequena melhora na execução com 8 *cores*, a qual apresentou diâmetro mais amplo no valor de $0,85144$. Já em relação ao *solver* inicial, houve um leve aumento dos diâmetros em geral. Apesar disso, todos os intervalos contêm a solução exata. Adicionalmente, cabe observar que mesmo um diâmetro mais amplo é considerado uma solução verificada, especialmente se tratando de sistemas mal condicionados como é o caso do gerado pela Equação de *Boothroyd/Dekker*.

O próximo teste de exatidão se deu através da resolução de um sistema de ordem 10 com valores para A e b gerados aleatoriamente entre 0 e 1. O número de condicionamento do sistema é igual a $1.56.10^2$. Dessa forma, tem-se a avaliação do *solver* considerando um sistema bem condicionado. Gerou-se diversos outros sistemas de maneira análoga e os testes levaram a resultados

de comportamento semelhantes. A Tabela 5.2 apresenta os resultados numéricos obtidos para tal sistema. Novamente, executou-se a implementação otimizada do *solver* variando o número de núcleos entre 1 e 8. Entretanto, nesse caso obteve-se resultados exatamente iguais para todas as situações, o que é esperado pelo fato de o sistema ser bem condicionado. Assim, a Tabela 5.2 condensa o resultado das execuções do algoritmo paralelo, sem distinção do número de *cores*, e os valores são apresentados com exatidão de 10 casas decimais.

Tabela 5.2: Resultados numéricos do *solver* otimizado na resolução de um sistema na resolução de um sistema aleatório de ordem 10.

	Inicial		**Paralela**	
X	Ínfimo	Supremo	Ínfimo	Supremo
0	0,5054478992	0,5054479183	0,5054478992	0,5054479183
1	-0,2703378490	-0,2703378227	-0,2703378490	-0,2703378227
2	-1,9912072093	-1,9912071468	-1,9912072093	-1,9912071468
3	1,6842676360	1,6842676980	1,6842676360	1,6842676980
4	1,0781538443	1,0781538745	1,0781538443	1,0781538745
5	-0,0046079482	-0,0046079310	-0,0046079482	-0,0046079310
6	1,5142040332	1,5142040828	1,5142040332	1,5142040828
7	0,6424340341	0,6424340694	0,6424340341	0,6424340694
8	-0,7596261221	-0,7596261024	-0,7596261221	-0,7596261024
9	-0,1501569783	-0,1501569534	-0,1501569783	-0,1501569534

Conforme esperado, obteve-se resultados bastante satisfatórios para o sistema bem condicionado. No exemplo da Tabela 5.2, exceto pelas posições 0 e 2, os valores apresentaram diâmetro 0 até a sétima casa decimal. Já nas posições 0 e 2, o diâmetro corresponde a 1.10^{-7}, ou seja, variação de apenas uma unidade na última casa decimal. Somente após a oitava casa decimal os diâmetros apresentaram crescimento. Tal situação confirma que, de fato, para problemas bem condicionados o *solver* encontra *enclosures* iguais ou muito próximas da solução exata. Observa-se, também, que em tal situação os valores calculados pela implementação otimizada são exatamente iguais aos obtidos pela solução inicial. Diante de tais resultados, é possível afirmar que as estratégias de otimização adotadas no desenvolvimento do *solver* paralelo não comprometeram a confiabilidade dos resultados. Cabe ainda observar que, para problemas bem condicionados, além da maior exatidão a ferramenta apresenta também melhor desempenho. Isso porque, para o sistema mal condicionado, o passo de verificação do *solver* obteve êxito em encontrar a *enclosure* após três iterações, enquanto ao resolver sistemas bem condicionados as *enclosures* foram obtidas na segunda iteração.

5.3 Avaliações de Desempenho

Executou-se testes de desempenho com objetivo de avaliar o sucesso das estratégias adotadas na otimização da ferramenta. Para esses, gerou-se matrizes A e vetores b de ordens variadas formados por valores aleatórios entre 0 e 1. Dessa forma, avalia-se também a escalabilidade da solução desenvolvida e tamanhos de problema para os quais a mesma pode não ser adequada.

Da mesma forma que no Capítulo 4, as contabilizações de tempo se deram com o emprego de funções das bibliotecas do *Linux*. O número de amostras utilizado nos testes foi 10. Assim, para cada tamanho de problema e variação no número de *threads* e/ou algoritmo, executou-se 10 vezes a contabilização do tempo consumido por cada passo. Na sequência, eliminou-se os dois valores extremos, ou seja, os tempos maior e menor de execução. Obteve-se o resultado final através da média aritmética das amostras restantes. Na eliminação dos extremos foram considerados os tempos para cada passo em particular e não para as execuções como todo. Pretende-se, com isso, amenizar vieses relacionados às atividades do sistema operacional. Adicionalmente, cabe observar que o computador utilizado encontra-se em um *cluster* o qual é acessado remotamente e, portanto, o ambiente de teste não é totalmente controlado. Todas as medições de tempo foram realizadas através de novas execuções do programa.

O custo das operações de leitura dos arquivos contendo a matriz A e o vetor b foram, também, contabilizados em todas as situações, porém, não constituem os totais descritos nas tabelas. Tais tempos serão apresentados juntamente com o número de condicionamento dos sistemas. Todas as demais operações de entrada e saída são consideradas, ou seja, seus tempos encontram-se inclusos nas medições. Isso porque, a operação de leitura dos arquivos é um passo anterior à execução do *solver* propriamente dito, enquanto as demais, como, por exemplo, alocação de memória, são operações componentes dos passos de resolução do sistema.

De modo a facilitar a visualização dos tempos, adotou-se para todas as tabelas a seguinte convenção: na coluna "Algoritmo", o símbolo "S.Al." representa o **Algoritmo Inicial Alterado** enquanto o símbolo "Para." refere-se à **Implementação Paralela do *Solver***. A coluna "*Cores*" apresenta o número de núcleos do processador empregados na respectiva execução. A coluna "Total" apresenta o tempo total consumido para execução completa do *solver*, desconsiderando a leitura dos arquivos. Por fim, as colunas de rótulo iniciado por "Passo" referem-se aos passos de execução da ferramenta. Nestes casos, condensou-se os tempos consumidos pelos passos de 6 a 15 em um único tempo. Portanto, os tempos consumidos pelos passos do Algoritmo 4.1 são transpostos nas tabelas a seguir da seguinte forma:

1. Computação de R, matriz inversa aproximada de A (Passo 1);
2. Cálculo da solução aproximada de x (Passo 2);
3. Cálculo da inclusão do resíduo de x, ou seja, cálculo do z (Passo 3);
4. Computação da inclusão da matriz de iteração (pré-condicionamento), ou seja, cálculo de $[C]$ (Passo 4);
5. Inicialização do vetor intervalar de máquina (Passo 5);
6. Refinamento e teste de inclusão, passo final da verificação (passos 6 à 15);
7. Execução completa (incluindo operações de entrada e saída, exceto leitura dos arquivos).

O **Algoritmo Inicial Alterado** referido nas tabelas corresponde a uma paralelização parcial do *solver* inicial. Trata-se de uma modificação do algoritmo original em que se manteve inalterada a maior parte dos passos, incluindo as chamadas às rotinas MKL. A alteração, neste caso, refere-se à inclusão no algoritmo de chamadas às rotinas da MKL que manipulam o número de *threads* para que as operações da biblioteca passem a executar no modo *multithread*. Compilou-se essa versão do *solver* vinculada à implementação *multithread* da MKL. Com isso, obtém-se dois tipos de avaliação: a primeira sobre o comportamento das rotinas *multithread* do LAPACK, em especial no passo de inversão da matriz, quando executadas com diferentes quantidades de *threads* e *cores*. A segunda visa avaliar uma estratégia alternativa para paralelização do Passo 4, ou seja, do pré-condicionamento do Sistema Intervalar (cálculo de $[C]$).

Tal estratégia corresponde a formar os limites de $[C]$ sequencialmente, porém utilizando-se os n *cores* disponíveis para calcular cada um dos limites. Basicamente, mantém-se a computação dos limites superior e inferior da matriz $[C]$ em momentos separados, ou seja, calcula-se o limite inferior e, na sequência, o superior. Porém, dado que o trecho de maior custo computacional no cálculo dos limites corresponde à operação de multiplicação de matrizes e que, além disso, tal operação é executada por uma chamada à rotina *dgemm* da BLAS, o fato de utilizar a MKL *multithread* permite, por si só, obter ganhos de desempenho para este passo. Assim, tem-se uma abordagem alternativa, em que cada limite é calculado em n núcleos, à paralelização original, na qual cada limite é formado em $n/2$ *cores*.

A Tabela 5.3 apresenta os resultados obtidos para a resolução de um sistema de ordem 1.000. O tempo médio consumido para a leitura dos arquivos nesta situação é de $0,089153$ segundos.

Tabela 5.3: Tempos médios, em segundos, consumidos pelo *solver* otimizado para resolução de um sistema aleatório de ordem 1.000.

Algoritmo	*Cores*	Passo 1	Passo 2	Passo 3	Passo 4	Passo 5	P. 6 a 15	Total
S.Al.	1	0,482017	0,011851	0,036666	0,801094	0,000050	0,132465	1,464143
S.Al.	8	0,438240	0,011796	0,035259	0,305541	0,000030	0,133238	0,924104
Para.	1	0,386321	0,011537	0,039588	0,808041	0,000013	0,112830	1,358328
Para.	2	0,523196	0,011844	0,037804	0,520011	0,000011	0,114347	1,207212
Para.	3	0,376516	0,012335	0,037004	0,616041	0,000010	0,113486	1,020974
Para.	4	0,252078	0,014298	0,149682	0,491420	0,000012	0,114273	1,194762
Para.	5	0,261205	0,011537	0,135444	0,370268	0,000011	0,114273	0,892737
Para.	6	0,272508	0,014355	0,142991	0,480497	0,000012	0,167381	1,077742
Para.	7	0,197835	0,015793	0,166651	0,497059	0,000011	0,112837	0,990186
Para.	8	0,239283	0,014553	0,248553	0,561926	0,000012	0,255576	1,319901

Observa-se que, para os resultados da Tabela 5.3, as variações de tempo foram desprezíveis. Esse comportamento é esperado devido ao tamanho do problema, pois, embora um sistema de ordem 1.000 seja considerado de grande porte, trata-se de um problema pequeno para o contexto das tecnologias aplicadas. Com isso, de modo geral os ganhos oriundos da paralelização costumam não compensar o *overhead* introduzido pelos mecanismos de controle e sincronização. De fato, percebe-se que, em alguns casos, a execução com número maior de *threads* obteve desempenho

inferior às execuções com quantidades menores. Ademais, em intervalos de tempo tão pequenos (todos menores do que um segundo), a imprecisão de medição e/ou cálculo das médias pode ser suficiente para gerar tais diferenças.

Assim, para os próximos testes aumentou-se o valor de n dos sistema para quantidades com as quais se espera obter resultados mais adequados. A próxima avaliação deu-se, portanto, com um sistema de ordem 6.000. O número de condicionamento do mesmo é igual a $2,61.10^6$ e o tempo médio consumido pela leitura dos arquivos com matrizes e vetores foi de $3,51$ segundos. Os tempos são apresentados na Tabela 5.4.

Tabela 5.4: Tempos médios, em segundos, consumidos pelo *solver* otimizado para resolução de um sistema aleatório de ordem 6.000.

Algoritmo	*Cores*	Passo 1	Passo 2	Passo 3	Passo 4	Passo 5	P. 6 a 15	Total
S.Al.	1	124,558584	0,735408	2,124209	142,875247	0,000109	6,235101	294,991597
S.Al.	8	119,270953	0,734500	2,073683	34,241740	0,000124	6,142029	169,066108
Para.	1	74,355709	0,719932	2,019119	143,298416	0,000116	5,219667	225,612958
Para.	2	37,591799	0,739751	1,977294	76,615387	0,000089	5,326843	122,251161
Para.	3	25,710658	0,716448	1,946953	68,845745	0,000091	5,128198	102,348092
Para.	4	19,325366	0,738670	1,969044	43,072403	0,000088	5,308355	70,413926
Para.	5	16,837877	0,714760	2,015751	41,036739	0,000090	5,151568	65,756784
Para.	6	14,043252	0,737538	2,031508	33,147206	0,000089	5,328518	55,288110
Para.	7	12,147409	0,713390	2,030166	31,841523	0,000088	5,155478	51,888053
Para.	8	10,939175	0,727385	2,126578	30,044676	0,000089	5,328105	49,166008

Na avaliação do sistema de ordem 6.000 já é possível perceber os benefícios advindos das otimizações. A Tabela 5.5 apresenta os *speedups* relativos à Tabela 5.4, onde a coluna "*Cores*" representa o número de núcleos no qual o *solver* paralelo foi executado. A coluna "Sp T.T.Par." apresenta os *speedups* considerando os tempos totais de execução do algoritmo paralelo executando em "*cores*" núcleos em relação ao algoritmo paralelo executando em 1 núcleo, ou seja, $S_p = \frac{T\_Para.(n)}{T\_Para.(1)}$.

Na coluna "Sp T.T.Seq." são descritos os *speedups* obtidos pela execução paralela em "*cores*" núcleos em comparação ao algoritmo original, ou seja, $S_p = \frac{T\_Para.(n)}{T\_S.Al.(1)}$. Tal comparação é importante por permitir demonstrar os ganhos reais obtidos, ou seja, os benefícios da solução final em relação à versão inicial desenvolvida com o LAPACK. Ao calcular-se o *speedup* apenas com base na versão paralela executando em 1 núcleo, negligencia-se as otimizações introduzidas pelo emprego da biblioteca PLASMA. Isso porque, conforme visto anteriormente, as otimizações da PLASMA não se resumem apenas ao paralelismo das tarefas mas, também, a novas abordagens no gerenciamento dos dados e no escalonamento das operações de maneira mais adequada aos computadores *multicore*.

Cabe, ainda, avaliar em separado os *speedups* obtidos apenas nos passos que foram otimizados para execução em paralelo, uma vez que, ao comparar-se somente os tempos totais de execução, tem-se sobre os *speedups*, a influência dos diversos trechos não paralelizados. A coluna "Sp P1. Par." descreve os *speedups* do Passo 1 do algoritmo (cálculo da matriz inversa) relacionando o *solver* paralelo em execução com 1 e com "*cores*" núcleos. Já na coluna "Sp P1. Seq." são apresentados os *speedups* relativos ao algoritmo paralelo executando em "*cores*" núcleos em comparação à implementação desenvolvida com o LAPACK. De maneira análoga, as colunas "Sp P4. Par." e "Sp

P4. Seq." descrevem os *speedups* do Passo 4 (pré-condicionamento do sistema) comparando-se, respectivamente, a execução do *solver* paralelo em "*cores*" núcleos com sua execução em 1 núcleo e com a solução inicial.

Tabela 5.5: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema aleatório de ordem 6.000.

Cores	Sp T.T.Par.	Sp T.T.Seq.	Sp P1. Par.	Sp P1. Seq.	Sp P4. Par.	Sp P4. Seq.
2	1,85	2,41	1,98	3,31	1,87	1,86
3	2,20	2,88	2,89	4,84	2,08	2,08
4	3,20	4,19	3,85	6,45	3,33	3,32
5	3,43	4,49	4,42	7,40	3,49	3,48
6	4,08	5,34	5,29	8,87	4,32	4,31
7	4,35	5,69	6,12	10,25	4,50	4,49
8	4,59	6,00	6,80	11,39	4,77	4,76

A Figura 5.2 ilustra os dados da Tabela 5.5. Observa-se que a coluna "Sp T.T.Par." iniciou com *speedup* próximo ao linear e, gradualmente, afastou-se do mesmo. Já sua equivalente em relação ao *solver* inicial, ou seja, a linha "Sp T.T.Seq.", apresentou, primeiramente, *speedup* superlinear, porém, decresceu gradualmente ficando abaixo da reta linear a partir de 5 *cores*. Esse resultado pode ser explicado pela influência dos trechos sequenciais, uma vez que o cálculo de tal reta considera o tempo total de execução. A análise do mesmo em conjunto com as colunas seguintes reforça tal conclusão.

A reta "Sp P1. Par." apresentou *speedup* bastante próximo ao linear nos primeiros pontos. Após, iniciou-se um distanciamento pequeno e gradual. Acredita-se que tal situação seja resultado da granularidade do sistema. Já a coluna "Sp P1. Seq.", conforme esperado, manteve *speedup* superlinear para todos os valores. A avaliação dessas duas retas evidencia os fortes ganhos de desempenho advindos do emprego da biblioteca PLASMA, principalmente ao comparar-se com os resultados apresentados pelo LAPACK.

Por fim, as retas "Sp P4. Par." e "Sp P4. Seq." apresentaram, conforme esperado, comportamentos bastante similares ao longo de todo o gráfico. Ambas iniciaram próximas ao *speedup* linear e se distanciaram do mesmo com o aumento do número de *cores*. Acredita-se que, assim como na inversão da matriz, tal comportamento deva-se à granularidade do sistema. Portanto, espera-se que aumentando a ordem do mesmo os *speedups* se mantenham mais próximos à reta linear.

A próxima avaliação de desempenho foi realizada considerando um sistema de ordem 10.000. O número de condicionamento do mesmo é igual a $1,52.10^7$ e o tempo médio consumido pela leitura dos arquivos com matrizes e vetores foi de $15,38$ segundos. Os tempos contabilizados são apresentados na Tabela 5.6. Os *speedups* obtidos para tal sistema são apresentados na Tabela 5.7 e ilustrados na Figura 5.3.

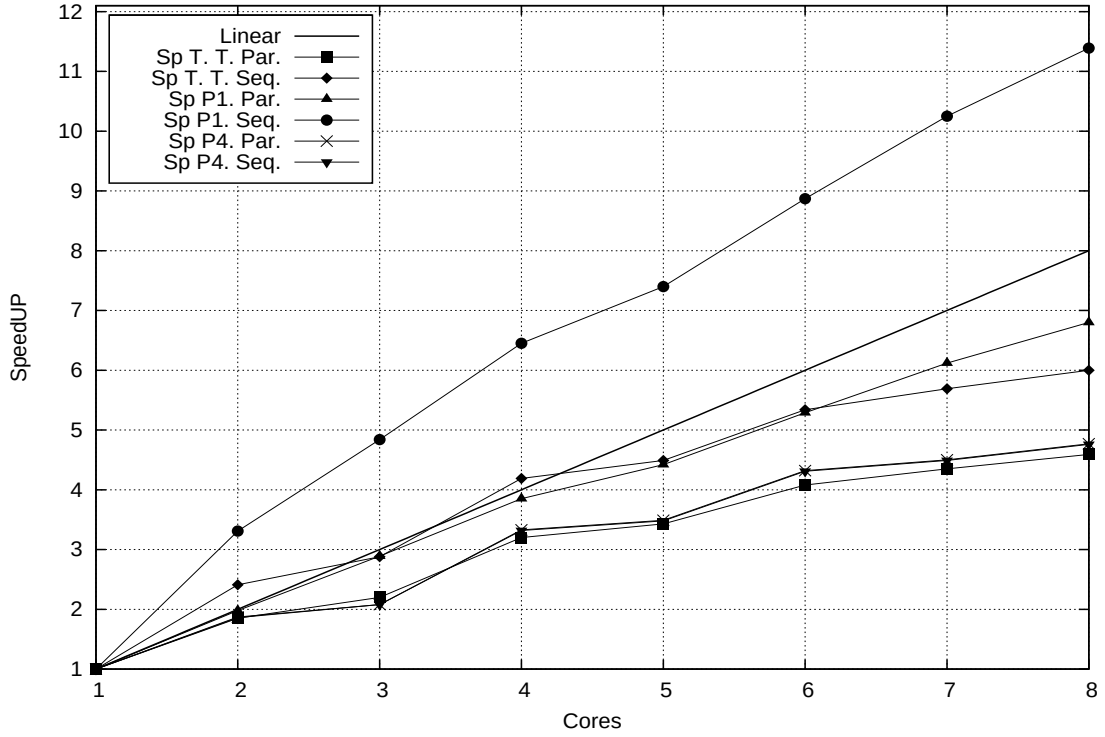

Figura 5.2: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema gerado aleatoriamente de ordem 6.000.

Tabela 5.6: Tempos médios, em segundos, consumidos pelo *solver* otimizado para resolução de um sistema aleatório de ordem 10.000.

Algoritmo	*Cores*	Passo 1	Passo 2	Passo 3	Passo 4	Passo 5	P. 6 a 15	Total
S.Al.	1	584,086666	2,304787	7,712834	653,770835	0,000135	22,081504	1.351,004891
S.Al.	8	556,433856	2,296160	7,039239	121,834738	0,000173	22,342203	731,123191
Para.	1	341,735230	2,253594	6,380963	655,898987	0,000166	18,965825	1.025,234764
Para.	2	171,948754	2,296224	6,247250	348,158280	0,000133	20,053259	548,703898
Para.	3	116,454460	2,239915	5,994912	311,705933	0,000135	19,010674	455,406028
Para.	4	87,704754	2,294053	6,254536	194,079865	0,000134	19,992894	310,326235
Para.	5	74,958302	2,242206	6,010089	182,959313	0,000136	18,937187	285,107231
Para.	6	62,890188	2,294290	6,297094	147,081513	0,000135	19,757659	238,320876
Para.	7	54,831938	2,240176	6,088160	139,760971	0,000138	18,745075	221,666457
Para.	8	48,598866	2,295655	6,293763	133,515389	0,000133	19,988863	210,692667

A análise dos dados para o sistema de ordem 10.000 confirma a hipótese de que um sistema de maior porte do que aquele de ordem 6.000 poderia alcançar resultados ainda melhores. De fato, em todos os cálculos obteve-se valores de *speedup* maiores para $n = 10.000$. Acredita-se que esse fato seja devido ao tamanho dos blocos de dados carregados na memória *cache* serem mais adequados para exploração da localidade de dados e do paralelismo, em especial no nível 3 da BLAS, o qual se sabe tem desempenho diretamente dependente do tamanho dos blocos de dados.

De modo geral, em termos de comportamento ao longo do gráfico, as curvas de *speedup* apresentaram-se semelhantes àquelas da Figura 5.2, embora com valores majorados e, portanto, mais próximos da reta linear. Cabe ressaltar que a reta "Sp P1. Seq.", conforme esperado, manteve

Tabela 5.7: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema aleatório de ordem 10.000.

Cores	Sp T.T.Par.	Sp T.T.Seq.	Sp P1. Par.	Sp P1. Seq.	Sp P4. Par.	Sp P4. Seq.
2	1,87	2,46	1,99	3,40	1,88	1,88
3	2,25	2,97	2,93	5,02	2,10	2,10
4	3,30	4,35	3,90	6,66	3,38	3,37
5	3,60	4,74	4,56	7,79	3,58	3,57
6	4,30	5,67	5,43	9,29	4,46	4,44
7	4,63	6,09	6,23	10,65	4,69	4,68
8	4,87	6,41	7,03	12,02	4,91	4,90

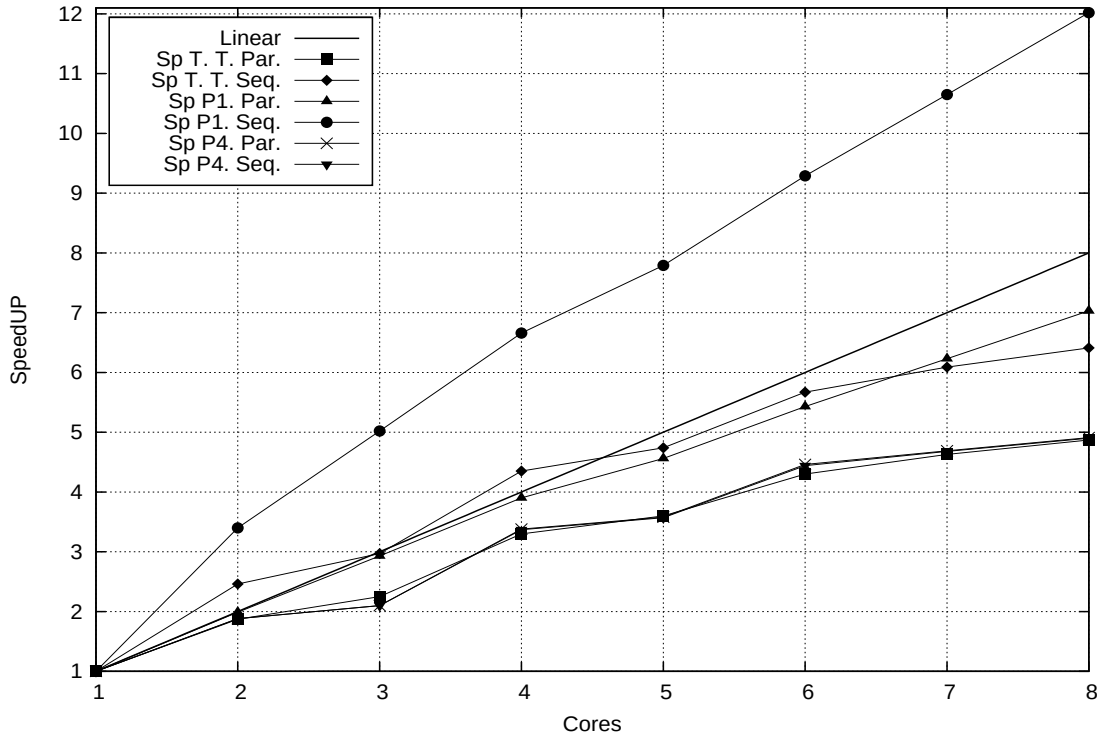

Figura 5.3: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema gerado aleatoriamente de ordem 10.000.

speedup superlinear em todas as execuções.

Visando avaliar o comportamento do *solver* para problemas maiores, resolveu-se, na sequência, sistemas de ordem 15.000 e 18.000. O primeiro possui número de condicionamento igual a $1,92.10^7$ e o tempo médio gasto na leitura dos arquivos com matrizes e vetores foi de $61,59$ segundos. Para o segundo, tem-se um condicionamento igual a $2,11.10^7$ e consumo de tempo para leitura dos arquivos com matrizes e vetores de $136,57$ segundos. Os tempos obtidos para resolução de tais sistemas são apresentados, respectivamente, nas tabelas 5.8 e 5.9 e os *speedups* descritos nas tabelas 5.10 e 5.11. As figuras 5.4 e 5.5 ilustram tais resultados.

Para o sistema de ordem 15.000 as curvas de *speedup* mantêm comportamento semelhantes às situações anteriores. Entretanto, considerando-se o sistema de ordem 18.000, tem-se algumas

Tabela 5.8: Tempos médios, em segundos, consumidos pelo *solver* otimizado para resolução de um sistema aleatório de ordem 15.000.

Algoritmo	*Cores*	Passo 1	Passo 2	Passo 3	Passo 4	Passo 5	P. 6 a 15	Total
S.Al.	1	1.905,496988	8,391690	23,631681	2.204,062042	0,000236	73,072866	4.488,746761
S.Al.	8	1.864,168661	7,517064	21,125423	421,749094	0,000228	34,416718	2.415,799849
Para.	1	1.147,871643	5,671851	19,137455	2.218,513062	0,000221	70,410126	3.461,604355
Para.	2	575,892914	5,702488	19,380062	1.169,313110	0,000205	64,806824	1.835,095602
Para.	3	387,977372	5,624678	18,184693	1.058,497315	0,000210	68,973812	1.539,258078
Para.	4	292,683976	5,692326	19,453861	646,021881	0,000226	32,453377	996,305645
Para.	5	249,250519	5,569979	18,195426	626,273224	0,000226	34,953677	934,243050
Para.	6	209,516846	5,740062	19,301625	493,246094	0,000220	33,680182	761,485027
Para.	7	182,304703	5,598719	17,885877	474,702958	0,000232	34,173462	714,665950
Para.	8	160,889222	5,686706	18,929379	451,520676	0,000219	32,993433	670,019633

Tabela 5.9: Tempos médios, em segundos, consumidos pelo *solver* otimizado para resolução de um sistema aleatório de ordem 18.000.

Algoritmo	*Cores*	Passo 1	Passo 2	Passo 3	Passo 4	Passo 5	P. 6 a 15	Total
S.Al.	1	3.374,025160	11,830346	34,319644	4.308,423647	0,016515	148,559452	7.877,174762
S.Al.	8	3.238,280031	9,172430	31,816384	1.294,515632	0,184354	157,921811	4.731,890641
Para.	1	1.997,913269	8,418813	42,727008	4.645,010743	0,000232	115,091442	6.809,161505
Para.	2	1.007,107361	8,435385	42,211746	2.793,399536	0,000236	221,293587	4.072,447849
Para.	3	691,537171	8,422550	37,828542	2.839,557495	0,000244	208,983643	3.786,329643
Para.	4	526,794861	8,367283	41,606862	1.869,193726	0,000240	175,131134	2.621,094106
Para.	5	470,019638	8,378324	34,885034	1.691,649475	0,000236	168,282517	2.373,215223
Para.	6	382,170685	8,393160	38,038727	1.762,313355	0,000234	218,002755	2.408,918915
Para.	7	344,572724	8,428928	37,691143	2.012,635315	0,000246	224,665573	2.627,993928
Para.	8	301,910462	8,396498	38,956726	1.586,507507	0,000236	169,679116	2.105,450544

Tabela 5.10: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema aleatório de ordem 15.000.

Cores	Sp T.T.Par.	Sp T.T.Seq.	Sp P1. Par.	Sp P1. Seq.	Sp P4. Par.	Sp P4. Seq.
2	1,89	2,45	1,99	3,35	1,90	1,88
3	2,25	2,92	2,96	4,88	2,10	2,08
4	3,47	4,51	3,92	6,40	3,43	3,41
5	3,71	4,80	4,61	7,18	3,54	3,52
6	4,55	5,89	5,48	8,83	4,50	4,47
7	4,84	6,28	6,30	9,79	4,67	4,64
8	5,17	6,70	7,13	11,18	4,91	4,88

diferenças, exceto pelas retas "Sp P1. Seq." e "Sp P1. Par". Os valores de "Sp P1. Seq." se mantiveram superlineares, conforme esperado. A coluna "Sp P1. Par" apresenta valores tangentes à reta linear, entretanto, para o sistema de ordem 18.000, o distanciamento entre essas passa a ocorrer com maior intensidade do que nas situações anteriores. Já as demais retas apresentaram um forte achatamento nesse caso. Acredita-se que tal comportamento deva-se ao fato de a quantidade requerida de dados ser superior ao limite suportado pela memória *cache*, pois, nos sistemas de maior ordem, os blocos necessários para o cálculo de uma incógnita são também maiores. Como não é possível controlar a ordem exata de operação dos *threads* nas fatias de dados, pode haver maior movimentação de dados através da hierarquia de memória devido ao escalonamento e, também, à

Tabela 5.11: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema aleatório de ordem 18.000.

Cores	Sp T.T.Par.	Sp T.T.Seq.	Sp P1. Par.	Sp P1. Seq.	Sp P4. Par.	Sp P4. Seq.
2	1,67	1,93	1,98	3,31	1,66	1,54
3	1,80	2,08	2,89	4,91	1,64	1,52
4	2,60	3,01	3,79	6,51	2,49	2,30
5	2,87	3,32	4,25	7,64	2,75	2,55
6	2,83	3,27	5,23	9,09	2,64	2,44
7	2,59	3,00	5,80	10,45	2,31	2,14
8	3,23	3,74	6,62	11,84	2,93	2,72

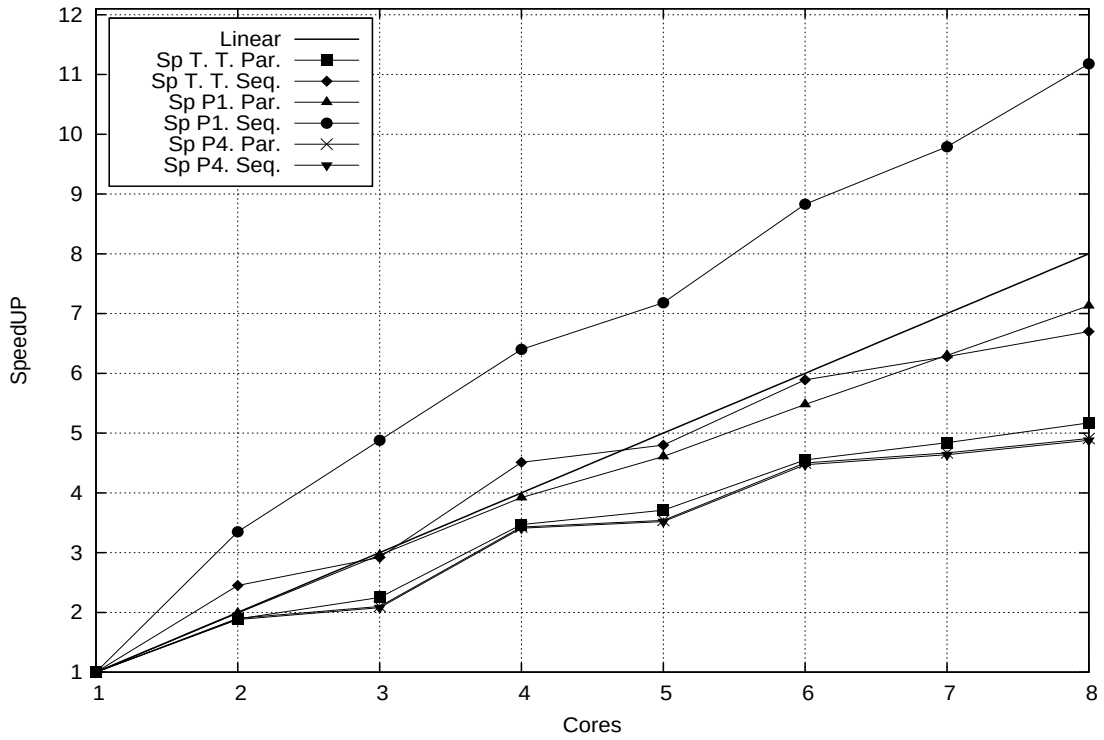

Figura 5.4: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema gerado aleatoriamente de ordem 15.000.

concorrência dos *threads* pelo *cache*, causando ociosidade nos *threads*.

Observa-se, também, que nos sistemas de maior ordem a abordagem alternativa de paralelização do Passo 4, ou seja, a coluna "Sp P4. Seq." obteve resultados ligeiramente melhores do que a abordagem principal ("Sp P4. Par."). É importante observar que os ganhos de desempenho da paralelização indicada por "Sp P4. Par." se devem principalmente ao reuso dos dados em *cache*, ou seja, no momento em que um *thread* carrega uma posição ou bloco da memória para o cálculo do limite superior (ou inferior), por ser o *cache* compartilhado, o mesmo pode ser também acessado pelo(s) *thread(s)* do limite inferior (ou superior) sem que haja necessidade de buscar novamente os dados na memória. Acredita-se, nesse caso, que o não sincronismo dos *threads* que operam sobre os mesmos dados, compartilhando *cache*, seja responsável pela diferença. Enquanto na abordagem

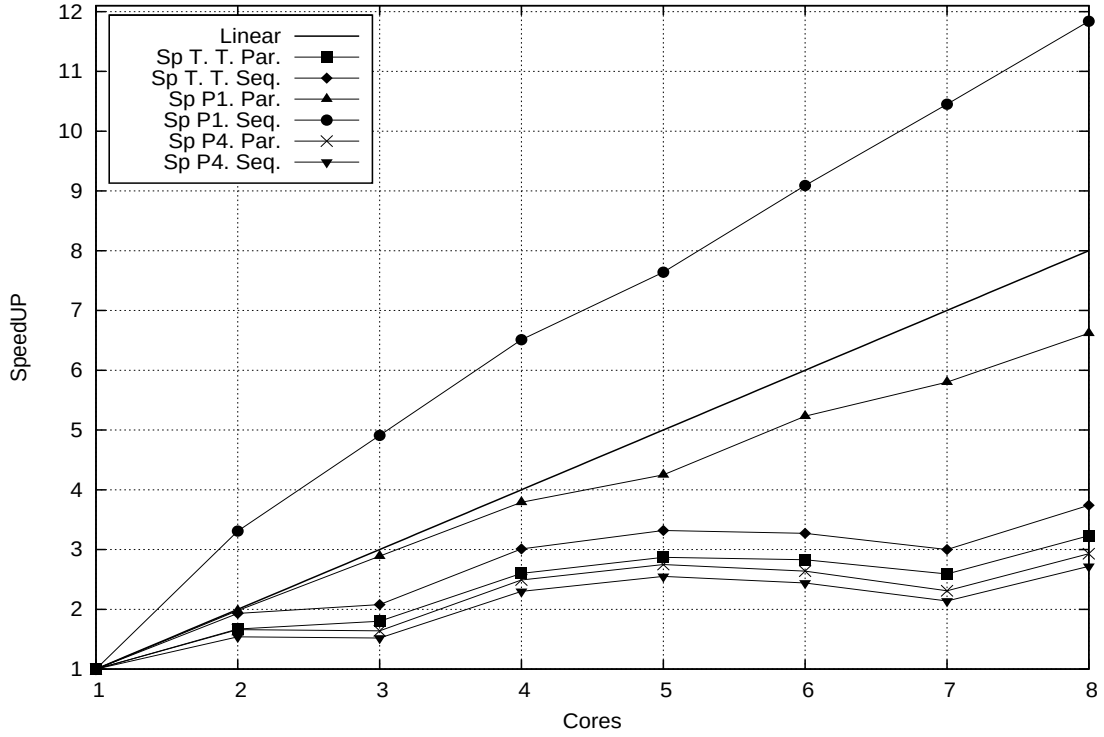

Figura 5.5: *Speedups* obtidos pelo *solver* otimizado na resolução de um sistema gerado aleatoriamente de ordem 18.000.

principal pode-se ter uma situação com $n/2$ *threads* operando no bloco $A(1,1)$ do limite superior e, em paralelo, $n/2$ *threads* operando o bloco $A(5,1)$ do limite inferior, na abordagem alternativa tem-se sempre os n *threads* operando em conjunto um mesmo bloco. Assim, em uma situação em que o *cache* é insuficiente, tem-se um resultado de tempo imprevisível, uma vez que haverá disputa pelo mesmo.

Cabe ainda observar que, embora o passo de verificação não tenha sido explicitamente paralelizado no desenvolvimento do *solver* otimizado, o fato de empregar as rotinas *multithread* da MKL fez com que o mesmo obtivesse ganhos de desempenho ao aumentar o número de *cores* mantendo, assim, constante a proporcionalidade do tempo consumido por tal passo. Ao resolver, com 1 *thread*, o sistema de ordem 6.000, o passo de verificação representava 2% do tempo total de execução do *solver* paralelo e 10% quando executando com 8 *threads*. No sistema de ordem 10.000, sua participação nas mesmas situações corresponde a, respectivamente, 1,75% e 9,5% do custo total. Já para o sistema com $n = 15.000$, tal passo responde por, aproximadamente, 2% em 1 núcleo e 4% em 8 núcleos. Por fim, no sistema de ordem 18.000, tais custos correspondem, respectivamente, a 1% e 8%.

Executou-se ainda o *solver* para sistemas de ordem 20.000. Entretanto, embora a resolução de SELAs dessa ordem seja suportada pela ferramenta no ambiente de testes disponível, não foi possível concluir tais avaliações. Isso porque, após 2 horas, em média, de execução do *solver*, o processo recebe sinal de *kill* do escalonador do sistema operacional. A Figura 5.6 ilustra o crescimento do tempo de execução do *solver* em relação ao crescimento da ordem do SELA resolvido.

Por fim, a comparação entre os tempos consumidos pelas rotinas do LAPACK em ambiente com processadores *multicore* confirma, em todas as situações, que de fato o pacote não oferece ganhos de desempenho em arquiteturas do tipo *multicore*. O passo de inversão da matriz, por exemplo, apresentou, para os sistemas de ordem 15.000 e 18.000, respectivamente, *speedups* de apenas $1,02$ e $1,04$ ao executar em 8 *cores*.

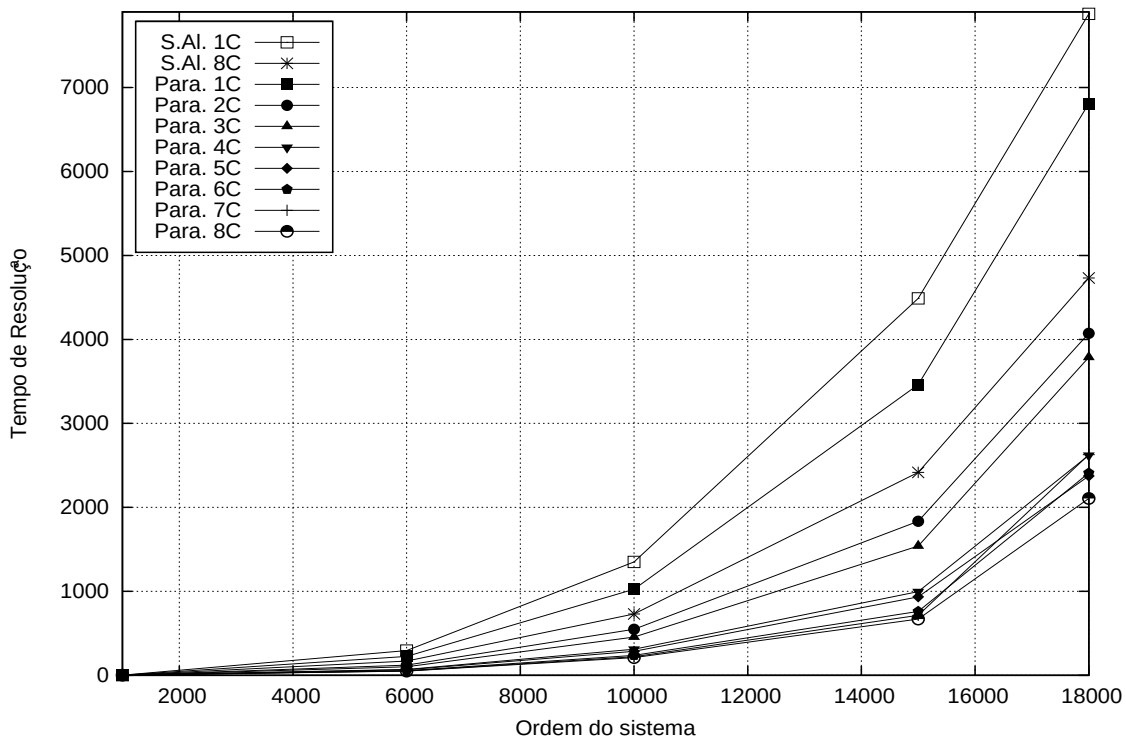

Figura 5.6: Crescimento do tempo médio de execução do *solver* em relação ao crescimento do tamanho do problema.

5.4 Considerações Finais

Neste capítulo, diversas avaliações do *solver* otimizado foram apresentadas. Em relação à exatidão dos resultados numéricos, avaliou-se tanto sistemas bem condicionados quanto mal condicionados, obteve-se resultados satisfatórios em ambas as situações. Tais resultados foram comparados à versão original da ferramenta e, dessa forma, comprovou-se que não houve perda de qualidade com a otimização da solução proposta.

Em relação aos testes de desempenho, sistemas de diferentes ordens de grandeza foram resolvidos. De modo geral, obteve-se como resultado o comportamento esperado, apenas para o sistema de ordem 18.000 os tempos de execução se apresentaram imprevisíveis. Em [NAT10] encontram-se matrizes correspondentes a problemas reais das mais diversas áreas. Cabe observar, de modo a contextualizar o porte dos sistemas resolvidos neste trabalho, a variedade das ordens das matrizes disponíveis. Na disciplina de Engenharia Química, por exemplo, diversas matrizes apresentam n menor do que 1.000. Já em problemas relacionados a energia elétrica, matrizes com ordem de

5.000 a 12.000 são disponibilizadas. No contexto da Engenharia Estrutural, observa-se matrizes de ordem 5.000, 12.000, dentre outras. O sistema de maior porte encontrado pertence à disciplina de Mecânica Estrutural, a ordem do mesmo é de 90.449.

Assim, pode-se considerar os resultados bastante satisfatórios, uma vez que foi possível resolver, com ganhos de desempenho, sistemas de ordens variadas que abrangem uma diversidade de problemas reais. Adicionalmente, os *speedups* apresentados são bastante significativos para a classe de arquitetura em questão. Confirmou-se, ainda, através das avaliações deste capítulo, que, de fato, o LAPACK não apresenta ganhos de desempenho significativos em arquiteturas *multicore*.

6. CONCLUSÃO

Este trabalho apresentou uma aplicação da Computação Verificada em conjunto com a Computação Paralela para resolução de Sistemas de Equações Lineares Intervalares. Dentro desse contexto, a principal contribuição do mesmo é a disponibilização de uma ferramenta para resolução de Sistemas Intervalares Densos de Grande Porte com verificação automática dos resultados otimizada para execução em arquiteturas dotadas de processadores *multicore*. Tal ferramenta é capaz de prover resultados auto-validados para Sistemas Lineares cujos dados de entrada podem ser tanto intervalos quanto números pontuais, permitindo, dessa forma, ao usuário resolver problemas utilizando dados incertos.

A exploração do paralelismo inerente ao algoritmo permite ao *solver* se beneficiar das características presentes nas arquiteturas paralelas obtendo, assim, ganhos de desempenho. Neste projeto, explorou-se os processadores *multicore* de modo a tornar o tempo de execução da solução desenvolvida mais satisfatório para o usuário. Adicionalmente, bibliotecas de *software* para Álgebra Linear, altamente otimizadas para as arquiteturas em questão, foram empregadas para otimizar a execução do *solver* desenvolvido. Para viabilizar essa estratégia, identificou-se como o algoritmo mais adequado da Computação Verificada para resolução de SELAs o Método Verificado Baseado no Método de *Newton*. Além de ser um algoritmo da Computação Verificada, o principal diferencial desse método é o fato de permitir sua implementação empregando a Aritmética de Ponto-Médio e Raio, o que possibilita o desenvolvimento baseado nas bibliotecas MKL e PLASMA.

De fato, as avaliações de desempenho demonstraram êxito no atendimento daquele que pode ser resumido como objetivo principal deste trabalho: otimização do *solver* para execução com ganhos de desempenho em arquiteturas *multicore*. Com efeito, observou-se, devido à utilização das rotinas da biblioteca PLASMA, que não apenas o paralelismo em nível de *threads* foi responsável pelo ganhos. A estratégia de execução do *software* de modo análogo a um *pipeline* superescalar, através de práticas como escalonamento e execução fora de ordem, antecipação de operações, não sincronismo e divisão das matrizes em submatrizes, trouxe também forte contribuição para o desempenho do *solver*.

Comparando-se, o passo de inversão da matriz A executado por chamadas às rotinas da PLASMA com o mesmo passo executado pelo LAPACK, ambos utilizando somente 1 núcleo, tem-se uma diferença em torno de aproximadamente $1,65$ vezes. Além disso, ao aumentar-se o número de núcleos, enquanto a solução que utiliza a PLASMA apresentou *speedup* crescente e bastante satisfatório, seu equivalente utilizando LAPACK apresentou variação mínima no tempo de execução e *speedup* médio abaixo de $1,05$ para 8 *cores*. Comparando-se as execuções *multithread* empregando a PLASMA com aquela utilizando LAPACK, obteve-se *speedups* bastante superiores em todas as situações.

As demais constatações acerca do desempenho apresentaram-se também satisfatórias. Em geral, obteve-se *speedups* de valor considerável tanto em relação aos passos paralelizados quanto aos tempos totais de execução, especialmente considerando a classe de arquitetura empregada. Realizou-se,

ainda, avaliações utilizando números de *threads* superiores ao número de processadores disponíveis. Tais testes apresentaram, em todos os casos, perda de desempenho. Cabe observar que, apesar de o computador utilizado durante as avaliações possuir o recurso de *Hyperthreading* habilitado, essa situação é esperada devido ao fato de o algoritmo estar paralelizado e de o *Hyperthreading* ser uma técnica que, em geral, traz ganhos para os trechos de *softwares* não paralelos.

A escalabilidade é, também, uma característica presente no *solver* desenvolvido. As avaliações do Capítulo 5 mostraram que, exceto para o sistema de ordem 18.000, ao comparar-se a resolução de dois sistemas em um mesmo número de *cores*, em todos os casos o *speedup* é maior para o sistema de maior ordem. Adicionalmente, considerando-se a resolução de um mesmo sistema e a variação apenas do número de *cores*, o crescimento do *speedup*, embora não-linear, é mantido em todas as situações.

Em relação a exatidão, obteve-se resultados bastante satisfatórios. Mesmo para o sistema fortemente mal condicionado (matriz de *Boothroyd/Dekker*), a ferramenta encontrou soluções verificadas. Já considerando os sistemas gerados aleatoriamente que, por sua vez, apresentam bom condicionamento, os resultados são ainda mais precisos. Em diversos casos o intervalo apresentou diâmetro igual a 0 para uma exatidão de sete casas decimais, em outros houve variação de apenas uma unidade na última casa decimal.

Portanto, de modo geral os resultados obtidos podem ser considerados bastante satisfatórios, uma vez que todos os objetivos da Computação Verificada foram atingidos e que as estratégias de desenvolvimento adotadas permitiram ao *solver* atender o requisito de desempenho definido. Além disso, conforme observado no Capítulo 5, a ordem dos sistemas resolvidos é suficiente para abranger uma série de problemas científicos reais. Cabe ainda observar que, embora o *solver* utilize um algoritmo destinado a resolução de Sistemas Densos, o mesmo é, também, capaz de resolver Sistemas Esparsos. Entretanto, por não ser otimizado para esse tipo de problema, tais sistemas são tratados pelo *solver* de maneira análoga aos SELAs Densos e, por isso, os potenciais benefícios de desempenho e armazenamento inerentes às estruturas esparsas são desprezados.

A habilidade de resolver Sistemas Intervalares Densos de Grande Porte com verificação automática dos resultados é de notória importância para diversas áreas do conhecimento, principalmente ao trabalhar com dados incertos. Nesse contexto, a possibilidade de executar tais computações com tempo reduzido em arquiteturas *multicore*, as quais possuem baixo custo e se tornam cada vez mais populares, oferece grandes facilidades a diversos pesquisadores para o correto entendimento dos problemas por eles modelados.

6.1 Trabalhos Futuros

Os resultados deste projeto encontram-se parcialmente publicados em [MIL10]. Durante a realização do mesmo, identificou-se alguns pontos de interesse para a continuação e aperfeiçoamento da pesquisa desenvolvida. Tratam-se de aspectos para melhoria da solução aqui apresentada tanto considerando o contexto desenvolvido quanto as possibilidades para ampliação do escopo. São eles:

- Otimização dos passos do Algoritmo 4.1 não paralelizados no presente trabalho;
- Aperfeiçoamento da paralelização de modo a aumentar a escalabilidade do *solver*;
- Desenvolvimento de uma estratégia que permita detectar, em tempo de execução, a abordagem de paralelização mais adequada em relação ao Passo 4 do Algoritmo 4.1 para resolução de um dado sistema;
- Implementação de algoritmos alternativos para verificação dos resultados, os quais devem permitir obter benefícios das matrizes com estruturas especiais como, por exemplo, Sistemas Esparsos;
- Integração da solução desenvolvida com soluções para ambientes paralelos baseados em troca de mensagem visando exploração do *solver* em agregados de computadores cujos nós são constituídos por computadores dotados de processadores *multicore*;
- Desenvolvimento de estratégias que permitam obter ganhos de desempenho em computadores heterogêneos/híbridos, ou seja, naqueles em que é possível explorar o paralelismo não apenas em CPUs com múltiplos *cores* mas também em GPUs. Uma possibilidade nesse sentido é a utilização de resultados do projeto MAGMA (*Matrix Algebra on GPU and Multicore Architectures*) [TOM10] bem como uma possível integração quando houverem bibliotecas disponibilizadas pelo mesmo.

Bibliografia

[ALE98] G. Alefeld e D. M. Claudio. "The Basic Properties of Interval Arithmetic, its Software Realizations and Some Applications", *Computers and Structures*, Elsevier, vol. 67–(1–3), Abril 1998, pp. 3–8.

[ALE00] G. Alefeld e G. Mayer. "Interval Analysis: Theory and Applications", *Journal of Computational and Applied Mathematics*, Elsevier, vol. 121–(1–2), Setembro 2000, pp. 421–464.

[AND90] E. Anderson, Z. Bai, J. J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. W. Demmel, C. Bischof e D. Sorensen. "LAPACK: A Portable Linear Algebra Library for High-Performance Computers". In: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, 1990, pp. 2–11.

[AND99] E. Anderson, Z. Baia, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney e D. Sorensen. "LAPACK Users' Guide". Philadelphia:Society for Industrial and Applied Mathematics (SIAM), 1999, 3^a Edição, 407p.

[ANS85] ANSI/IEEE. "A Standard for Binary Floating-Point Arithmetic, STD.754-1985". Technical Report, USA, 1985, 20p.

[BLA02] L. S. Blackford, J. W. Demmel, J. J. Dongarra, I. S. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. P. Petitet, R. Pozo, K. Remington e R. C. Whaley. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)", *ACM Transactions on Mathematical Software*, ACM, vol. 28–2, Junho 2002, pp. 135–151.

[BUT07] A. Buttari, J. J. Dongarra, J. Kurzak, J. Langou, P. Luszczek e S. Tomov. "The impact of Multicore on Math Software", *Lecture Notes in Computer Science - Applied Parallel Computing*, Springer-Verlag, vol. 4699, Setembro 2007, pp. 1–10.

[BUT08] A. Buttari, J. Langou, J. Kurzak e J. J. Dongarra. "Parallel Tiled QR Factorization for Multicore Architectures", *Lecture Notes in Computer Science - Parallel Processing and Applied Mathematics*, Springer-Verlag, vol. 4967, Maio 2008, pp. 639–648.

[BUT09] A. Buttari, J. Langou, J. Kurzak e J. J. Dongarra. "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures", *Parallel Computing*, Elsevier, vol. 35–1, Janeiro 2009, pp. 38–53.

[CHA07a] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti e R. van de Geijn. "Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures". In: Proceedings of the 9^{th} Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM, 2007, pp. 116–125.

[CHA07b] E. Chan, F. G. Van Zee, E. S. Quintana-Orti, G. Quintana-Orti e R. van de Geijn. "Satisfying your Dependencies with Supermatrix". In: Proceedings of the 2^{nd} IEEE International Conference on Cluster Computing (CLUSTER), IEEE Computer Society Press, 2007, pp. 91–99.

[CHA08] E. Chan, F. G. V. Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti e R. van de Geijn. "Supermatrix: a Multithreaded Runtime Scheduling System for Algorithms-by-Blocks". In: Proceedings of the 13^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, 2008, pp. 123–132.

[CHO94] J. Choi, J. J. Dongarra e D. W. Walker. "PB-BLAS: a Set of Parallel Block Basic Linear Algebra Subprograms". In: Proceedings of the 2^{nd} Scalable High-Performance Computing Conference (SHPCC), IEEE Computer Society Press, 1994, pp. 534–541.

[CHO95a] J. Choi e J. J. Dongarra. "Scalable Linear Algebra Software Libraries for Distributed Memory Concurrent Computers". In: Proceedings of the 5^{th} IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems (FTDCS), IEEE Computer Society Press, 1995, pp. 170–177.

[CHO95b] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet e D. M. H. Walker. "LAPACK Working Note 100: A Proposal for a Set of Parallel Basic Linear Algebra Subprograms". Technical Report, Knoxville, TN, USA, 1995, 39p.

[CHO96] J. Choi, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, K. Stanley, D. W. Walker e R. C. Whaley. "ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance", *Computer Physics Communications*, IEEE Computer Society Press, vol. 97–(1–2), Agosto 1996, pp. 1–15.

[CLA00] D. M. Claudio e J. M. Marins. "Cálculo Numérico Computacional". São Paulo:Atlas, 2000, 2^a Edição, 464p.

[DEM89] J. W. Demmel. "LAPACK: a Portable Linear Algebra Library for Supercomputers". In: Proceedings of the 6^{th} IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD), IEEE Computer Society Press, 1989, pp. 1–7.

[DON79] J. J. Dongarra, G. B. Moler, J. R. Bunch e G. W. Stewart. "LINPACK Users' Guide". Philadelphia:Society for Industrial and Applied Mathematics (SIAM), 1979, 320p.

[DON88a] J. J. Dongarra, J. Du Croz, S. Hammarling e R. J. Hanson. "Algorithm 656: an Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs", *ACM Transactions on Mathematical Software*, ACM, vol. 14–1, Março 1988, pp. 18–32.

[DON88b] J. J. Dongarra, J. Du Croz, S. Hammarling e R. J. Hanson. "An Extended Set of Fortran Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, ACM, vol. 14–1, Março 1988, pp. 1–17.

[DON90] J. J. Dongarra, J. Du Croz, S. Hammarling e I. S. Duff. "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, ACM, vol. 16–1, Março 1990, pp. 1–17.

[DON97] J. J. Dongarra e R. C. Whaley. "LAPACK Working Note 94: A User's Guide to the BLACS v1.1". Technical Report, Knoxville, TN, USA, 1997, 66p.

[DON00] J. J. Dongarra e V. Eijkhout. "Numerical Linear Algebra Algorithms and Software", *Journal of Computational and Applied Mathematics*, Elsevier, vol. 123–(1–2), Novembro 2000, pp. 489–514.

[DON03] J. J. Dongarra, P. Luszczek e A. Petitet. "The LINPACK Benchmark: Past, Present and Future", *Concurrency and Computation: Practice and Experience*, John Wiley & Sons, vol. 15–9, Julho 2003, pp. 803–820.

[GEI94] Al Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek e V. Sunderam. "PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing". MIT Press, 1994, 299p.

[GOT02] K. Goto e R. A. van de Geijn. "On reducing TLB Misses in Matrix Multiplication - FLAME Working Note #9". Technical Report, Austin, Texas, USA, 2002, 19p.

[GOT08] K. Goto e R. A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication", *ACM Transactions on Mathematical Software*, ACM, vol. 34–3, Maio 2008, pp. 1–25.

[GUN01] J. A. Gunnels, F. G. Gustavson, G. M. Henry e R. A. van de Geijn. "FLAME: Formal Linear Algebra Methods Environment", *ACM Transactions on Mathematical Software*, ACM, vol. 27–4, Dezembro 2001, pp. 422–455.

[HAM97] R. Hammer, D. Ratz, U. W. Kulisch e M. Hocks. "C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems". Secaucus:Springer-Verlag, 1997, 400p.

[HAY03] B. Hayes. "A Lucid Interval", *American Scientist*, The Scientific Research Society, vol. 91–6, Outubro 2003, pp. 484–488.

[INT10] Intel. "Intel Xeon Processor E5520". Capturado em: http://ark.intel.com/Product.aspx?id=40200, Janeiro 2010.

[KEA96] R. B. Kearfott. "Interval Computations: Introduction, Uses, and Resources", *Euromath Bulletin*, European Mathematical Trust, vol. 2–1, Junho 1996, pp. 95–112.

[KLA93] R. Klatte, U. W. Kulisch, C. Lawo, R. Rauch e A. Wiethoff. "C-XSC - A C++ Class Library for Extended Scientific Computing". Berlin:Springer-Verlag, 1993, 269p.

[KNU94] O. Knuppel. "PROFIL BIAS - A Fast Interval Library", *Computing*, vol. 53–(3–4), Springer Wien, Setembro 1994, pp. 277–287.

[KOL06] M. L. Kolberg, L. Baldo, P. Velho, T. Webber, L. G. Fernandes, P. Fernandes e D. M. Claudio. "Parallel Selfverified Method for Solving Linear Systems". In: Proceedings of the 7^{th} International Meeting of High Performance Computing for Computational Science (VECPAR), Springer-Verlag , 2006, pp. 179–190.

[KOL07] M. L. Kolberg, L. Baldo, P. Velho, L. G. Fernandes e D. M. Claudio. "Optimizing a Parallel Self-Verified Method for Solving Linear Systems", *Springer Lecture Notes in Computer Science - Applied Parallel Computing*, Springer-Verlag, vol. 4699, Setembro 2007, pp. 949–955.

[KOL08a] M. L. Kolberg, G. Bohlender e D. M. Claudio. "Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computation", *Lecture Notes in Computer Science - Applied Parallel Computing: State of the Art in Scientific Computing - VECPAR 2008 Revised Selected Papers*, Springer-Verlag, vol. 5336, Junho 2008, pp. 13–26.

[KOL08b] M. L. Kolberg, L. G. Fernandes e D. M. Claudio. "Dense Linear System: A Parallel Self-Verified Solver", *International Journal of Parallel Programming*, Springer Netherlands, vol. 36–4, Agosto 2008, pp. 412–425.

[KOL08c] M. L. Kolberg, M. Dorn, G. Bohlender e L. G. Fernandes. "Parallel Verified Linear System Solver for Uncertain Input Data". In: 20^{th} International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE Computer Society Press, 2008, pp. 89–96.

[KRA09] W. Krämer e M. Zimmer. "Fast (Parallel) Dense Linear Interval Systems Solvers in C-XSC Using Error Free Transformations and BLAS". In: Numerical Validation in Current Hardware Architectures: International Dagstuhl Seminar, Springer-Verlag, 2009, pp. 230–249.

[KUM02] V. Kumar. "Introduction to Parallel Computing". Boston:Addison-Wesley Longman Publishing Co., 2002, 856p.

[KUR07] J. Kurzak e J. J. Dongarra. "Implementing Linear Algebra Routines on Multi-Core Processors With Pipelining and a Look Ahead", *Springer Lecture Notes in Computer Science - Applied Parallel Computing*, Springer-Verlag, vol. 4699, Setembro 2007, pp. 147–156.

[LAW79a] C. L. Lawson, R. J. Hanson, D. R. Kincaid e F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage", *ACM Transactions on Mathematical Software*, ACM, vol. 5–3, Setembro 1979, pp. 308–323.

[LAW79b] C. L. Lawson, R. J. Hanson, F. T. Krogh e D. R. Kincaid. "Algorithm 539: Basic Linear Algebra Subprograms for Fortran Usage [F1]", *ACM Transactions on Mathematical Software*, ACM, vol. 5–3, Setembro 1979, pp. 324–325.

[LOW04] T. M. Low e R. A. van de Geijn. "An API for Manipulating Matrices Stored by Blocks - FLAME Working Note #12". Technical report, USA, 2004, 16p.

[MIL10] C. R. Milani, M. L. Kolberg e L. G. Fernandes. "Solving Dense Interval Linear Systems With Verified Computing on Multicore Architectures". In: Aceito para o 9^{th} International Meeting of High Performance Computing for Computational Science (VECPAR), 2010, 14p.

[NAT10] National Institute of Standards and Technology. "Matrix market". Capturado em: http://math.nist.gov/MatrixMarket, Janeiro 2010.

[QUI00] E. S. Quintana, G. Quintana, X. Sun e R. van de Geijn. "A Note on Parallel Matrix Inversion", *SIAM Journal on Scientific Computing*, Society for Industrial and Applied Mathematics (SIAM), vol. 22–5, Maio 2000, pp. 1762–1771.

[RUM83] S. M. Rump. "Solving Algebraic Problems With High Accuracy". In: Symposium on A New Approach to Scientific Computation, Academic Press Professional, 1983, pp. 51–120.

[RUM99] S. M. Rump. "Fast and Parallel Interval Arithmetic", *BIT Numerical Mathematics*, Springer Netherlands, vol. 39–3, Setembro 1999, pp. 534–554.

[RUM01] S. M. Rump. "Self-Validating Methods", *Linear Algebra and Its Applications*, Elsevier, vol. 324–(1–3), Fevereiro 2001, pp. 3–13.

[SKI98] D. B. Skillicorn e D. Talia. "Models and Languages for Parallel Computation", *ACM Computing Surveys*, ACM, vol. 30–2, Junho 1998, pp. 123–169.

[SMI76] B. T. Smith, J. M. Boyle e J. J. Dongarra. "Matrix Eigensystem Routines - EISPACK Guide", *Springer Lecture Notes in Computer Science*, Springer-Verlag, vol. 34, 1976, 6p.

[SNI96] M. Snir, S. Otto, S. Huss-Lederman, D. W. Walker e J. J. Dongarra. "MPI: The Complete Reference". MIT Press, 1996, 350p.

[SON09] F. Song, A. YarKhan e J. J. Dongarra. "Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems". Technical report, USA, 2009, 10p.

[TOM10] S. Tomov, R. Nath, H. Ltaief e J. J. Dongarra. "Dense Linear Algebra Solvers for Multicore With GPU Accelerators". In: Aceito para o 15^{th} International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), 2010, 8p.

[TOP10] TOP500.Org. "Top500 Supercomputing Sites". Capturado em: http://www.top500.org/, Janeiro 2010.

[WHA98] R. C. Whaley e J. J. Dongarra. "Automatically Tuned Linear Algebra Software". In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM) (SUPERCOMPUTING), ACM, 1998, pp. 1–27.

[WHA04] R. C. Whaley e A. P. Petitet. "Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS". *Software: Practice and Experience*, John Wiley & Sons, vol. 35–2, Novembro 2004, pp. 101–121.