Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-graduação em Ciência da Computação

Parallel Self-Verified Solver for Dense Linear Systems

Mariana Luderitz Kolberg

Tese de Doutorado apresentada à Faculdade de Informática como parte dos requisitos para obtenção do título de Doutor em Ciência da Computação. Área de concentração: Ciência da Computação.

Orientador: Dalcidio Moraes Claudio
Co-Orientador: Luiz Gustavo Leão Fernandes

Porto Alegre, RS
2009

Dados Internacionais de Catalogação na Publicação (CIP)

K81p Kolberg, Mariana Luderitz
Parallel self-verified solver for dense linear systems / Mariana Luderitz Kolberg. – Porto Alegre, 2009.
124 f.

Tese (Doutorado em Ciência da Computação) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Dalcidio Moraes Claudio.

1. Informática. 2. Processamento Paralelo. 3. Aritmética Computacional. 4. Sistemas Lineares. I. Título. II. Claudio, Dalcidio Moraes.

CDD 004.0151

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

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "***Parallel Self-Verified Solver for Dense Linear Systems***", apresentada por Mariana Luderitz kolberg, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, Computação Científica, aprovada em 09/01/09 pela Comissão Examinadora:

[illegible]

Prof. Dr. Dalcidio Moraes Claudio – Orientador — PPGCC/PUCRS

[illegible]

Prof. Dr. Luiz Gustavo Leão Fernandes - Co-orientador — PPGCC/PUCRS

[illegible]

Prof. Dr. Paulo Henrique Lemelle Fernandes - — PPGCC/PUCRS

Prof. Dr. Rudnei Dias da Cunha - [illegible] — UFRGS

[illegible]

Prof. Dr. Alfredo Goldman vel Lejbman– — USP

Homologada em 16/06/09, conforme Ata No. 010 pela Comissão Coordenadora.

[illegible]

Prof. Dr. Fernando Gehm Moraes
Coordenador.

To my family and friends, who offered me unconditional love and support throughout the course of this thesis.

Acknowledgments

To my family, for the support and encouragement at all times. I would like to thank especially my parents and my brother who always believed in me and encouraged me to pursue my goals. I also thank my grandfather Oscar, who gave me all necessary support and care throughout my life, always believing that I would reach my goals.

To all my friends and colleagues in both Porto Alegre and Karlsruhe, I would like to thank for all support and understanding during these 4 years.

To Professor Dalcidio Claudio, for the guidance since 1998. For over 10 years, Professor Dalcidio has been my mentor, always supporting and encouraging me to find new challenges. I also thank the knowledge he passed me over the years.

To Professor Luiz Gustavo Fernandes, for his advise and support during my PhD, especially during the last months. I also would like to thank for introducing me to the parallel computing field with a joint paper during the first year of my PhD, which led to the definition of the subject of this thesis.

To the Professors Gerd Bohlender and Rudi Klatte, from University of Karlsruhe, for the necessary advise and support in the elaboration of this research during my PhD, especially during my stay in Karlsruhe. In this period more than mentors, they were wonderful people, whose concerns about me went beyond the work and helped me a lot to adapt to Germany and made me feel part of their families.

To Professor Alfredo Goldman, for his suggestions, which led to the development of new and important ideas. To Professor Paulo Fernandes, who has been following my work since the graduation dissertation (as advisor) and during all my master and PhD course (as evaluator), for pointing out improvements and suggestions that have helped me a lot. To Professor Rudnei Cunha, for his participation as evaluator, with interesting questions and suggestions that led me to important conclusions, thanks.

To the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), for the financial support during my studies in Porto Alegre and Karlsruhe, including the scholarship, university fees, plane tickets, and all financial support necessary for the completion of this PhD.

To Baden-Württembergische Brasilien-Zentrum der Universität Tübingen, for the financial support during my second year at the University of Karlsruhe. This support was essential for a better development of my PhD thesis.

Resumo

Esta tese apresenta uma ferramenta de resolução de sistemas lineares densos pontuais e intervalares. As principais características desta ferramenta são rapidez, confiabilidade e precisão. Esta ferramenta é baseada em um método de resolução de sistemas densos verificado usando arredondamentos direcionados e aritmética intervalar associados a bibliotecas otimizadas e primitivas MPI para prover resultados confiáveis e alto desempenho.

A primeira versão paralela foi desenvolvida usando a biblioteca C-XSC. Esta versão não alcançou o desempenho global esperado uma vez que não foi paralelizada totalmente devido a particularidades do C-XSC (variáveis especiais e produto escalar ótimo). Como o C-XSC não se mostrou eficiente para aplicações de grande porte, foi proposta e implementada uma nova versão seqüencial para sistemas lineares densos usando tanto a aritmética de ínfimo e supremo como a aritmética de ponto médio e raio, baseada nas bibliotecas BLAS e LAPACK. Testes de desempenho mostraram que o algoritmo que implementa a aritmética de ponto médio e raio possui um desempenho melhor do que o algoritmo que implementa a aritmética de ínfimo e supremo. Considerando este resultado, a aritmética de ponto médio e raio foi escolhida para a próxima etapa: a implementação paralela.

Uma versão paralela para solução de sistemas lineares pontuais e intervalares densos foi então desenvolvida utilizando a aritmética de ponto médio e raio, arredondamentos direcionados e as bibliotecas otimizadas PBLAS e ScaLAPACK. Os resultados mostraram que foi possível alcançar um bom desempenho utilizando um número de processadores variado e proporcionando considerável aceleração na obtenção dos resultados para diferentes tamanhos de matrizes (pontuais e intervalares).

A fim de superar as limitações impostas pelo uso da memória na geração de toda a matriz em um só processador, uma nova versão foi implementada. Esta versão gera as sub-matrizes da matriz principal em cada processador, permitindo uma melhor utilização da memória global disponibilizada pelo Cluster. Estas alterações tornaram possível resolver sistemas densos de dimensão 100 000.

Para investigar a portabilidade da solução proposta, os testes foram realizados em 3 Clusters diferentes na Alemanha (ALiCEnext, XC1 e IC1). Cada um destes Clusters possui configurações distintas e apresentaram resultados significativos, indicando que a versão paralela possui uma boa escalabilidade para sistemas lineares muito grandes usando um número variado de processadores.

Outros estudos foram realizados em duas direções. O primeiro diz respeito ao uso de threads dedicadas para aumentar o desempenho da solução de sistemas lineares usando memória compartilhada (em especial para processadores dual-core). Também foi estudada a utilização dessas idéias para aumentar o desempenho da solução usando C-XSC.

Abstract

This thesis presents a free, fast, reliable and accurate solver for point and interval dense linear systems. The idea was to implement a solver for dense linear systems using a verified method, interval arithmetic and directed roundings based on MPI communication primitives associated to optimized libraries, aiming to provide both self-verification and speed-up at the same time.

A first parallel implementation was developed using the C-XSC library. However, the C-XSC parallel method did not achieve the expected overall performance since the solver was not 100% parallelized due to its implementation properties (special variables and optimal scalar product).

C-XSC did not seem to be the most efficient tool for time critical applications, consequently we proposed and implemented a new sequential verified solver for dense linear systems for point and interval input data using both infimum-supremum and midpoint-radius arithmetic based on highly optimized libraries (BLAS/ LAPACK). Performance tests showed that the midpoint-radius algorithm needs approximately the same time to solve a linear system with point or interval input data, while the infimum-supremum algorithm needs much more time for interval data. Considering that, midpoint-radius arithmetic was the natural choice for the next step of this work: the parallel implementation.

We then developed a new parallel verified solver for point and interval dense linear systems using midpoint-radius arithmetic, directed roundings and optimized libraries (PBLAS/ ScaLAPACK). The performance results showed that it was possible to achieve very good speed-ups in a wide range of processor numbers for large matrix dimensions for both point and interval input data.

In order to overcome the memory limitation imposed by the generation of the whole matrix in one processor, we decided to generate sub-matrices of the input matrix individually on each available node, allowing a better use of the global memory. These modifications made it possible to solve dense systems with up to $100\,000$ dimension.

In addition to that, in order to investigate the portability of the proposed solution, during this thesis, tests were performed using 3 different clusters in Germany (ALiCEnext, XC1 and IC1) with distinct configurations presenting significant results, indicating that the parallel solver scales well even for very large dense systems over many processors.

Further investigations were done in two directions: study of the use of dedicated threads to speed up the solver of dense linear systems on shared memory, specially dual-core processors and the use of the ideas presented in this thesis to speed-up the C-XSC library.

List of Figures

Figure 1 Fixed-Point Accumulator. 46
Figure 2 Communication Schemes. 55
Figure 3 Experimental Results - Speed-up. 58
Figure 4 Experimental Results - Speed-up for Large Matrices. 58
Figure 5 Experimental Results - Execution Time. 59
Figure 6 ScaLAPACK Software Hierarchy. 77
Figure 7 Experimental Results - Speed-up. 84
Figure 8 Experimental Results - Efficiency. 85
Figure 9 Structure View of Matrix QH1484. 85
Figure 10 Speed-up for Interval Input Data. 90
Figure 11 Graphic Representation of the Mapping Function. 95
Figure 12 Execution Time for Solving Matrix 1 on XC1. 98
Figure 13 Execution Time for Solving Matrix 1 on IC1. 99
Figure 14 Relative Speed-up Using 64 and 128 Processors on XC1. 99
Figure 15 Relative Speed-up Using 128 and 256 Processors on IC1. 100
Figure 16 Time for the Computation of C (Multithreaded x Sequential). 105
Figure 17 Total Execution Time for the Multithreaded Version. 106

List of Tables

Table 1	IEEE Data Types.	22
Table 2	Solution of Equation (6.1) by the Numerical Tools.	51
Table 3	Performance of Numerical Tools.	52
Table 4	Comparison among C-XSC, INTLAB and LAPACK.	52
Table 5	Experimental Results - Efficiency.	59
Table 6	Results on XC1.	67
Table 7	Accuracy on XC1 Compared with C-XSC.	67
Table 8	Performance Comparison (in Seconds) on XC1.	68
Table 9	Execution Time (in Seconds) for Point Input Data on XC1.	68
Table 10	Execution Time for Point Input Data with Dimension 8 001.	69
Table 11	Results on XC1.	72
Table 12	Execution Time (in Seconds) for Interval Input Data on XC1.	73
Table 13	Execution Time for Interval Input Data with Dimension $8\,001$.	74
Table 14	Sequential Approach Used in the New Solver.	74
Table 15	Midpoint-Radius and the Equivalent Infimum-Supremum Result.	82
Table 16	Relation Between Condition Number and Diameter.	83
Table 17	Results for the Boothroyd/Dekker 10x10 Matrix.	83
Table 18	Results for QH1484: Quebec Hydroelectric Power System.	85
Table 19	Real Problem Execution Time and Speed-up.	86
Table 20	Sequential and Parallel Midpoint-Radius Result.	90
Table 21	Summary of the Test Cases: Execution Times and Efficiency.	91
Table 22	Speed-up for Point and Interval Input Data for Dimension $5\,000$.	92
Table 23	Comparison of Clusters - Hardware.	94
Table 24	Comparison of Clusters - Software.	94
Table 25	Result on Both Clusters.	97
Table 26	Memory per Processor.	98
Table 27	Execution time needed for each step of the algorithm.	100
Table 28	Results for the Boothroyd/Dekker 10×10 Matrix.	104
Table 29	Results for MHD3200B: Alfven Spectra in Magnetohydrodynamics.	107
Table 30	Cluster Comparison.	110
Table 31	Execution Time.	110

List of Algorithms

1 Infimum-Supremum Addition in $\mathbb{I}^*\mathbb{F}$. 26
2 Infimum-Supremum Subtraction in $\mathbb{I}^*\mathbb{F}$. 26
3 Infimum-Supremum Multiplication in $\mathbb{I}^*\mathbb{F}$. 27
4 Infimum-Supremum Inversion in $\mathbb{I}^*\mathbb{F}$. 27
5 Optimized Infimum-Supremum Addition in $\mathbb{I}^*\mathbb{F}$. 28
6 Optimized Infimum-Supremum Subtraction in $\mathbb{I}^*\mathbb{F}$. 28
7 Optimized Infimum-Supremum Inversion in $\mathbb{I}^*\mathbb{F}$. 28
8 Optimized Infimum-Supremum Multiplication in $\mathbb{I}^*\mathbb{F}$. 29
9 Midpoint-Radius Addition and Subtraction in $\mathbb{F}$. 31
10 Midpoint-Radius Multiplication in $\mathbb{F}$. 31
11 Midpoint-Radius Inversion in $\mathbb{F}$. 31
12 Rounding Alternative to Midpoint-Radius Inversion in $\mathbb{F}$. 31
13 Midpoint-Radius Scalar Product in $\mathbb{F}^n$. 32
14 Midpoint-Radius Matrix Multiplication in $\mathbb{F}^n$. 32
15 Enclosure of a Square Linear System Using C-XSC. 47
16 Enclosure of a Square Linear System Using INTLAB. 48
17 Workload Distribution Algorithm. 56
18 Enclosure of a Point Linear System Using LAPACK, BLAS. 63
19 Calculation of z. 64
20 Enclosure of an Interval Linear System Using LAPACK, BLAS. 70
21 Mapping Global Position of A to Local Positions in MyA. 95

List of Abbreviations

ALiCEnext	-	Advanced Linux Cluster Engine Next Generation
AMD	-	Advanced Micro Devices
ANSI	-	American National Standards Institute
ATM	-	Asynchronous Transfer Mode
BLACS	-	Basic Linear Algebra Communication Subprograms
BLAS	-	Basic Linear Algebra Subprograms
COW	-	Cluster of Workstations
CPU	-	Central Processing Unit
C-XCS	-	C for extended Scientific Computing
DSM	-	Distributed Shared Memory
FMA	-	Fused Multiply-Add
FORTRAN	-	Formula Translation
GB	-	Giga Byte
IEEE	-	Institute of Electrical and Electronics Engineers
INTLAB	-	Interval Laboratory
LAPACK	-	Linear Algebra Package
MATLAB	-	Matrix Laboratory
MB	-	Mega Byte
MHD	-	Magnetohydrodynamics
MKL	-	Math Kernel Library
MPI	-	Message Passing Interface
MPMD	-	Multiple Program Multiple Data
MPP	-	Massively Parallel Programming
NORMA	-	Non-Remote Memory Access
NOW	-	Network of Workstations
NUMA	-	Non-Uniform Memory Access
OPENMP	-	Open Multi Processing
PBLAS	-	Parallel Basic Linear Algebra Subprograms
PC	-	Personal Computer
PVM	-	Parallel Virtual Machine

RAM	-	Random Access Memory
SCALAPACK	-	Scalable Linear Algebra Package
SIAM	-	Society for Industrial and Applied Mathematics
SMP	-	Symetric Multiprocessor
SPMD	-	Single Program Multiple Data
TFLOPS	-	Tera Floating-point Operations Per Second
UMA	-	Uniform Memory Access

Contents

1 Introduction **16**

2 Background **20**

2.1 Computer Arithmetic 20

2.1.1 The IEEE Standard 21

2.2 Uncertain Data 23

2.3 Interval Arithmetic 24

2.3.1 Infimum-Supremum Interval Arithmetic 25

2.3.2 Midpoint-Radius Interval Arithmetic 30

2.3.3 Comparison 32

2.4 Verified Computing 34

2.5 Linear Systems 35

2.5.1 Direct Methods 36

2.5.2 Iterative Methods 36

2.5.3 Condition Number 37

2.5.4 Verified Method 38

2.6 Parallel Computing 40

2.6.1 Parallel Architectures 40

2.6.2 Parallel Programming 41

2.6.3 Performance Analysis 42

3 Tools and Solvers for Dense Linear Systems **45**

3.1 C-XSC 45

3.1.1 Solver 46

3.2 MATLAB and INTLAB 47

3.2.1 Verifylss.m 47

3.3 LAPACK 49

3.3.1 Solver 49

3.4 Comparison 49

3.4.1 Availability 50

3.4.2 Accuracy 50

3.4.3 Performance 52
3.5 Conclusions 52

4 First Parallel Approach using C-XSC 53
4.1 Implementation Issues 53
4.1.1 The Inverse Matrix Computation 54
4.1.2 Parallel Preconditioning 55
4.1.3 Load Balancing Approach 56
4.2 Experimental Results 56
4.2.1 Accuracy 57
4.2.2 Performance 57
4.3 Conclusions 60

5 New Sequential Solver 61
5.1 Directed Roundings 61
5.2 Point Input Data 62
5.2.1 Infimum-Supremum Approach 64
5.2.2 Midpoint-Radius Approach 65
5.2.3 Experimental Results for Point Input Data 65
5.3 Interval Data 69
5.3.1 Infimum-Supremum Approach 71
5.3.2 Midpoint-Radius Approach 71
5.3.3 Tests and Results for Interval Input Data 72
5.4 Conclusions 74

6 Highly Optimized Distributed Memory Solver 76
6.1 Parallel Optimized Libraries 76
6.1.1 Data Distribution 78
6.2 Platform 80
6.3 Approach for Point Input Data 80
6.3.1 Experimental Results 82
6.4 Approach for Interval Input Data 86
6.4.1 Implementation Issues 87
6.4.2 Experimental Results 89
6.5 Conclusions 92

7 Solving Very Large Matrices in Clusters 93
7.1 Platform 93
7.2 Approach for Very Large Systems 94
7.3 Experimental Results 95

7.3.1 Input Data 96
7.3.2 Accuracy 96
7.3.3 Performance 97
7.4 Conclusions 101

8 Shared Memory Approach for Dual-Cores 102
8.1 Implementation Issues 102
8.2 Experimental Results 103
8.2.1 Accuracy 104
8.2.2 Performance 104
8.2.3 Real Problem 106
8.3 Conclusions 107

9 SIAM 100-Digits Challenge 108
9.1 The Problem 7 of the Siam 100-Digit Challenge 108
9.2 Implementation 109
9.3 Interval Results and Some Timing Information 109
9.4 Solution with 100 Digits 111
9.5 Conclusions 112

10 Conclusions 113
10.1 Summary 113
10.2 Contributions 115
10.3 Future Work 116

References 117

1 Introduction

Many numerical problems can be solved by a dense linear system of equations. Therefore, the solution of systems like $Ax = b$ with an $n \times n$ matrix $A \in \mathbb{R}^{n \times n}$ and a right hand side $b \in \mathbb{R}^n$ is very usual in numerical analysis. This is true for linear equations that occur like in partial differential equations and integral equations that appear in many problems of Physics and Engineering [16]. Many different numerical algorithms contain this task as a subproblem.

When such a system is solved using computers, many rounding errors can occur during its computation. Even well conditioned systems can lead to completely wrong results and this effect can be worse for ill conditioned matrices.

Even very small systems may deliver a wrong result due to lack of accuracy. This problem can easily be seen in the following system:

$$A = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

This is a small system, but it is very ill conditioned with an estimated condition of $1.2 \cdot 10^{17}$. The correct solution would be

$$x_1 = \frac{a_{22}}{a_{11}a_{22} - a_{12}a_{21}} = 205117922, \quad x_2 = \frac{-a_{21}x_1}{a_{22}} = 83739041.$$

However, solving this problem in a Intel Itanium2 processor, with 1.5 GHz, 12 GB of main memory, and using IEEE double precision arithmetic, it would lead to the following wrong result:

$$\tilde{x}_1 = 102558961 \quad \tilde{x}_2 = 41869520.5$$

One possible solution to cope with that would be the use of verified computing [8]. Verified computing provides an interval result that surely contains the correct result [44, 55]. The classical solutions for linear systems of equations using floating-point arithmetic can only deliver an approximation. Since the correct result is unknown, it is not clear how good these approximations are, or if there exists a unique solution at all. From a mathematical point of view, such results might be worthless. In contrast, a verified algorithm will, in general, succeed in finding an enclosure of the solution. If the solution is not found, the algorithm will let the user know and ensure that no wrong result will be delivered. This indicates that the problem is ill conditioned or unsolvable, i.e. there exists no solution. Then the user may e.g. apply a different algorithm. This is the most important advantage of such algorithms compared with ordinary methods. Numerical applications providing automatic result verification may be useful in many fields like Simulation and Modeling [16]. Especially, when accuracy is mandatory in safety critical applications.

The program committee organizing Validated Computing 2002 [17] wrote: "Ever increasing reliance on computer systems brings ever increasing need for reliability. Validated computing is one essential technology to achieve increased software reliability. Validated computing uses controlled rounding of computer arithmetic to guarantee that hypotheses of suitable mathematical theorems are (or are not) satisfied. Mathematical rigor in the computer arithmetic, in algorithm design, and in program execution allow us to guarantee that the stated problem has (or does not have) a solution in an enclosing interval we compute. If the enclosure is narrow, we are certain that we know the answer reliably and accurately. If the enclosing interval is wide, we have a clear warning that our uncertainty is large, and a closer study is demanded. Intervals capture uncertainty in modeling and problem formulation, in model parameter estimation, in algorithm truncation, in operation round off, and in model interpretation".

The verified method for solving linear systems of equations is based on the enclosure theory and uses an interval Newton-like iteration and Brouwer´s fixed point theorem. It is composed by the computation of an approximate solution and the verification step, increasing the computational cost to solve such a system [70], specially when dealing with large matrices. The research already developed shows that the execution times of verified algorithms using e.g. C-XSC (C for eXtended Scientific Computing) [44] are much larger than the execution times of algorithms that do not use this concept, even for parallel implementations [33, 34, 46, 51]. Additionally, if the input data are an interval matrix and an interval vector, this cost is two times larger since every interval is defined by two numbers: the infimum and the supremum or the midpoint and the radius of the interval (see section 2.3). The additional effort is unavoidable to guarantee the quality of the result. Therefore we have to use all available techniques to accelerate the computational process.

In this context, the use of high performance computation techniques appears as a useful tool to drop down the time needed to solve point and interval systems using verified computing. In recent years, parallel computing techniques and platforms have become widespread. With the advent of large clusters composed by hundreds or thousands of nodes connected through a gigabit network, huge linear systems can be computed in parallel [24].

Nowadays, many persons already have multi core processors at home. It is possible to buy CPUs with dual core and even quad core processors. Intel has also a prototype of a 80 core processor, that will be available in 5 years [86]. Soon everyone will be able to use parallel computers not just in computational centers, and clusters, but also at home, in their own notebooks and PCs.

The parallel solutions for linear solvers found in the literature explore many aspects and constraints related to the adaptation of the numerical methods to high performance environments [26, 60]. However, those implementations do not deal with verified methods. The well-known libraries such as PBLAS (Parallel Basic Linear Algebra Subprograms) and ScaLAPACK (Scalable Linear Algebra PACKage) [5] alone are not suited for interval systems based on verified methods since these methods introduce several steps to deliver a guaranteed result.

In the field of verified computing many important contributions have been done in the last past years. Some works related to verified solvers for dense linear systems [25,31,44,69,75] can be found in the literature. However, only a few research on verified solvers for dense systems together with parallel implementations were found [43,77,87], but these authors implement other numerical problems (e.g. global optimization) or use a parallel approach for other architectures than clusters.

To compensate the lack of performance of such verified algorithms, some of those works suggest the use of midpoint-radius arithmetic since its implementation can be done using only floating-point operations [77, 78]. This would be a way to increase the performance of verified methods. These ideas were implemented on a verified solver in an interval toolbox called INTLAB. However, this solver can be used just together with the commercial MATLAB environment, what can increase the costs to prohibitive values.

C-XSC (C for eXtended Scientific Computation) is a programming environment for verified computing developed at Karlsruhe University. It is a free programming tool for the development of numerical algorithms which provide highly accurate and automatically verified results. Despite the advantage of been free and providing accurate and verified results, this tool presents a limiting disadvantage: its execution time for large matrices can last several hours.

Therefore, a comparison of verified computing tools and the well known LAPACK is presented aiming to find the best sequential library to be used in a new implementation. This comparison takes into account the availability, accuracy and performance of these tools.

A first parallel approach uses C-XSC and MPI. Some parts of the C-XSC solver for dense linear systems were parallelized for Cluster computers [46, 47, 51]. In this research, two main parts of this method (which represents the larger part of the computational costs), were studied, parallelized and optimized. This implementation achieved significant speed-ups, reinforcing the statement that the parallelization of such a method is an interesting alternative to increase its usability. However, the total time still remains much higher than the time of a floating-point linear system solver in other libraries.

Based on the comparison of existing tools and the results of the first parallel approach, decisions about a new sequential implementation were taken. The main idea was to use popular and highly optimized libraries to gain performance and verified methods to have guaranteed results, providing both self-verification and speed-up at the same time. In other words, these parallel implementations try to join the best aspects of each library:

- LAPACK and BLAS: performance;
- Verified method like the one used in C-XSC toolbox: guaranteed results.

Sequential versions were implemented for infimum-supremum and midpoint-radius arithmetic, for point and interval input data. Performance tests showed that the midpoint-radius algorithm needs approximately the same time to solve a linear system with point or interval input data, while the infimum-supremum algorithm needs much more time for interval data. Considering that, midpoint-radius arithmetic was the natural choice for the next step of this work: the parallel implementation.

Aiming to solve large systems very fast, this new implementation was parallelized. The parallel version uses ScaLAPACK and PBLAS libraries to maintain the idea of using highly optimized libraries. The algorithms should find an enclosure even for very ill conditioned problems with condition number up to $2.5 \cdot 10^{15}$ on a personal computer. Both point and interval systems were considered in order to treat the uncertainty of the input data.

A final version of this parallel implementation was written to solve very large matrices, e.g. dimension $100\,000$. In this implementation, each processor generates its own part of the matrix A and the vector b, different from the other version where one processor creates the whole matrix and distributes it.

An experiment using shared memory was implemented using threads and was specially designed for dual-core machines. The idea is to avoid the changing of rounding mode, i.e., each processor independently uses its own floating-point rounding strategy to do the computations. This strategy can reduce the computation time, since changing the rounding mode is a very expensive operation compared to floating-point operations.

Finally, the usage of a reliable linear system solver to compute the solution of problem 7 of the SIAM 100-digit challenge [1, 83] is presented. To find the result, a system of linear equations with dimension $20\,000 \times 20\,000$ had to be solved using interval computations.

Moreover, the major goal of this research is to provide a free, fast, reliable and accurate solver for point and interval dense linear systems.

The organization of this document is as follows: the next chapter presents the background needed to understand the premises of this research. A comparison of several tools and the presentation of its methods can be found in Chapter 3. Chapter 4 shows the first parallel approach using the C-XSC method. Chapter 5 presents the new sequential algorithms implemented using infimum-supremum and midpoint-radius arithmetic and a comparison of those implementations. The parallel approach with some experimental results can be found in Chapter 6. Chapter 7 presents an optimization of the parallel method for very large matrices and tests in two different clusters. The experiments using shared memory and the usage of a reliable linear systems solver are shown in Chapters 8 and 9 respectively. Finally, Chapter 10 presents the conclusion and future work.

2 Background

This Chapter presents some important basic concepts concerning the background of this thesis. Both mathematical and computational aspects are considered. Section 2.1 presents the arithmetic used in computers followed by some concerns about uncertain data in Sections 2.2. The arithmetic for interval numbers is defined in Section 2.3. The main concepts about verified computing can be found in Section 2.4. Section 2.5 discusses the traditional verified methods for solving linear systems. Finally, Section 2.6 presents an overview about parallel computing.

2.1 Computer Arithmetic

Computers have two different number systems built into their hardware: integers and floating-point numbers [31]. Integer arithmetic and its software extensions operate without error provided that there is no overflow. In this way, the computer carries along as many digits as necessary to represent the result of an operation exactly. On the other hand, in scientific analysis, one works with real numbers. A real number can be represented by an infinite decimal or binary fraction. For numerical or scientific computation, real numbers must be approximated in a computer by a finite fraction called a floating-point number. Floating-point arithmetic is built into the hardware, and is consequently very fast. However, each floating-point operation is subject to error. Although, each floating-point operation on most computers is of maximum accuracy (i.e. the exact result and the computed result of each operation differ only by one unit in the last place), the result after only few operations can be completely wrong. The following examples show that effect.

Let $x = (10^{20}, 1223, 10^{18}, 10^{15}, 3, -10^{12})$ and $y = (10^{20}, 2, -10^{22}, 10^{13}, 2111, 10^{16})$ be two real vectors. The scalar product is denoted by $x \cdot y$. The result using exact integer arithmetic is:

$$x \cdot y = 10^{40} + 2446 - 10^{40} + 10^{28} + 6333 - 10^{28} = 8779$$

In contrast, depending on the order of the summation, the floating-point arithmetic on a computer can give a completely wrong result, e.g. the value zero for this scalar product [31]. The reason for this is that the summands are of such different orders of magnitude that they cannot be processed correctly in ordinary floating-point format. Some libraries, e.g. BLAS, are implemented using some tests to define the order of the summation and reduce the occurrence of this type of error.

Another example can be shown when considering a floating-point system with base 10 and a mantissa length of 5 digits. The goal is to evaluate the difference of two numbers $x = 0.10005 \cdot 10^5$ and $y = 0.99973 \cdot 10^4$ using floating-point arithmetic. In this example, both operands are of the same order of magnitude. The computer gives the completely correct result $x - y = 0.77000 \cdot 10^1$. But if those two numbers x and y are the result of two previous multiplications, the result could be different. Since we are dealing with 5-digit arithmetic, these products of course have 10 digits. Suppose the unrounded products are

$$x_1 \cdot x_2 = 0.1000548241 \cdot 10^5 \text{ and } y_1 \cdot y_2 = 0.9997342213 \cdot 10^4.$$

If we subtract these two numbers, normalize the result and round it to 5 places, we get $0.81402 \cdot 10^1$, which differs in every digit from the previous result computed in floating-point arithmetic. In this case, the value of the expression $x_1 \cdot x_2 - y_1 \cdot y_2$ was computed to the closest 5-digit floating-point number. In contrast, pure floating-point arithmetic with rounding after each individual operation gave a completely wrong result.

From the mathematical point of view, the problem of correctness of computed results is of central importance because of the high computational speeds attainable today [31]. The determination of the correctness of computed results is essential in many applications such as simulation or mathematical modeling. One needs an assurance of correct computational results to distinguish between the inaccuracies in computation and the actual properties of the model.

2.1.1 The IEEE Standard

When implementing any arithmetic on a computer, the biggest problem is that a computer can represent in practice only a finite set of numbers. The best representation of real numbers on computers is the floating-point arithmetic. It is used today in most computers as described in the IEEE Standard for Binary Floating-point Arithmetic [67]. The main goal of IEEE 754 is to define how to implement floating-point binary arithmetic. It defines the representation, basic operations: addition, subtraction, multiplication, division square root, comparison and situations that will lead to exceptions.

IEEE 754 is a binary standard that requires a base $b = 2$, and precision $p = 24$ for single precision and $p = 53$ for double precision. It also specifies the precise layout of bits in single and double precision. Single precision is encoded in 32 bits using 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand. However, it uses a hidden bit, so the significand is 24 bits ($p = 24$), even though it is encoded using only 23 bits. Double precision is encoded in 64 bits using 1 bit for the sign, 11 bits for the exponent, and 52 bits for the significand plus a hidden bit.

The IEEE standard defines four different precisions: single, double, single-extended, and double-extended. Today, nearly every floating-point processors implements operations for single and double precision according to the IEEE standard. Support for extended precision is not so common. Single precision occupies a single 32 bit word, double precision two consecutive 32 bit words. Extended precision is a format that offers at least a little extra precision and exponent range. More detailed information can be seen in table 1.

Table 1 – IEEE Data Types.

Parameter	Single	Single-Extended	Double	Double-Extended
p	24	≥ 32	53	≥ 64
e_{max}	127	≥ 1023	1023	> 16383
e_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

In the IEEE standard, rounding occurs whenever an operation has a result that is not exact. Such a rounded operation (with the exception of binary decimal conversion) is performed as if each operation is computed exactly and then rounded. By default, rounding means round toward nearest. The standard requires that three other rounding modes must be provided, namely round toward 0, round toward $+\infty$, and round toward $-\infty$.

Since 2001, a new revision of IEEE 754 is under way. The addition of a half (16-bit) and quadruple (128-bit) precision for the binary format is one of the most important enhancements of this revision, as well as a description of three new decimal formats, matching the lengths of the 32/128-bit binary formats. These give decimal formats with 7, 16, and 34-digit significands, which may be normalized or unnormalized. Additional important improvements are presented in the following list:

- the round-to-nearest, ties away from zero rounding mode has been added (required for decimal operations only);
- review of the definitions for clarification and consistency;
- min and max with specifications for the special cases ± 0, $\pm\infty$;
- inclusion of several new operations like fused multiply-add (FMA), classification predicates (isnan(x), etc.), various min and max functions, a total ordering predicate, and two decimal-specific operations (samequantum and quantize).

As of May 2008, the IEEE 754r draft is in the third phase; it has completed eight ballots and has been submitted for approval as an IEEE standard. On 11 June 2008, it was approved unanimously by the IEEE Revision Committee (RevCom), and it was formally approved by the IEEE-SA Standards Board on 12 June 2008. The new IEEE 754 [62] (formally IEEE Std 754-2008, the IEEE Standard for Floating-Point Arithmetic) was published by the IEEE Computer Society on 29 August 2008 [68].

The Standardization of Interval Arithmetic

Since January a group of interval researchers are discussing about a Standardization for Interval Arithmetic. On 12 June 2008 the IEEE-SA Standards Board approved the project of a committee for the standardization of interval arithmetic, until 31 December 2012. This project bears the number P1788.

This group proposes editorial changes to draft 1.9.0 of the IEEE standard for floating-point arithmetic to specify optional support for interval arithmetic and complete arithmetic. The intent is to specify how these arithmetics should be implemented if a system chooses to do so, not to require that they be provided.

When completed, the standard is designed to help improve the availability of reliable computing in modern hardware and software environments by defining the basic building blocks needed for performing interval arithmetic. While systems for interval arithmetic are in use, the lack of existing standards inhibited development, portability and the ability to verify correctness of codes, said the IEEE [36].

This process is just in the beginning. There are no practical decisions until now and many aspects will be discussed. This is an evidence that the community had recognized how important interval implementations are, and that interval arithmetic plays an important role in the future of computer science. As IEEE says: "Interval arithmetic is a method of putting bounds on rounding errors in mathematical computation to yield more reliable results" [36].

2.2 Uncertain Data

Engineering and scientific problems are frequently modeled by a numerical simulation on a computer. Such simulations have many advantages:

- usually, a computer simulation is much less expensive; e.g. new vehicles are often designed and tested in a “numerical wind tunnel" and in simulated crash tests instead of building a prototype and performing real tests (which may even lead to the destruction of the prototype);
- this is also less time consuming and leads to a much faster design process;
- a simulation does not have the risks which may be caused by a true experiment, e.g. in nuclear energy or aviation and space engineering;
- many experiments are difficult to perform or they even cannot be performed at all; e.g. in astronomy, particle physics, biology and geology.

However, the question arises, how reliable are the results of such numerical simulations. A reliable solution of an engineering or scientific problem should not only give an approximate numerical result but also rigorous error bounds and a proof that the solution is correct within these bounds; e.g. that a solution exists (and possibly is unique) in a mathematical sense.

There are many sources of uncertainty or error which have to be taken care of in a numerical simulation. These include:

- measurement errors, unknown or imprecise data;
- errors introduced by the mathematical model of the problem, frequently described by differential equations;
- mathematical errors introduced by the discretization / linearization of the model, leading to linear systems;
- rounding errors in the floating-point evaluation of the numerical approximation.

Interval arithmetic methods can handle most of these sources of error:

- imprecise measured data can be represented by intervals;
- errors in the model or in the mathematical treatment of the model can be represented by an additional interval term;
- rounding errors may be controlled by interval arithmetic operations.

Another important aspect is that intervals represent the continuum of numbers: a floating-point interval $[a, b]$ is the set of all real numbers between a and b, not just the floating-point numbers. Therefore, in contrast to floating-point computations, interval methods can give rigorous mathematical results; e.g. let $F(X)$ be an interval extension of a mathematical function $f(x)$, and let $F([a, b]) > F([c, d])$, then there exists no global maximum of the function f(x) in the interval $[c, d]$, and no global minimum in $[a, b]$, respectively.

In this research, a special aspect of this numerical simulation is considered: linear equations with interval coefficients. These interval coefficients may be the result of an imprecise model or of measurement errors. Therefore it is important to compute an interval enclosure of the result set.

2.3 Interval Arithmetic

The power of interval arithmetic lies in the possibility of achieving verified results on computers. In particular, rounded interval arithmetic allows rigorous enclosures for the ranges of operations and functions [42]. An enclosure can be defined as an interval that contains the correct result. This makes a qualitative difference in scientific computations, since the results are now intervals in which the exact result must lie.

Let $\mathbb{R}$ denote the set of real numbers and $\mathbb{PR}$ the power set over $\mathbb{R}$. The two most frequently used representations for intervals over $\mathbb{R}$, are the infimum-supremum representation

$$[a_1, a_2] := \{x \in \mathbb{R} : a_1 \leq x \leq a_2\} \text{ for some } a_1, a_2 \in \mathbb{R},\ a_1 \leq a_2, \tag{2.1}$$

and the midpoint-radius representation

$$\langle a, \alpha \rangle := \{x \in \mathbb{R} : |x - a| \leq \alpha\} \text{ for some } a \in \mathbb{R},\ 0 \leq \alpha \in \mathbb{R}. \tag{2.2}$$

Throughout the thesis we refer by $\mathbb{I}^{+}\mathbb{R}$ for midpoint-radius interval representation and $\mathbb{I}^{*}\mathbb{R}$ for infimum-supremum representation. Operations in $\mathbb{I}^{+}\mathbb{R}$ and $\mathbb{I}^{*}\mathbb{R}$ are denoted by $\oplus, \ominus, \odot, \oslash$. The same symbols are used for operations on interval vectors $\mathbb{I}^{(+/*)}\mathbb{R}^{n}$ and interval matrices $\mathbb{I}^{(+/*)}\mathbb{R}^{n \times m}$, and operations among those. The corresponding power set operations are denoted by $+, -, \cdot, /$. To refer the set of floating-point numbers, we use $\mathbb{F}$. Intervals using floating-point numbers are represented as $\mathbb{I}^{+}\mathbb{F}$ for midpoint-radius interval representation and $\mathbb{I}^{*}\mathbb{F}$ for infimum-supremum representation. The following symbols are used for the directed roundings: $\bigtriangledown$ for rounding down, $\square$ for rounding to nearest and $\bigtriangleup$ for rounding up.

Interval operations always satisfy the fundamental property of isotonicity:

$$\forall A, B, C, D \in \mathbb{IF},\ A \subseteq C, B \subseteq D \Longrightarrow A \circledcirc B \subseteq C \circledcirc D \mid \circ \in \{+, -, \cdot, /\} \tag{2.3}$$

The special case for point data is the following:

$$\forall a \in A,\ \forall b \in B : a \circ b \in A \circledcirc B \mid \circ \in \{+, -, \cdot, /\} \tag{2.4}$$

and all suitable A, B. Any representation and implementation of interval arithmetic must obey isotonicity.

The next sections present the infimum-supremum and midpoint-radius representations and issues of implementing these on computers.

2.3.1 Infimum-Supremum Interval Arithmetic

Considering the real intervals $A = [a_1, a_2] := \{x \in \mathbb{R} : a_1 \leq x \leq a_2\}\ for\ a_1, a_2 \in \mathbb{R},\ a_1 \leq a_2$ and $B = [b_1, b_2] := \{x \in \mathbb{R} : b_1 \leq x \leq b_2\}\ for\ b_1, b_2 \in \mathbb{R},\ b_1 \leq b_2$. The basic operations are defined as follows [18]:

$$A + B := [(a_1 + b_1), (a_2 + b_2)] \quad (2.5)$$

$$A - B := [(a_1 - b_2), (a_2 - b_1)] \quad (2.6)$$

$$A \cdot B := [min\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}, max\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}] \quad (2.7)$$

$$1/B := [1/b_2, 1/b_1] \; where \; 0 \notin B \quad (2.8)$$

$$A/B := [min\{a_1/b_1, a_1/b_2, a_2/b_1, a_2/b_2\}, max\{a_1/b_1, a_1/b_2, a_2/b_1, a_2/b_2\}] \; for \; 0 \notin B \quad (2.9)$$

All operations satisfy the isotonicity property (2.4).

Infimum-Supremum Interval Arithmetic for Computers

As shown in section 2.1.1, a computer with IEEE 754 arithmetic allows the result of an arithmetic operation to be rounded down to the nearest machine number less than or equal to the mathematically correct result, rounded up to the nearest machine number greater than or equal to the mathematically correct result, or rounded to the machine number nearest to the mathematically correct result [42]. For example, take $C = A + B = [(a_1 + b_1), (a_2 + b_2)]$. If $a_1 + b_1$ is rounded down after computation and $a_2 + b_2$ is rounded up after computation, then the resulting interval $C = [c_1, c_2]$ that is represented in the machine must contain the exact range of $a + b$ for $a \in A$, $b \in B$.

The following algorithms describe the operations using infimum-supremum arithmetic and IEEE 754 arithmetic standard. Let

$$A = [a_1, a_2] \in \mathbb{I}^*\mathbb{F} \text{ and } B = [b_1, b_2] \in \mathbb{I}^*\mathbb{F}$$

be given. Let the symbols $\Box$, $\triangle$ and $\bigtriangledown$ mean that the operation will be done respectively with rounding to nearest, rounding up and rounding down. The definition of interval addition $C := A \oplus B \in \mathbb{I}^*\mathbb{F}$, with $C = [c_1, c_2]$ can be found in Algorithm 1.

Algorithm 1 Infimum-Supremum Addition in $\mathbb{I}^*\mathbb{F}$.

```
1: $c_1 = \bigtriangledown(a_1 + b_1)$
2: $c_2 = \triangle(a_2 + b_2)$
```

The interval subtraction $C := A \ominus B \in \mathbb{I}^*\mathbb{F}$, with $C = [c_1, c_2]$ is defined in Algorithm 2.

Algorithm 2 Infimum-Supremum Subtraction in $\mathbb{I}^*\mathbb{F}$.

```
1: $c_1 = \bigtriangledown(a_1 - b_2)$
2: $c_2 = \triangle(a_2 - b_1)$
```

The interval multiplication $C := A \odot B$ is not so trivial as the addition and subtraction. It´s definition can be seen in Algorithm 3.

The definition $C := 1 \oslash B \in \mathbb{I}^*\mathbb{F}$ is presented in Algorithm 4.

Algorithm 3 Infimum-Supremum Multiplication in $\mathbb{I}^*\mathbb{F}$.

```
1: $aux1 = \bigtriangledown(a_1 \cdot b_1)$
2: $aux2 = \bigtriangledown(a_1 \cdot b_2)$
3: $aux3 = \bigtriangledown(a_2 \cdot b_1)$
4: $aux4 = \bigtriangledown(a_2 \cdot b_2)$
5: $c_1 = min\{aux1, aux2, aux3, aux4\}$
6: $aux1 = \bigtriangleup(a_1 \cdot b_1)$
7: $aux2 = \bigtriangleup(a_1 \cdot b_2)$
8: $aux3 = \bigtriangleup(a_2 \cdot b_1)$
9: $aux4 = \bigtriangleup(a_2 \cdot b_2)$
10: $c_2 = max\{aux1, aux2, aux3, aux4\}$
```

Algorithm 4 Infimum-Supremum Inversion in $\mathbb{I}^*\mathbb{F}$.

```
1: $c_1 = \bigtriangledown(1/b_2)$
2: $c_2 = \bigtriangleup(1/b_1)$
```

The interval division $C := A \oslash B$ is defined (using the previous definitions of interval multiplication and interval inversion) by $C := A \otimes (1 \oslash B)$.

In each operation, the switching of rounding mode appears many times. The cost of this operation is much higher than a floating-point operation. Next section presents a strategy to avoid the frequent switching of rounding mode.

Switching the Rounding Mode in a Clever Way

In interval operations, lower bounds have always to be rounded down, and upper bounds have to be rounded up. Therefore, in a straightforward implementation, the rounding mode has to be switched twice per interval operation [10].

On many common processors (e.g. Intel Pentium), switching the rounding mode is a very expensive operation which may require up to 10 times as much execution time as a floating-point operation:

- without switching rounding modes: operations are pipelined, i.e. 1 clock cycle per operation (addition, multiplication of registers);
- with switching rounding modes: operation *fldcw* costs 7 cycles, and arithmetic operations cannot be pipelined, i.e. 3 clock cycles per operation;
- above code needs 20 clock cycles instead of 2, i.e. the overhead is 900 $\%$.

Therefore, the rounding mode should be switched as infrequently as possible.

In interval vector- and matrix operations, switching the rounding mode can be avoided by modifying the sequence of operations:

1. set the rounding mode to "round down";

2. compute all lower bounds of all components;
3. set the rounding mode to "round up";
4. compute all upper bounds.

Here, the rounding mode is switched only twice instead of $2{\cdot}n$ times. This method is applied e.g. in INTLAB [38], however it can only be applied in vector and matrix operations, not in single operations.

The formula $\triangledown(a) = - \triangle(-a)$ can be used, to replace the switching of rounding modes by negations. Negation usually requires much less overhead than switching the rounding modes. The number of additional negations can be minimized by storing intervals in a special representation: instead of infimum and supremum, an interval is stored in the form infimum and negated supremum. This negation has to be taken care of in all component references and arithmetic operations [9]. The overhead (in terms of additional negations) is between 0 and 2 negations per interval operations. On the average, the overhead is about 10%.

This implementation is based on several assumptions: in particular, the rounding mode must be preserved between operations, and the logical sequence of operations must be preserved. The compiler should not optimize the negations that avoid the switching of rounding mode several times.

Applying these ideas in each operation, the algorithms 5, 6, 7, define respectively the addition, subtraction, and inversion using this strategy. It is possible to see that no extra effort must be done, despite some negations.

Algorithm 5 Optimized Infimum-Supremum Addition in $\mathbb{I}^*\mathbb{F}$.

```
1: $c_1 = \triangledown(a_1 + b_1)$
2: $c_2 = - \triangledown((-a_2) - b_2)$
```

Algorithm 6 Optimized Infimum-Supremum Subtraction in $\mathbb{I}^*\mathbb{F}$.

```
1: $c_1 = \triangledown(a_1 - b_2)$
2: $c_2 = - \triangledown((-a_2) + b_1)$
```

Algorithm 7 Optimized Infimum-Supremum Inversion in $\mathbb{I}^*\mathbb{F}$.

```
1: $c_1 = \triangledown(1/b_2)$
2: $c_2 = - \triangledown(1/(-b_1))$
```

The multiplication definition can be found in Algorithm 8. This operation demands several case distinctions, and even being able to save some time avoiding to switch the rounding mode, these case distinctions will delay the algorithm.

The requirements for this implementation are:

Algorithm 8 Optimized Infimum-Supremum Multiplication in $\mathbb{I}^*\mathbb{F}$.

```
1: $setrounding = down$
2: **if** $a_1 >= 0.0$ **then**
3:    **if** $b_1 >= 0.0$ **then**
4:       $c_1 = a_1 \cdot b_1$ and $c_2 = -(a_2 \cdot -b_2)$ {case: $a$ and $b$ are positive}
5:    **else**
6:       **if** $b_2 > 0.0$ **then**
7:          $c_1 = a_2 \cdot b_1$ and $c_2 = -(a_2 \cdot -b_2)$ {case: $a$ is positive and $b$ contains $0$}
8:       **else**
9:          $c_1 = a_2 \cdot b_1$ and $c_2 = -(a_1 \cdot -b_2)$ {case: $a$ is positive and $b$ is negative}
10:      **end if**
11:   **end if**
12: **else**
13:   **if** $a_2 > 0.0$ **then**
14:      **if** $b_1 >= 0.0$ **then**
15:         $c_1 = a_1 \cdot b_2$ and $c_2 = -(a_2 \cdot -b_2)$ {case: $a$ contains $0$ and $b$ is positive}
16:      **else**
17:         **if** $b_2 > 0.0$ **then**
18:            $aux1 = -(a_1 \cdot -b_2)$ and $aux2 = -(a_2 \cdot -b_1)$ {case: $a$ and $b$ contains $0$}
19:            $c_1 = min(aux1, aux2)$ and $c_2 = -(a_2 \cdot -b_2)$
20:         **end if**
21:      **else**
22:         $c_1 = a_2 \cdot b_1$ and $c_2 = -(a_1 \cdot -b_1)$ {case: $a$ contains $0$ and $b$ is negative}
23:      **end if**
24:   **end if**
25: **else**
26:   **if** $b_1 >= 0.0$ **then**
27:      $c_1 = a_1 \cdot b_2$ and $c_2 = -(a_2 \cdot -b_1)$ {case: $a$ is negative and $b$ is positive}
28:   **else**
29:      **if** $b_2 > 0.0$ **then**
30:         $c_1 = a_1 \cdot b_2$ and $c_2 = -(a_1 \cdot -b_1)$ {case: $a$ is negative and $b$ contains $0$}
31:      **else**
32:         $c_1 = a_2 \cdot b_2$ and $c_2 = -(a_1 \cdot -b_1)$ {case: $a$ and $b$ are negative}
33:      **end if**
34:   **end if**
35: **end if**
```

- There must be a possibility to select the rounding mode. This should be no problem in C, but it is not possible for basic data types in e.g. Java;

- The rounding mode must be preserved between operations (the operating system, runtime system, etc. should not change it);

- No "round to nearest" floating-point operations should be performed between interval operations - programs should be split in floating-point part and interval part;

- The logical sequence of operations must be preserved, and no changes should be performed by optimizing compiler and also no out of order execution in processor.

2.3.2 Midpoint-Radius Interval Arithmetic

In this section, midpoint-radius representation, also known as circular arithmetic, will be defined. Considering the real intervals $A = \langle a, \alpha \rangle := \{x \in \mathbb{R} : a - \alpha \leq x \leq a + \alpha\}\ for\ a \in \mathbb{R},\ 0 \leq \alpha \in \mathbb{R}$ and $B = \langle b, \beta \rangle := \{x \in \mathbb{R} : b - \beta \leq x \leq b + \beta\}\ for\ b \in \mathbb{R},\ 0 \leq \beta \in \mathbb{R}$. The basic operations are defined like in [2]:

$$\begin{aligned} A + B &:= \langle a + b, \alpha + \beta \rangle & (2.10)\\ A - B &:= \langle a - b, \alpha + \beta \rangle & (2.11)\\ A \cdot B &:= \langle a \cdot b, |a|\beta + \alpha|b| + \alpha\beta \rangle & (2.12)\\ 1/B &:= \langle b/D, \beta/D \rangle\ where\ D := b^2 - \beta^2\ and\ 0 \notin B & (2.13)\\ A/B &:= A \cdot (1/B)\ for\ 0 \notin B & (2.14) \end{aligned}$$

All operations satisfy the isotonicity property (2.4).

Midpoint-Radius Interval Arithmetic for Computers

In the computer implementation of interval arithmetic, special care has to be taken for the rounding [77]. Consider a set $\mathbb{F} \subseteq \mathbb{R}$ of real floating-point numbers, and define

$$\mathbb{I}^{+}\mathbb{F} := \{\langle \tilde{a}, \tilde{\alpha} \rangle : \tilde{a}, \tilde{\alpha} \in \mathbb{F},\ \tilde{\alpha} \geq 0\}$$

As before, we set

$$\langle \tilde{a}, \tilde{\alpha} \rangle := \{x \in \mathbb{R} : \tilde{a} - \tilde{\alpha} \leq x \leq \tilde{a} + \tilde{\alpha}\}$$

Then $\mathbb{I}^{+}\mathbb{F} \in \mathbb{IR}$. Note that the pair of floating-point numbers $\tilde{a}, \tilde{\alpha}$ describes an infinite set of real numbers for $\tilde{\alpha} \neq 0$. Also note that, there will not always exist a pair of floating-point numbers $\tilde{a}_1, \tilde{a}_2 \in \mathbb{F}$ with $[\tilde{a}_1, \tilde{a}_2] = \langle \tilde{a}, \tilde{\alpha} \rangle$.

Moreover, the smallest nonzero relative precision of an interval in infimum-supremum representation is limited by the relative rounding error unit ϵ , whereas much narrower intervals are possible in midpoint-radius representation.

Assuming that the floating-point arithmetic satisfies the IEEE 754 arithmetic standard [67], this implies availability of rounding modes rounding to nearest, rounding downwards and rounding upwards, among other definitions.

When implementing this arithmetic on computers, an error will be generated in the midpoint evaluation. This error should be compensated using the relative error unit. Denote the relative rounding error unit by ϵ, set $\epsilon' = \frac{1}{2}\epsilon$, and denote the smallest representable (unnormalized) positive floating-point number by η. In IEEE 754 double precision, $\epsilon = 2^{-52}$ and $\eta = 2^{-1074}$. If an expression is to be evaluated in floating-point arithmetic, we put the expression in parentheses with a preceding rounding symbol. Note that all operations within the expression are performed with the same rounding.

As presented in [77], the following algorithms describe the operations using midpoint-radius arithmetic and IEEE 754 arithmetic standard. Let

$$A = \langle \tilde{a}, \tilde{\alpha} \rangle \in \mathbb{I}^{+}\mathbb{F} \; and \; B = \langle \tilde{b}, \tilde{\beta} \rangle \in \mathbb{I}^{+}\mathbb{F}$$

be given. Let the symbols $\Box$, $\triangle$ and $\bigtriangledown$ mean that the operation will be done respectively with rounding to nearest, rounding up and rounding down. Then the interval addition and subtraction $C := A \circledcirc B \in \mathbb{I}^{+}\mathbb{F}, \circ \in \{+, -\}$, with $C = \langle \tilde{c}, \tilde{\gamma} \rangle$ is defined by the following algorithm:

Algorithm 9 Midpoint-Radius Addition and Subtraction in $\mathbb{F}$.

```
1: $\tilde{c} = \Box(\tilde{a} \circ \tilde{b})$
2: $\tilde{\gamma} = \triangle(\epsilon'|\tilde{c}| + \tilde{\alpha} + \tilde{\beta})$
```

The interval multiplication $C := A \otimes B$ is defined by the algorithm 10.

Algorithm 10 Midpoint-Radius Multiplication in $\mathbb{F}$.

```
1: $\tilde{c} = \Box(\tilde{a} \cdot \tilde{b})$
2: $\tilde{\gamma} = \triangle(\eta + \epsilon'|\tilde{c}| + (|\tilde{a}| + \tilde{\alpha})\tilde{\beta} + \tilde{\alpha}\tilde{\beta})$
```

The following definition was used for the interval reciprocal $C := 1 \oslash B \in \mathbb{F}$, assuming $\tilde{\beta} < |\tilde{b}|$, presented in Algorithm 11.

Algorithm 11 Midpoint-Radius Inversion in $\mathbb{F}$.

```
1: $\tilde{c_1} = \bigtriangledown((-1)/(-|\tilde{b}| - \tilde{\beta}))$
2: $\tilde{c_2} = \triangle((-1)/(-|\tilde{b}| - \tilde{\beta}))$
3: $\tilde{c} = \triangle(\tilde{c_1} + 0.5(\tilde{c_2} - \tilde{c_1}))$
4: $\tilde{\gamma} = \triangle(\tilde{c} - \tilde{c_1})$
5: $\tilde{c} = sign(\tilde{b})\tilde{c}$
```

It is important to say that in the steps 1 and 2 of Algorithm 11 all the operations are performed with the same rounding. When using different roundings, the operation would be like in Algorithm 12.

Algorithm 12 Rounding Alternative to Midpoint-Radius Inversion in $\mathbb{F}$.

```
1: $aux = \triangle(|\tilde{b}| + \tilde{\beta})$
2: $\tilde{c_1} = \bigtriangledown(1/(aux)$
```

The interval division $C := A \oslash B$ is defined as before by $C := A \otimes (1 \oslash B)$.

Rump also presented algorithms for scalar product and matrix multiplication [77]. Both are very useful in the implementation of linear systems. Towards vector and matrix operations consider a scalar product A^TB for $A, B \in \mathbb{I}^+\mathbb{F}$, $A = \langle\tilde{a}, \tilde{\alpha}\rangle$, $B = \langle\tilde{b}, \tilde{\beta}\rangle$ with $\tilde{a}, \tilde{b}, \tilde{\alpha}, \tilde{\beta} \in \mathbb{F}^n, \tilde{\alpha}, \tilde{\beta} \geq 0$. Algorithm 13 presents the implementation for scalar products, where the result is an inclusion of the exact solution.

Algorithm 13 Midpoint-Radius Scalar Product in $\mathbb{F}^n$.

```
1: $\tilde{c_1} = \bigtriangledown(\tilde{a}^T\tilde{b})$
2: $\tilde{c_2} = \triangle(\tilde{a}^T\tilde{b})$
3: $\tilde{c} = \triangle(\tilde{c_1} + 0.5(\tilde{c_2} - \tilde{c_1}))$
4: $\tilde{\gamma} = \triangle(\tilde{c} - \tilde{c_1}) + (|\tilde{a}|^T + \tilde{\alpha}^T)\tilde{\beta} + \tilde{\alpha}^T|\tilde{b}|$
```

All estimations are independent of the order of summation [77]. Therefore, midpoint-radius arithmetic may take full advantage of any computational scheme to compute scalar and matrix products. This includes especially vector and parallel architectures, blocked algorithms etc., where frequently the order of summation is changed for improved performance.

The core of many numerical algorithms is the solution of systems of linear equations. The popular Krawczyk-type iteration [74] for the verified solution of dense linear systems requires computation of an approximation of the inverse R as preconditioner, and the verified computation of the residual $I - RA$.

In any case, those algorithms split into

1. a pure floating-point part and
2. a verification part.

For the first part, the fastest floating-point algorithm may be used, and there are many choices available. When just counting operations, regardless whether (floating) point or interval operations, both parts have roughly the same operation count. Therefore, the second part with the verified computation of a matrix product is the computationally intensive part. In other words, the computational speed of the verified solution of a linear system is determined by the computational speed of verified matrix multiplication. The algorithm for this multiplication can be seen in Algorithm 14.

Algorithm 14 Midpoint-Radius Matrix Multiplication in $\mathbb{F}^n$.

```
1: $\tilde{c_1} = \bigtriangledown(R \cdot mid(A))$
2: $\tilde{c_2} = \triangle(R \cdot mid(A))$
3: $\tilde{c} = \triangle(\tilde{c_1} + 0.5(\tilde{c_2} - \tilde{c_1}))$
4: $\tilde{\gamma} = \triangle(\tilde{c} - \tilde{c_1}) + |R| \cdot rad(A)$
```

2.3.3 Comparison

The two representations are identical for real intervals (not for floating-point intervals), whereas for complex intervals the first representation are rectangles and the second one represents discs in the complex plane.

Today mostly the infimum-supremum arithmetic is used. There are two main reasons for that. First, the standard definition of midpoint-radius using floating-point arithmetic causes overestimation for multiplication and division, and second, the computed midpoint of the floating-point result of an operation is, in general, not exactly representable in floating-point, thus again causing overestimation and additional computational effort. However, in [77], Rump shows that the overestimation of operations using midpoint-radius representation compared to the result of the corresponding power set operation is limited by at most a factor 1.5 in radius.

It looks like as if midpoint-radius interval arithmetic is always worse than infimum-supremum arithmetic, when both are implemented in computers. This is indeed true for the theoretical operations in $\mathbb{R}$ where no final rounding of the results is necessary. However, this changes when dealing with rounding errors.

First, the midpoint-radius representation offers the possibility to define very accurate intervals such as $\langle 1; 10^{-25} \rangle$, whereas the relative accuracy of infimum-supremum representation is restricted to the machine precision. This may occasionally be useful.

Second, the result of an operation using midpoint-radius representation in $\mathbb{F}$ may be narrower than the corresponding result of infimum-supremum arithmetic. For example, consider a 2-digit decimal arithmetic and the two intervals

$$A = \langle 1.3, 0.1 \rangle \text{ and } B = \langle 9.2, 0.1 \rangle$$

Both have exactly representable endpoints, and a corresponding (exact) infimum-supremum representation is

$$A = [1.2, 1.4] \text{ and } B = [9.1, 9.3]$$

In rounded 2-digit decimal arithmetic we obtain for the infimum-supremum representation

$$A \cdot B = [1.2, 1.4] \cdot [9.1, 9.3] \subseteq [10, 14]$$

By assumption it follows for our model arithmetic $\epsilon' = 0.05$, and Algorithm 10 for midpoint-radius representation yields the result

$$\tilde{c} = 12 \text{ and } \tilde{\gamma} = 1.7 \text{ which means } A \otimes B = \langle 12, 1.7 \rangle = [10.3, 13.7].$$

This compares to the result $[10, 14]$ in rounded infimum-supremum arithmetic. Thus, the radius of the result of midpoint-radius representation is only 85% of the radius of the result of infimum-supremum representation.

On the other hand, there are some intervals that are very difficult to represent with midpoint-radius arithmetic. One typical example is when the bounds of an infimum-supremum interval have very different orders of magnitude, e.g. $[10^{-20}, 10^{20}]$. This interval could be represented with midpoint-radius like: $\langle 0.5 \cdot 10^{20}, 0.5 \cdot 10^{20} \rangle$. However, these two intervals are not equal, as we can see if we represent this midpoint-radius interval in infimum-supremum representation: $[0.0, 10^{20}]$.

Both arithmetics, when implemented using floating-point, have advantages and disadvantages. But as presented in [77], the main point in using midpoint-radius arithmetic is that no case distinctions, switching of rounding mode in inner loops, etc. are necessary, only pure floating-point matrix multiplications. And for those the fastest algorithms available may be used, for example, BLAS. The latter bear the striking advantages that

1. they are available for almost every computer hardware, and that
2. they are individually adapted and tuned for specific hardware and compiler configurations.

This gives an advantage in computational speed which is difficult to achieve by other implementations.

2.4 Verified Computing

Verified computing provides an interval result that surely contains the correct result [55]. Verified algorithms will, in general, succeed in finding an enclosure of the solution. If the solution is not found, the algorithm will let the user know. Numerical applications providing automatic result verification, e.g. C-XSC Toolbox, may be useful in many fields like Simulation and Modeling [16], especially when accuracy is mandatory.

The program committee organizing Validated Computing 2002 [17] wrote: "Ever increasing reliance on computer systems brings ever increasing need for reliability. Validated computing is one essential technology to achieve increased software reliability. Validated computing uses controlled rounding of computer arithmetic to guarantee that hypotheses of suitable mathematical theorems are (or are not) satisfied. Mathematical rigor in the computer arithmetic, in algorithm design, and in program execution allow us to guarantee that the stated problem has (or does not have) a solution in an enclosing interval we compute. If the enclosure is narrow, we are certain that we know the answer reliably and accurately. If the enclosing interval is wide, we have a clear warning that our uncertainty is large, and a closer study is demanded. Intervals capture uncertainty in modeling and problem formulation, in model parameter estimation, in algorithm truncation, in operation round off, and in model interpretation."

One possibility to implement these ideas is using interval arithmetic. Interval arithmetic defines the operations for interval numbers, such that the result is a new interval that contains the set of all possible solutions. This ensures that the result is an enclosure, verifying the result.

To achieve a more accurate result, one can use also high accuracy arithmetic. This arithmetic ensures that the operation is performed without rounding errors, and rounded only once.

The requirements for this arithmetic are: the four basic operations with high accuracy, an optimal scalar product, and directed roundings. There are many options on how to implement this optimal scalar product [8, 54, 65]. One possibility is to accumulate intermediate results in a fixed-point number, e.g. the number is stored like an integer, and the computation will be done simulating infinite precision. This guarantees that no rounding error will happen in the accumulation. These properties must be carefully implemented through well suitable algorithms.

The use of verified computing guarantee the mathematical rigor of the result. This is the most important advantage of such algorithms compared with ordinary methods. However, finding the verified result often increases the execution times dramatically [70]. The research already developed shows that the execution times of verified algorithms are much larger than the execution times of algorithms that do not use this concept [33, 34].

2.5 Linear Systems

Many numerical problems can be solved by a dense linear system of equations. Therefore, the solution of a system with n equations and m unknowns

$$Ax = b, \tag{2.15}$$

where $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ is very usual in numerical analysis. This is true for linear equations that occur like in partial differential equations and integral equations that appear in many problems of Physics and Engineering [16]. Many different numerical algorithms contain this task as a subproblem.

The solution of linear system deals with matrices. Some of the matrix concepts that will be useful for this research are the following:

1. A matrix is said to be square if the number of rows n and columns m are the same;

2. A matrix that is filled with zeros in all positions besides the positions where $i = j$ is known as diagonal matrix;

3. A matrix with all elements of the main diagonal and above it are different from zero is named upper triangular matrix;

4. A matrix with all elements of the main diagonal and below it are different from zero is named lower triangular matrix;

5. If the elements of the main diagonal are zeros, the matrix defined in 3 and 4 are said strictly triangular matrix;

6. A matrix with determinant different from zero is named regular or not singular;

7. A matrix with almost all elements different from zero is named dense matrix;

8. A matrix that contains approximately 80% or more of zeros elements is named sparse matrix;

9. If the matrix is invertible, the equation $Ax = b$ has exactly one solution for each b.

The methods for solving linear systems of equations are divided into two classes of methods: Direct Methods and Iterative Methods.

2.5.1 Direct Methods

A method is said direct when, after a finite number of operations, the solution x is found. There are many direct methods, and they are, in general, used for solving dense matrices. Some examples are: Gauss and Gauss-Jordan method.

Gauss method is one of the most popular methods for solving dense linear systems [16]. It consists of 2 phases:

1. transform A in a triangular matrix through the use of elementary operations;

2. use backward substitution to find the solution of the system.

Gauss-Jordan elimination is a variation of Gauss method. It is considerably less efficient than Gaussian elimination with backward substitution when solving a system of linear equations. However, it is well suited for calculating the matrix inverse. It is also composed of 2 steps:

1. apply LU-decomposition

2. solve the system using backward substitution.

The LU-decomposition [16] is a method to transform the matrix A into two other matrices: L and U, where L is a lower triangular matrix and U is an upper triangular matrix. This is one of the most used methods for solving linear systems, since in this case, to solve the system, you just have to solve 2 triangular systems, which is much easier than to solve a dense system. This method is also used for finding the inverse matrix A^{-1}.

2.5.2 Iterative Methods

A method is said iterative when it is based on the repetition of a finite sequence of operations. That means, x is obtained as a limit, after several approximations $x_1, x_2, x_3 ...$.

One idea is to guess some values for x_0, and using it as a starting point, repeat a sequence of steps. After each step, the starting value is modified to one closer to the correct solution. The process has finished when the iterative process reaches sufficient accuracy, and the approximation found is very close to the exact solution. Because the stop criteria is based on the accuracy, the number of iterations is not known before the execution. This kind of method is more suitable for large problems, with a large number of variables, where A is typically sparse. Direct methods do not deal with the important advantage of sparse systems: due to the number of zeros, not all operations have to be executed.

Jacobi and Gauss-Seidel are well known iterative methods. Since sparse systems are not the focus of this work, the reader can find more details about this class of methods in [16].

2.5.3 Condition Number

The condition number of a problem is a measure of how stable is the problem, and how numerically well-posed the problem is. A problem is said ill conditioned when small modifications in the input values generate considerable errors in the final solution. The condition number is calculated as [16]

$$cond(A) = \|A\| \cdot \|A^{-1}\|, \tag{2.16}$$

where $\|A\|$ is a matrix norm. A problem with a low condition number is said to be well conditioned, while a problem with a high condition number is said to be ill conditioned.

Some important properties are:

- $cond(A) \geq 1$
- $cond(I) = 1$

This is an important measure since many methods, e.g. Gauss, do not estimate the accuracy of the solution when solved in a computer. The condition number associated with the linear equation $Ax = b$ gives a bound on how inaccurate the result x will be after finding the approximate solution [64].

The condition number is measured by its magnitude order and its quality is direct related with machine precision. If the condition of a matrix is 10^5 and the machine precision is 17, this system will find an approximation without a problem, and will have a loss of accuracy of about 5 digits. However, if the machine precision is 6, the accuracy will be very low.

2.5.4 Verified Method

A solution of the system $Ax = b$ is equivalent to finding a zero of $f(x) = Ax - b$, such that $A \in \mathbb{R}^{n \times n}$ and $b, x \in \mathbb{R}^n$. Using Newton's method, it is possible to find the fixed-point iteration [75] presented in equation 2.17:

$$x_{k+1} = x_k - A^{-1}(Ax_k - b) \tag{2.17}$$

where x_0 is an arbitrary starting value. The inverse of A is not known, so we use $R \approx A^{-1}$ as presented in 2.18:

$$x_{k+1} = x_k - R(Ax_k - b). \tag{2.18}$$

When replacing x_k for the interval $[x_k]$, the fixed-point theorem will not be satisfied, i.e. $[x_{k+1}]$ will never be a subset of $[x_k]$. For this reason, we modify the right-hand side of equation 2.18, using I as the $n \times n$ identity matrix:

$$x_{k+1} = Rb + (I - RA)x_k. \tag{2.19}$$

An approximate solution $\tilde{x}$ of $Ax = b$ may be improved if we try to enclose the error of the approximate solution. This can be done by solving (2.20) to find the residual, yielding a much higher accuracy. The error $y = x - \tilde{x}$ of the true solution x satisfies the equation

$$Ay = b - A\tilde{x} \tag{2.20}$$

which can be multiplied by R and rewritten in the form

$$y = R(b - A\tilde{x}) + (I - RA)y. \tag{2.21}$$

Let $f(y) := R(b - A\tilde{x}) + (I - RA)y$. Then Equation (2.21) has the form

$$y = f(y) \tag{2.22}$$

of a fixed point equation for the error y. If R is a sufficiently good approximation of A^{-1}, then an iteration based on Equation (2.22) can be expected to converge since $(I - RA)$ will have a small spectral radius. These results remain valid if we replace the exact expression by interval extensions. However, to avoid overestimation effects, it is recommended to evaluate it without any intermediate rounding. Therefore, we derive the following iteration from Equation (2.22), where we use interval arithmetic and intervals $[y_k]$ for y

$$[y]_{k+1} = R \diamond (b - A\tilde{x}) + \diamond(I - RA)[y]_k \ \ or \ \ [y]_{k+1} = F([y]_k), \tag{2.23}$$

where F is the interval extension of f. Here, $\diamond$ means that the succeeding operations have to be executed exactly and the result is rounded to an enclosing interval (vector or matrix). In the computation of the defect $(b - A\tilde{x})$ and of the iteration matrix $(I - RA)$, serious cancellations of leading digits must be expected. Each component must be computed exactly and then rounded to a machine interval. With $z = R(b - A\tilde{x})$ and $C = (I - RA)$ Equation 2.23 can be rewritten as

$$[y]_{k+1} = z + C[y]_k. \tag{2.24}$$

In order to prove the existence of a solution of Equation (2.20) and thus of Equation (2.15), we use Brouwer's fixed point theorem, which applies as soon as we have at some iteration index $k + 1$ an inclusion of the form

$$[y]_{k+1} = F([y]_k) \overset{\circ}{\subset} [y]_k, \tag{2.25}$$

where a $\overset{\circ}{\subset} [y]_k$ means an interior subset of $[y]_k$. If this inclusion test (Equation (2.25)) holds, then the iteration function f maps $[y]_k$ into itself. From Brouwer's fixed point theorem, it follows that f has a fixed point $y*$ which is contained in $[y]_k$ and in $[y]_{k+1}$. The requirement that $[y]_k$ is mapped into its interior ensures that this fixed point is also unique, *i.e.*, Equation (2.20) has an unique solution $y*$, and thus Equation (2.15) also has a unique solution $x* = \tilde{x} + y*$.

According to [75], if the inclusion test (Equation (2.25)) is satisfied, the spectral radius of C (and even that of $|C|$, which is the matrix of absolute values of C) is less than 1, ensuring the convergence of the iteration (also in the interval case). Furthermore, this implies also the nonsingularity of R and of A and thus the uniqueness of the fixed point.

A problem which still remains is that we do not know whether we can succeed in achieving this condition, because it may be never satisfied. To guarantee that the inner inclusion condition will be satisfied in Equation (2.25), the concept of ε-inflation is introduced, which blows up the intervals somewhat, in order to "catch" a nearby fixed point. It can be shown (*e.g.* in [76]) that Equation (2.25) will always be satisfied after a finite number of iteration steps, whenever the absolute value $|C|$ of iteration matrix C has spectral radius less than 1.

To compute the floating-point approximation of the solution $\tilde{x}$ and the floating-point approximation R of the inverse of A, we used the LU-decomposition. In principle, there is no special requirements about these quantities, we could even just guess them. However, the results of the enclosure algorithm will of course depend on the quality of the approximations. The procedure fails if the computation of an approximation of the exact solution of the inverse of A fails or if the inclusion in the interior cannot be established.

This is a brief summary of the enclosure methods theory. A more detailed presentation can be found in [75].

2.6 Parallel Computing

Many applications need high computational power and would run for days and sometimes weeks on normal personal computers or workstations. This can be a problem for applications like weather forecast: we need to know how the weather will be in 3 days, but a computer may need 7 days to finish the processing. This is a typical example where parallel processing would be essential to increase the performance. The advent of parallel computing and its impact in the overall performance of many algorithms on numerical analysis can be seen in the past years [24]. The use of clusters plays an important role in such a scenario as one of the most effective manner to improve the computational power without increasing costs to prohibitive values.

Parallel computing is the use of a parallel machine to reduce the time needed to solve a single computational problem [72]. A parallel machine is a multiple-processor computer system supporting parallel programming, i.e, one can explicitly indicate how different portions of the computation may be executed simultaneously by different processors.

Parallel programs use several processors to distribute the work of a problem aiming to reduce the global processing time. A parallel implementation must have some characteristics:

- portability: to have similar behavior in different architectures;
- scalability: the ability to gain proportionate increase in parallel speed-up with the addition of more processors.

The next section will present some considerations about parallel architectures and how it should be chosen. Section 2.6.2 shows the types of parallel programming as well as the environments for its implementation. In the last section, performance analysis will be discussed.

2.6.1 Parallel Architectures

An important classification of parallel machines is related to the memory access. Those systems can have shared memory, distributed memory, or a mixture of both. This division is based on the way that the memory and the processors are interconnected.

Shared memory computers consist of a number of processors that are connected to the same main memory. The memory connection is facilitated by a bus or a variety of switches (e.g. omega, butterfly, etc). It can be further divided into

- UMA (Uniform Memory Access) access times to all parts of memory are equal;
- NUMA (Non-Uniform Memory Access) access times to all parts of memory are not equal.

Distributed memory computers are composed of computational nodes consisting of a processor and a large local memory (there is no global memory) and are interconnected through a structured network. The memory access is called NORMA (non-remote memory access), where each of the processors can only access its own local memory and no global memory address space exists across them. In distributed memory architectures, the access to the memory is done using the message passing paradigm.

While code for distributed memory computers is relatively more difficult to write and debug, this architecture can be scaled to a large number of processors. Thus, all of the most powerful computers are currently designed in this way, however, their low-level blocks are often shared memory. For instance, the Roadrunner [82], currently the most powerful computer in the world, is a hybrid design with 12 960 IBM PowerXCell 8i CPUs and 6 480 AMD Opteron dual-core processors in specially designed server blades connected by Infiniband network connections.

Approaches to parallel computers include:

- NOW (Network of Workstations) - use workstations connected with traditional network technologies like *ethernet* and *ATM*;
- COW (Cluster of Workstations) - very similar with NOW, the difference is that this system is projected just for running parallel applications. Therefore it may use traditional networks like *ethernet* or fast network technologies like *myrinet*;
- SMP (Symmetric Multiprocessors) - use commercial processors connected with a shared memory. The most important characteristic is the uniform access to the memory;
- MPP (Massive Parallel Processing) - use a high speed network to communicate using message passing;
- DSM (Distributed Shared Memory) - very similar with SMP. The difference is that in this model, the distributed memory is shared among the processors.

2.6.2 Parallel Programming

There are two main paradigms for parallel programming:

- SPMD (Single Program Multiple Data): it is known as data parallelism. It consists of independent tasks using the same program in every node, where each node processes a part of this same program to different elements of a data set;
- MPMD (Multiple Program Multiple Data): known as functional parallelism. In this paradigm, there are independent tasks applying different operations to different data elements.

Another important aspect for implementing a parallel program is how the nodes will communicate. There are basically two possibilities: shared memory or message passing. In the first, the nodes write directly in the shared memory. It is necessary to maintain the integrity of the memory. Therefore normally semaphores and mutex are needed. Threads and OpenMP (Open Multi Processing) are often used in this approach.

Message passing is used normally in distributed memory, where each processor has its own local memory (like NOW and COW). The communication among processors is done through a network, sending and receiving messages from other nodes. Typical technologies for this implementation is PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

2.6.3 Performance Analysis

A clear difference between sequential and parallel program performance estimation is that the runtime of a parallel program depend on two variables: input size and number of processors. While the performance of a sequential program is defined by the function $T(m)$, the performance of a parallel program will be defined as $T_p(m)$, where m is the size of the problem and p is the number of processors. This time is defined as the time when the first process started to execute to the moment when the last process executed its last assignment [71].

There are some metrics to evaluate how good a parallel program is in comparison with the sequential program. The most commonly used measures are speed-up and efficiency.

Speed-up

Speed-up is a metric to compare the parallel runtime with the runtime of the best sequential program, to see if the gain of execution time is proportional to the resources invested.

That is, if T_s is the sequential runtime and T_p is the parallel runtime for p processors, then the speed-up of the parallel program is:

$$Sp_p = \frac{T_s}{T_p}. \tag{2.26}$$

There are 3 types of speed-up:

1. Absolute: this speed-up is measured using the fastest known sequential program as T_s;
2. Relative: this speed-up uses as T_s the runtime of the parallel program using just one processor;
3. Scalable: T_s is the runtime of a sequential version of the parallel algorithm (not necessarily the fastest).

For a fixed number of processors p, the speed-up value is normally between 0 and p, that means $0 < Sp_p \leq p$. If $Sp_p = p$, a program is said to have ideal or linear speed-up. This is of course rare, since most of the parallel programs will have an overhead because of synchronization and communication among the processors. Another known effect is slowdown. It results from an excessive amount of overhead.

Rarely it occurs a speed-up of more than p. This is called super linear speed-up, and is possible due to cache effects resulting from the different memory hierarchies of a modern computer. The size of accumulated caches from different processors can also change, and sometimes all data can fit into caches and the memory access time reduces dramatically, rising the speed-up drastically.

Amdahl´s Law

Suppose that a program has a fraction r, where $0 \leq r < 1$, that is parallelizable. And suppose also that this fraction has a linear speed-up. However, there is a fraction $1 - r$ that must be sequential. Under these assumptions, the speed-up of the parallelized program with p processors is

$$Sp_p = \frac{1}{(1 - r) + r/p} \tag{2.27}$$

Considering that Sp_p is an increasing function of p, and $p \to \infty$,

$$Sp_p = \frac{1}{(1 - r)} \tag{2.28}$$

If we can parallelize 90% of a program, we will have $r = 0.9$, and the maximum speed-up our problem could achieve is $Sp_p = \frac{1}{(1-0.9)}$. So if we have linear speed-up for the fraction r, according with Amdahl´s law, the maximal speed-up we could achieve is $Sp_p = 10$.

Efficiency

Efficiency is an alternative metric to speed-up. Efficiency is a measure of the process utilization in a parallel program, relative to the sequential program. It evaluates the percentage of resources that is used and can be calculated using the following equation:

$$Ef_p = \frac{Sp_p}{p} \tag{2.29}$$

where p is the number of processes and Sp_p the speed-up with p processes. Since $0 < Sp_p \leq p$, $0 < Ef_p \leq 1$. The ideal efficiency, $Ef_p = 1$, can be found when the parallelization has linear speed-up. If $Ef_p < 1/p$, then it is exhibiting slowdown. The ideal performance is achieved when the perfect balance between parallelism and communication overhead is found.

Sources of Overhead

There are three main sources of overhead:

1. Communication: each communication will contribute to grow the runtime;
2. Idle time: one or more processes remain idle while they wait for information from another process;
3. Extra computation: replicated calculations and calculations not required by the serial program.

Scalability

A parallel program is scalable if it is possible to maintain a given efficiency as the number of processors is increased by simultaneously increasing the problem size [71]. That means, as the number of processors p is increased, we find a rate of increase for the problem size m, so that the efficiency will remain constant.

3 Tools and Solvers for Dense Linear Systems

This Chapter presents a comparison among three tools and solvers for dense linear systems. An overview about each tool is presented, as well as their solvers. After that, three main points are considered in the comparison:

- Availability;
- Accuracy;
- Performance.

Based on the results of this comparison, the most appropriate sequential library for the new implementation will be chosen. The C-XSC, INTLAB and LAPACK are presented in Section 3.1, 3.2 and 3.3 respectively. Section 3.4 shows a comparison and finally Section 3.5 presents the conclusions.

3.1 C-XSC

C-XSC [44], C for eXtended Scientific Computation, is a free programming tool for the development of numerical algorithms which provide highly accurate and automatically verified results. The programming language C++, an object-oriented extension of the programming language C, does not provide better facilities than C to program numerical algorithms, but its new concept of abstract data structures (classes) and the concept of overloading operators and functions provide the possibility to create such a programming tool.

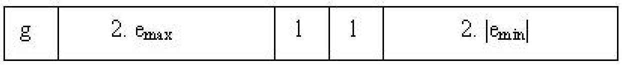
C-XSC is not an extension of the standard language, but a class library which is written in C++. Therefore, no special compiler is needed. With its abstract data structures, predefined operators and functions, C-XSC provides an interface between scientific computing and the programming language C++. Beside, C-XSC supports the programming of algorithms which automatically enclose the solution of given mathematical problems in verified bounds. Such algorithms deliver a precise mathematical statement about the true solution. The main concepts of C-XSC are:

- Real, complex, interval, and complex interval arithmetic with mathematically defined properties;
- Dynamic vectors and matrices;

- Subarrays of vectors and matrices;
- Dot-precision data type;
- Predefined arithmetic operators with highest accuracy;
- Standard functions of high accuracy;
- Dynamic multiple-precision arithmetic and standard functions;
- Rounding control for I/O data;
- Numerical results with mathematical rigor;

One of the most important features of C-XSC is the *dot-precision* data type. C-XSC simulates a fixed-point arithmetic that is essential to solve the optimal scalar product. The fixed-point accumulator provides a simulation of infinite precision according to the model of Figure 1. However, this accumulator has a defined size of 529 bytes (4227 bits), which is much larger than the data type double (8 bytes). In this figure, l is the number of digits (53 bits for each), e_{max} and e_{min} are the maximum (1023 bits for each) and the minimum (1022 bits for each) exponents, and g is the number of guard digits (31 bits).

Figure 1 – Fixed-Point Accumulator.



3.1.1 Solver

The C-XSC verified solver for dense linear systems of equations is based on the verified method presented in 2.5.4, and is fully described in [31]. It will, in general, succeed in finding and enclosing a solution or, if it does not succeed, will let the user know. In the latter case, the user will know that the problem is likely to be very ill conditioned or that the matrix A is singular. In this case, the user can try to use higher precision arithmetic.

The first parallel approach uses this solver as starting point. In Chapter 4, we present the mathematical issues behind this implementation.

Algorithm 15 Enclosure of a Square Linear System Using C-XSC.

```
 1: $R \approx A^{-1}$ {compute an approximation of the inverse using LU-Decomposition algorithm}
 2: $\tilde{x} \approx R \cdot b$ {compute the approximation of the solution}
 3: $[z] \supseteq R(b - A\tilde{x})$ {compute enclosure for the residuum}
 4: $[C] \supseteq (I - RA)$ {compute enclosure for the iteration matrix}
 5: $[w] := [z]$, $k := 0$ {initialize machine interval vector}
 6: **while not** ($[w] \subseteq int[y]$ or $k > 10$) **do**
 7:     $[y] := [w]$
 8:     $[w] := [z] + [C][y]$
 9:     $k++$
10: **end while**
11: **if** $[w]\overset{\circ}{\subset}[y]$ **then**
12:     $\Sigma(A, b) \subseteq \tilde{x} + [w]$ {The solution set ($\Sigma$) is contained in the solution found by the method}
13: **else**
14:     no verification
15: **end if**
```

3.2 MATLAB and INTLAB

INTLAB [77, 78] is a well designed interval toolbox for the interactive programming environment MATLAB [63]. It allows the more traditional infimum-supremum as well as the midpoint-radius representations of intervals.

The midpoint-radius interval arithmetic of INTLAB is entirely based on BLAS. Matrix and vector operations avoid in a clever way case distinctions and the time consuming switching of rounding mode in inner loops at the expense of some additional BLAS operations. So, in particular, INTLAB matrix operations are very fast.

Every computation using INTLAB is rigorously verified to be correct, including input and output. Portability is assured by implementing all algorithms in MATLAB itself. For some architectures and old versions of Matlab (5.3 or earlier) one routine for switching the rounding mode (downwards, upwards and to nearest) is necessary. However, rounding is already integral part of Matlab 5.3 and following under Windows (in this research, tests were done using Matlab 7.2). INTLAB itself may be freely copied from the home page [78], but the commercial product MATLAB [63] must be bought to be able to use INTLAB. INTLAB is a very powerful interactive tool to implement prototypes of verification algorithms and allows the algorithms to be written in a way which is very near to pseudo-code used in scientific publications.

3.2.1 Verifylss.m

INTLAB offers predefined problem solving routines for dense and sparse systems of linear and nonlinear equations and eigenvalue problems (verifylss, verifynlss, verifyeig). For this work, we are interested in the routine *verifylss* as shown in Algorithm 16.

Algorithm 16 Enclosure of a Square Linear System Using INTLAB.

```
1: $R \approx A^{-1}$ {Compute an approximation of the inverse using LU-Decomposition algorithm}
2: $xs = R \cdot midb$ {compute the approximation of the solution}
3: while $i < 15$ or $norm < 0.1 \cdot oldnorm$ do
4:    $res = R \cdot dot(-1, midb, midA, xs)$
5:    $xs = xs - res$
6: end while
7: if A is an interval then
8:    $ires = b - A \cdot xs$
9:    $z = R \cdot ires$
10:   $RA = R \cdot A$
11: else
12:   if b is an interval then
13:      if $all(rad(b) == 0)$ then
14:         $ires = dot(1, b.inf, A, -xs, -2)$
15:      else
16:         $ires = lssresidual(A, xs, intval(b))$
17:      end if
18:   else
19:      $ires = dot(1, b, A, -xs, -2)$
20:   end if
21:   $z = R \cdot ires$
22:   $RA = R \cdot intval(A)$
23: end if
24: $y = z$
25: $k = 0$
26: $ready = 0$
27: $e = 0.1 \cdot rad(y) \cdot hull(-1, 1) + midrad(0, 10 \cdot realmin)$
28: while not $([w] \subseteq int[y]$ or $k > 7)$ do
29:   $k = k + 1$
30:   $x = y + e$
31:   $y = z + C \cdot x$
32:   $ready = all(all(in0(y, x)))$
33: end while
34: if $ready$ then
35:   $x = xs + y$
36: else
37:   no verification
38: end if
```

As can be seen, this algorithm has a big similarity with the one implemented in C-XSC. Both use as start an approximation of the inverse R and use the Krawcyzk operator as iteration to find the enclosure.

3.3 LAPACK

LAPACK [57] is a FORTRAN 77 library for numerical linear algebra, with versions available in C and other languages. This package includes numerical algorithms for the more common linear algebra problems in scientific computing. It is based on LINPACK and EISPACK, libraries for solving linear equations, linear least squares, and eigenvalue problems for dense and banded systems.

The numerical algorithms in LAPACK are based on BLAS (Basic Linear Algebra Subprograms [20]) routines. It utilizes block-matrix operations, such as matrix-multiply in inner loops to achieve high performance. These operations improve the performance by increasing the granularity of the computations and keeping the most frequently accessed subregions of a matrix in the fastest level of memory [21].

3.3.1 Solver

The approximate solution of a linear system in LAPACK can be computed by the following steps:

1. DGETRF: factorize the general matrix A in two triangular matrices $A = L \cdot U$ (obviously not needed for triangular matrices);
2. DGETRS: use the factorization (or the matrix A itself if it is triangular) to solve the system by forward or backward substitution;
3. DGECON: estimate the reciprocal of the condition number;
4. DGERFS: compute bounds on the error in the computed solution (returned by the DGETRS routine), and refine the solution to reduce the backward error (see below);
5. DGETRI: use the factorization (or the matrix A itself if it is triangular) to compute A^{-1} (not provided for band matrices, because the inverse does not in general preserve bandedness).

All these routines are optimized and based on BLAS routines.

3.4 Comparison

Each tool has its advantages and disadvantages. As can be seen in the next sections, some are more accurate and some are faster. Another important aspect is the availability of these tools. For these tests, we used the following configuration:

- AMD Athlon 64 X2 dual core proc 4200+;
- Suse linux 10.0 64 bit;
- gcc/g++;
- CXSC release 2.1.1 hardware arithmetic;
- MATLAB version 7.2.0.283 (R2006a) and INTLAB version 5.3;
- CLAPACK (LAPACK in C).

3.4.1 Availability

Both C-XSC and LAPACK are free libraries and can be used for educational and scientific purposes. They can be downloaded from internet and used without any restrictions.

INTLAB, otherwise, cannot be used freely. INTLAB itself has an open source, but to be able to use INTLAB you have to buy the commercial product MATLAB [63]. This is a big disadvantage which can make the costs of using INTLAB very large.

3.4.2 Accuracy

Aiming to verify the accuracy of these 3 tools, we used an ill conditioned matrix generated by the well known Boothroyd/Dekker formula:

$$A_{ij} = \begin{pmatrix} \text{n+i-1} \\ \text{i-1} \end{pmatrix} \times \begin{pmatrix} \text{n-1} \\ \text{n-j} \end{pmatrix} \times \tfrac{n}{i+j-1}, \forall i, j = 1..n,$$
$$b = 1, 2, 3, ..., n,$$

which has a condition number $1.1 \cdot 10^{15}$, to see how these tools would react. Let

$$A = \begin{pmatrix} 10 & 45 & 120 & 210 & 252 & 210 & 120 & 45 & 10 & 1 \\ 55 & 330 & 990 & 1848 & 2310 & 1980 & 1155 & 440 & 99 & 10 \\ 220 & 1485 & 4752 & 9240 & 11880 & 10395 & 6160 & 2376 & 540 & 55 \\ 715 & 5148 & 17160 & 34320 & 45045 & 40040 & 24024 & 9360 & 2145 & 220 \\ 2002 & 15015 & 51480 & 105105 & 140140 & 126126 & 76440 & 30030 & 6930 & 715 \\ 5005 & 38610 & 135135 & 280280 & 378378 & 343980 & 210210 & 83160 & 19305 & 2002 \\ 11440 & 90090 & 320320 & 672672 & 917280 & 840840 & 517440 & 205920 & 48048 & 5005 \\ 24310 & 194480 & 700128 & 1485120 & 2042040 & 1884960 & 1166880 & 466752 & 109395 & 11440 \\ 48620 & 393822 & 1432080 & 3063060 & 4241160 & 3938220 & 2450448 & 984555 & 231660 & 24310 \\ 92378 & 755820 & 2771340 & 5969040 & 8314020 & 7759752 & 4849845 & 1956240 & 461890 & 48620 \end{pmatrix}$$

$$\text{and } b = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

The result found by each tool is presented in Table 3.4.2. The " ________ " in the INTLAB result symbolize the digits with uncertainty.

Table 2 – Solution of Equation (6.1) by the Numerical Tools.

Naive floating-point approximation	C-XSC verified solution
3.769740075654227E-010	[0.000000000000000E+000, 0.000000000000000E+000]
9.999999949022040E-001	[1.000000000000000E+000, 1.000000000000000E+000]
-1.999999975192608E+000	[-2.000000000000000E+000,-2.000000000000000E+000]
2.999999904595484E+000	[3.000000000000000E+000, 3.000000000000000E+000]
-3.999999691503490E+000	[-4.000000000000000E+000,-4.000000000000000E+000]
4.999999132005541E+000	[5.000000000000000E+000, 5.000000000000000E+000]
-5.999997815384631E+000	[-6.000000000000000E+000,-6.000000000000000E+000]
6.999994975405571E+000	[7.000000000000000E+000, 7.000000000000000E+000]
-7.999989266972989E+000	[-8.000000000000000E+000,-8.000000000000000E+000]
8.999978443156579E+000	[9.000000000000000E+000, 9.000000000000000E+000]
INTLAB verified result	LAPACK approximation
0,000000___________	-0.000000
1,00000___________	1.000001
-2,00000___________	-2.000003
3,0000___________	3.000013
-4,0000___________	-4.000041
5,000___________	5.000113
-6,000___________	-6.000279
7,00___________	7.000631
-8,00___________	-8.001326
9,00___________	9.002628

C-XSC and INTLAB present an interval as result. i.e. an enclosure of the exact result. C-XSC presented a point interval, while INTLAB present some uncertainty digits. This problem also shows how critical the instability of a problem can be.

The verified result makes possible to evaluate how good the floating-point approximation is. We can now determine that the LAPACK floating-point approximation has 3 to 6 correct digits. This shows how important it is to prevent these errors through verified computing.

3.4.3 Performance

Table 3 shows the execution time in seconds for each tool, for 4 matrix dimensions (N) and different repetition factors (RF).

Table 3 – Performance of Numerical Tools.

	LAPACK	C-XSC	INTLAB
N = 10 RF = 10.000	0.34	6.95	51.59
N = 100 RF = 10	0.03	4.25	0.276
N = 500 RF = 1	0.24	83.53	0.86
N = 1000 RF = 1	1.04	541.24	3.57

As expected, LAPACK presented the best performance in terms of execution time. INTLAB is based on BLAS, therefore it presents also a good performance compared with C-XSC. The performance presented by C-XSC is not so optimal because the algorithm uses special variables (dotprecision variables), which are simulated in software to achieve high accuracy.

3.5 Conclusions

Each of these libraries has a limitation in terms of availability, accuracy or performance. Table 4 resumes the results of the comparison.

Table 4 – Comparison among C-XSC, INTLAB and LAPACK.

	C-XSC	LAPACK	INTLAB
available	yes	yes	no
accurate	yes	no	yes
high performance	no	yes	yes

Since C-XSC has the free and verified solver, and also presented the more accurate result, it was used in the first parallel approach (see Chapter 4).

4 First Parallel Approach using C-XSC

This Chapter presents the first parallel experiments. To ensure that the mathematical properties of interval arithmetic as well as high accuracy arithmetic will be achieved, the C-XSC library was used.

To implement a first parallel version of Algorithm 15, we used an approach for cluster architectures with message passing programming model (MPI). Clusters of computers are considered a good option to achieve better performances without using parallel programming models oriented to very expensive (but not frequently used) machines. A parallel version for this algorithm runs on distributed processors, requiring communication among the processors connected by a fast network and the communication library.

As mentioned in section 3.4, C-XSC is an attractive option among the available tools to achieve verified results. On the other hand, the data types of the C-XSC library, which allow for self-verification and were used on the parallel approach, do not fit in the MPI library data types. In order to solve this problem, the data should be packed and unpacked before sending and receiving. Aiming at enabling this communication, we used a library that extends MPI for the use of C-XSC in parallel environments implementing the pack and unpack procedures in a low level [29].

4.1 Implementation Issues

We carried out some tests to find the most time-consuming steps in Algorithm 15. We found out that the steps 1 and 4 consume together more than 90% of the total processing time. The computation of an approximation of the inverse matrix of A (matrix R on step 1) takes more than 50% of the total time, due to the evaluation with high accuracy. The computation of the interval matrix $[C]$ (parallel preconditioning) takes more than 40% of the total time, since matrix multiplication requires O(n^3) execution time, and the other operations are mostly vector or matrix-vector operations which require at most O(n^2). We chose to optimize these two parts of the algorithm, using two different approaches that will be presented in the following sections.

We assume the coefficient matrix A in Equation (2.15) to be dense, i.e. in a C-XSC program, we use a square matrix of type rmatrix to store A, and we do not consider any special structure of the elements of A. Our goal is to make a parallel version of the C-XSC algorithm that verifies the existence of a solution and computes an enclosure for the solution of the system $Ax = b$ for a square $n \times n$ matrix A with a better performance than the sequential version.

4.1.1 The Inverse Matrix Computation

We may have two possibilities to achieve a better performance at the computation of R (the floating-point approximation of the inverse matrix of A). The first one is to parallelize it keeping the high accuracy in its evaluation. It will achieve a better performance, and be sure to evaluate an optimal approximation of the inverse of A.

Another possible optimization is to evaluate R without high accuracy. Since R is an approximation of the inversion of the matrix A, which does not always need exact results, in several cases it can be computed without high accuracy. In these cases, the approximation obtained with floating-point arithmetic is enough to find an accurate inclusion at the end of the algorithm. It is also known that the elimination of the use of high accuracy on R computation could decrease the overall executing time and in most cases would not compromise the results obtained.

On the other hand, in some cases the mathematical high accuracy is needed to find the correct result. In these cases, the non-accurate approximation R would not be enough, and the algorithm would not find the correct result for the linear system. Therefore, we made a parallel version of the computation of the approximation of the inverse matrix keeping the high accuracy.

Due to this condition, we have implemented the algorithm as follows:

- If the algorithm is able to find an enclosure of the solution of the linear system based on the computation of R without high accuracy, it stops;
- If the algorithm does not find the solution, it executes again evaluating R with high accuracy.

The approach used in the parallel implementation is explained in the next section.

Parallel Inverse Matrix Computation

A deeper analysis of the execution time spent in step 1 was carried out to find the bottleneck in this part of the algorithm. The computation of the inverse matrix is done in two major steps: the LU decomposition and the computation of the inverse column by column through backward/forward substitution. From the 52% of the total time that this step takes, the LU decomposition of A takes 35%, whereas the calculation of the inverse by backward/forward substitution takes 65% of step 1, due to C-XSC's special arithmetic and features like dotprecision variables. Through this analysis, it was decided that the parallelization of R computation would be focused only on the backward/forward substitution phase due to the higher computational cost. Moreover, the parallel LU decomposition is well known and many different proposals can be found [41, 59, 80, 84].

The algorithm implemented uses a parallel phases scheme to find the approximation of the inverse matrix (R) through backward/ forward substitution. Every processor computes a number of columns (co-named task) of matrix R and after this processing phase, all processes exchange information to construct the overall R. At the end of this process, all nodes must have the inverse matrix R, once R is used in the further parallel computation of the interval matrix C. Figure 2 (a) shows the communication strategy used in this step.

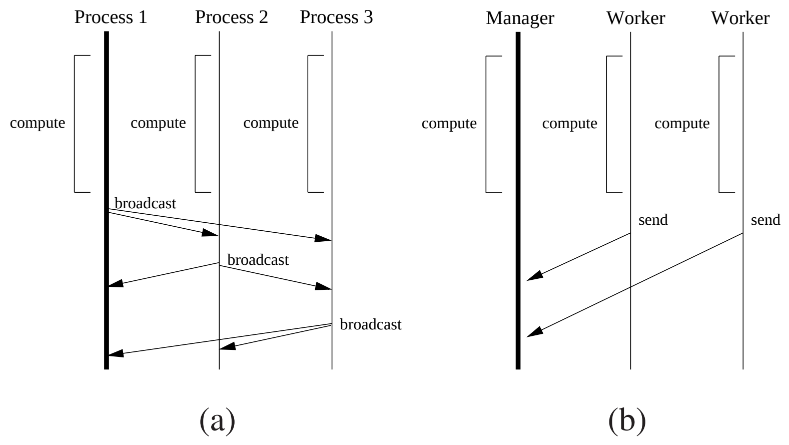


Figure 2 – Communication Schemes.

In section 4.2, we will present detailed results for the execution time concerning the computation of R with and without maximal accuracy.

4.1.2 Parallel Preconditioning

Preconditioning is known as a way to speed-up the convergence of a method based on establishing some algebraic constants used to approximate x at each iteration. The preconditioning in Algorithm 15 can be seen in step 4. In the computation of $[C] = (I - RA)$, I is the identity matrix of the same order as A, and R is an approximation of A^{-1}. This step was implemented in parallel using the worker/manager approach.

In this approach, each process finds a contiguous block of rows based on its rank, to compute some rows of the result matrix ($[C]$). After computing, the manager receives the computed lines from every process, puts them in the proper places and generates $[C]$. This step is illustrated by Figure 2 (b), where the thicker vertical line means the manager process, and the other vertical lines mean the other processes. After computing, all the worker processes send their result to the manager.

4.1.3 Load Balancing Approach

The algorithm representing the load balancing strategy for both parallelizations is shown in Algorithm 17. Aiming to decrease the overhead, the load balancing was done in parallel. Since the processes do not have to be concerned with load balancing, this approach takes less communication time to exchange information. Another important characteristic of this algorithm is that it should be simple but effective, and above everything, fast. It should not become a bottleneck in the computation process.

Algorithm 17 Workload Distribution Algorithm.

```
1: for ($pr = 0, i = 1; pr < P; pr++, i++$) do
2:     if $pr < (N\%P)$ then
3:         $lb_i = (\frac{N}{P} \times pr) + pr$
4:         $ub_i = lb_i + \frac{N}{P}$
5:     else
6:         $lb_i = (\frac{N}{P} \times pr) + (N\%P)$
7:         $ub_i = lb_i + \frac{N}{P} - 1$
8:     end if
9: end for
```

Algorithm 17 presents the workload distribution. The identification of a process goes from 0 to $P - 1$ and is called pr, where P is the number of processes involved in the computation and N is the number of rows or columns of a matrix. Also, ub_i and lb_i are the upper and lower bound of the i^{th} process respectively, i.e. the first and the last row/column that a process must compute.

Aiming to decrease the communication cost, the process will receive a continuous block of rows/columns, what can be ensured by this load balancing approach. Since the computational cost to evaluate a row/column of the matrix is the same, other strategies for load balancing could be used.

4.2 Experimental Results

In order to guarantee the success of these optimizations, two different experiments have been performed. The first test is related to the correctness of the result. Once we make a mathematical optimization, we need to verify that it did not change the accuracy of the result. Another test should be done to evaluate the speed-up achieved in this new implementation. Since the main objective is to minimize the execution time, achieving good speed-ups is the main goal.

Some test cases were executed varying the number of processes and the size of the input matrix A and vector b. To evaluate the correctness and the performance of the new implementation, matrix A and vector b were generated in two different ways: the Boothroyd/Dekker formula [31] and random numbers, respectively.

The results presented in this work were obtained over the parallel environment ALiCEnext (Advanced Linux Cluster Engine, next generation) installed at the University of Wuppertal (Germany). This cluster is composed of 1024 1.8 GHz AMD Opteron processors (64 bit architecture) on 512 nodes connected by Gigabit Ethernet. ALiCEnext processors employ Linux as operating system and MPI as communication interface for parallel communication. These experiments are presented in the following sections.

4.2.1 Accuracy

Aiming to verify the accuracy of our parallel solution, many executions were tackled with several matrices. Among them the well-known Boothroyd/Dekker matrices (already defined in Chapter 3, Section $3.4.2$).

We ran tests with a 10×10 matrix with different numbers of processes $(1..P)$. The tests generated by the Boothroyd/Dekker formula presented the same result for both versions (sequential and parallel) and indicated that the parallelization did not modify the accuracy of the results.

4.2.2 Performance

Performance analysis of this parallel solver was carried out varying the order of input matrix A. Matrices with five different orders were used as test cases: 500×500, $1\,000 \times 1\,000$, $2\,500 \times 2\,500$, $3\,000 \times 3\,000$, and $4\,500 \times 4\,500$. For each of those matrices, executions with the number of processors varying from 1 to P were performed. In the experiments presented in the next paragraphs, three different values were assigned to P: 18, 28, and 100. All matrices were specifically generated for these experiments and are composed of random numbers. Since the greater part of systems of equations can be solved evaluating R with floating-point precision, the experiments that generated the results presented in Figure 3 and Figure 4 **did not use high accuracy to compute** R (the floating-point approximation of the inverse of A).

Figure 3 presents a comparison of the scalable speed-ups achieved for small matrices (order 500, $1\,000$, and $2\,500$). The proposed parallel solution presents a good scalability and improves the performance of the application. As expected, the performance gain is proportional to the matrix order (which impacts directly on its computational cost): the larger the input matrix, the better is the scalable speed-up achieved. It is also important to remark that for the input matrix with order $2\,500$, the scalable speed-up achieved for the parallel part of the algorithm was around 21 for 28 processors, which is a representative result for the chosen platform. Results for more than 28 processors were not presented for these test cases since beyond this threshold

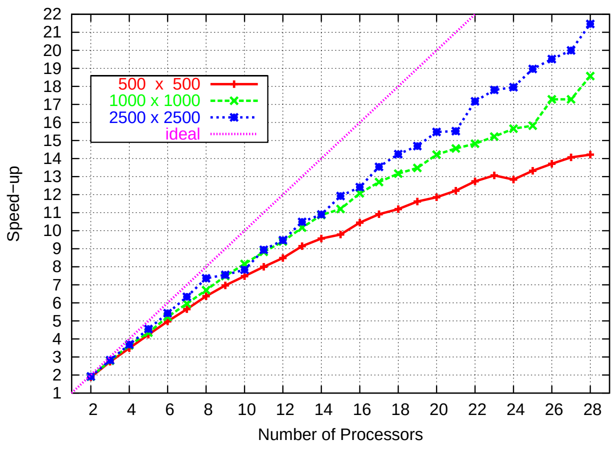


Figure 3 – Experimental Results - Speed-up.

the performance started to drop down.

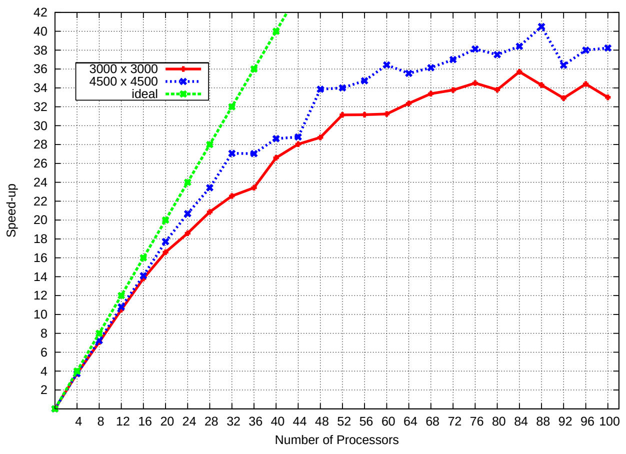


Figure 4 – Experimental Results - Speed-up for Large Matrices.

In Figure 4, results for larger input matrices are shown. In this experiment, the goal was to verify the behavior, in terms of performance gains, of the proposed parallel solution using a higher number of processors (*e.g.* one hundred). Parallel solutions developed for multi-computers, such as clusters based on the message passing paradigm, are usually not able to scale properly due to communication issues. Indeed, the curves show a loss of efficiency at different points for all three input cases, even considering that the proposed parallel solution decreases the execution times up to 92 or 96 processors. However, the best scalable speed-up factor achieved for each test case represents a significant gain of performance in the computation of those matrices. The oscillation effect presented in this figure could be due to hardware instability. The cluster had occasionally problems with some nodes, and some tests were executed just once.

Considering that in some cases the computation of R with high accuracy is needed, the results of its parallel version are presented in Figure 5. This figure presents a comparison of the total computation time needed using i) the parallel and highly accurate computation of R and

ii) the implementation that computes R in serial without high accuracy. The matrix size used in this test was $1\,000 \times 1\,000$ and only 18 processors were used.

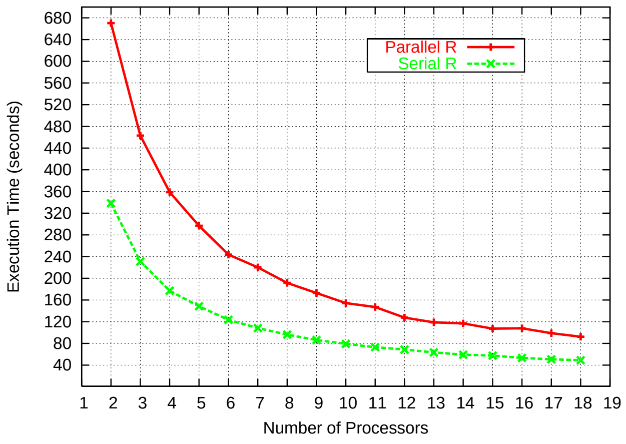


Figure 5 – Experimental Results - Execution Time.

Table 5 summarizes the execution time, scalable speed-up, and efficiency achieved by both implementations using a test matrix with dimension $1\,000$. Comparing the sequential time with the execution time for 18 processors, it is possible to notice that a significant decrease is achieved. In this scenario, the load balancing strategy adopted is confirmed to be a good choice, since the efficiencies presented vary from 72% up to 95%. Table 5 (a) presents the results when the computation of R does not use high accuracy. In the test case presented in Table 5 (b), where the highly accurate computation of R was parallelized, the efficiencies are slightly worse than for the test case (a).

Table 5 – Experimental Results - Efficiency.

# proc	time (sec)	Speed-up	Efficiency
1	649.46	1	100 %
2	340.87	1.90	95 %
3	235.82	2.75	91 %
4	180.52	3.59	89 %
5	147.11	4.41	88 %
6	122.92	5.28	88 %
7	106.58	6.09	87 %
8	95.07	6.83	85 %
9	84.86	7.65	85 %
10	78.88	8.23	82 %
11	70.76	9.17	83 %
12	67.62	9.60	80 %
13	61.33	10.58	81 %
14	57.81	11.23	80 %
15	54.69	11.87	79 %
16	52.56	12.35	77 %
17	49.57	13.10	77 %
18	47.47	13.68	76 %

(a)

# proc	time (sec)	Speed-up	Efficiency
1	1201.70	1	100 %
2	670.36	1.79	89 %
3	462.99	2.59	86 %
4	358.84	3.34	83 %
5	296.90	4.04	80 %
6	243.65	4.93	82 %
7	220.14	5.45	77 %
8	191.51	6.27	78 %
9	172.91	6.94	77 %
10	154.34	7.78	77 %
11	147.09	8.16	74 %
12	127.48	9.42	78 %
13	118.86	10.10	77 %
14	116.85	10.28	73 %
15	107.20	11.20	74 %
16	107.75	11.15	69 %
17	98.86	12.15	71 %
18	92.29	13.01	72 %

(b)

4.3 Conclusions

A parallel implementation for the self-verified method for solving dense linear systems of equations was presented in this Chapter. The self-verification provides reliability but also decreases the performance. Therefore two main parts of this method, which demand a higher computational cost, were studied, parallelized and optimized. The computation of the approximative inverse of A has been optimized in two different approaches: mathematically and with the use of parallel concepts. The preconditioning step was also parallelized to achieve a better performance.

Two different types of input matrices with five different sizes were used in several experiments aiming to evaluate the scalable speed-up achieved in this new implementation. All five granularities increased the performance with significant gain. The accuracy tests also point out a good implementation, where the results were obtained without any loss of accuracy. Thus, the gains provided by the self-verified computation could be kept with a significant decrease in the execution time through its parallelization. The load balancing strategy seems to be a good choice according to the results found in all tested input cases.

It is possible to notice rather interesting scalable speed-ups for the self-verified computation, reinforcing the statement related to the good parallelization choices. In the other hand, only around 75% of this implementation was parallelized. The scalable speed-up achieved for the parallel part was very promising, however according to Amdahl's law, there is a maximum theoretical speed-up for the whole implementation of 4, no matter how many processors we used.

The implementation allowed a quite good understanding of the problem, leading to new directions for further investigations. Based on the experience of this implementation, a new solver will be proposed in Chapter 5.

5 New Sequential Solver

This Chapter presents the approach used in the new sequential solver. The methods and tools used in this implementation were chosen aiming to find a compromise between these two important characteristics: performance and reliability. The algorithms are strongly based on C-XSC method with some modifications concerning the switching of rounding mode and optimizations of the residuum.

Two sequential versions were implemented, one using infimum-supremum arithmetic and another using midpoint-radius arithmetic. As explained in Section 2.3, these representations are very different, and its properties will affect the implementation decisions.

The use of directed roundings was carefully studied, so that we can avoid it in some cases to reduce the computation time. BLAS and LAPACK routines were also used to increase the performance.

Linear systems can be composed by point or interval elements. The use of interval input data is very important for many reasons, among others the handling of the uncertainty of input data. However, to solve an interval linear system, some different aspects must be considered.

Section 5.1 defines how direct roundings will be implemented in both infimum-supremum and midpoint-radius implementation. Section 5.2 discusses the implementation for point input data, first using infimum-supremum then using midpoint-radius. Section 5.3 shows the necessary changes in the algorithm to make it able to accept intervals as input data, and finally, Section 5.4 presents the conclusions of this Chapter.

5.1 Directed Roundings

As explained in Section 2.3.2, the switching of rounding modes is a very expensive operation which may require up to 10 times as much execution time as a floating-point operation and therefore, the rounding mode should be switched as infrequently as possible [10].

This is specially important when implementing infimum-supremum arithmetic in computers since during its elementary operations, it needs to change the rounding mode many times. This effect can be even more problematic if dealing with vector and matrix interval operations, because each of these matrix operations is composed of many basic operations.

Therefore, the idea of avoiding the switch of rounding mode was implemented in the sequential algorithms that use infimum-supremum arithmetic. The rounding was switched to rounding down during all the algorithm, and the operations were modified to ensure an enclosure even with just one rounding direction.

For the midpoint-radius arithmetic, however, this procedure was not necessary. Since there are no case distinctions in its basic operations, the matrix and vector operations can be done in two groups: first with rounding up, and then with rounding down, reducing the number of switching the rounding mode to only two, even for very large matrices.

5.2 Point Input Data

The new algorithm is essentially based on the algorithm used by C-XSC method. However, since C-XSC is not used, other strategies have to be used to guarantee the verification of the result. As explained before, to have a verified solution, it should be an enclosure. Therefore interval arithmetic operations were implemented and primitives to switch the rounding mode were used to ensure that the correct result will be inside the interval result.

Aiming at a better performance, BLAS and LAPACK routines were used in critical parts of the algorithm. The idea was to combine BLAS and LAPACK with interval arithmetic and directed roundings to ensure the verification and make it faster.

Algorithm 18 shows the complete algorithm used for this approach; $int[y]$ is used to denote the inner inclusion relation, meaning that the left hand side is contained in the interior of $[y]$. This algorithm was used for both infimum-supremum and midpoint radius implementation.

This method can be divided in the following 5 main steps. For each step, a different approach was used, but always ensuring the verification, optimizing the quality of the result and aiming at a good performance. The steps are:

1. LU decomposition and approximation of the inverse: since these steps aim to find an approximation, they do not need to use interval arithmetic and directed rounding. To achieve a very fast result, LAPACK routines were used:
   - *dgetrf*: for the LU decomposition;
   - *dgetri*: to find the approximation of the inverse matrix.
2. Calculation of an approximation of $x = R \cdot b$: this step uses the BLAS *dgemv* routine to find an approximation;
   - Residual of x: this sub-step was added to the original method. It is based on the INTLAB routine *verifylss*. It tries to optimize the result found for x based on the norm. For this purpose, the BLAS routine *dgemv* was used to execute the vector-matrix multiplication, aiming at a very fast execution.

Algorithm 18 Enclosure of a Point Linear System Using LAPACK, BLAS.

```
1: {STEP 1}
2: $R \approx A^{-1}$ {compute an approximation of the inverse using the LAPACK: dgetrf_ and dgetri_}
3: {STEP 2}
4: $\tilde{x} \approx R \cdot b$ {compute the approximation of the solution using BLAS: dgemv_}
5: $res = 0;$
6: while $cont < 15$ or $norm < 10^{-1} \cdot oldnorm$ do
7:     $cont++;$
8:     $oldnorm = norm;$
9:     for $(i = 0; i < SIZE; i++)$ do
10:        $norm = norm + abs(res);$
11:    end for
12:    $res = R \cdot (A \cdot x + b)$ {using BLAS: dgemv_}
13:    if $norm < normold$ then
14:        $x = x - res$
15:    end if
16: end while
17: {STEP 3}
18: $[z] \supseteq R(b - A\tilde{x})$ {enclosure for the residuum using BLAS: dgemv_ and directed roundings}
19: {STEP 4}
20: $[C] \supseteq (I - RA)$ {enclosure for the iteration matrix using directed roundings and BLAS method: gdemm_}
21: {STEP 5}
22: $[w] := [z]$, $k := 0$ {initialize machine interval vector}
23: while not $([w] \subseteq int[y]$ or $k > 10)$ do
24:    $[y] := [w]$ {using interval arithmetic}
25:    $[w] := [z] + [C][y]$ {using interval arithmetic}
26:    $k++$
27: end while
28: if $[w] \subseteq int[y]$ then
29:    $\Sigma(A, b) \subseteq \tilde{x} + [w]$ {using directed roundings} {the solution set ($\Sigma$) is contained in the solution found by the method}
30: else
31:    no verification
32: end if
```

3. Calculation of $z = R(b - A \cdot x)$: this step uses the BLAS *dgemv* routine but also directed rounding to find two bounds for z.

4. Calculation of $C = (I - R \cdot A)$: this is one of the most important steps, since it is the one that has the biggest complexity ($O(n^3)$) and needs more execution time. Therefore, it is very important to implement it in a clever way. The BLAS routine *dgemm* for matrix multiplication was used twice to find the lower and upper bound of C;

5. Verification step: it uses basically interval arithmetic and directed roundings. Since at this point all variables involved in the calculation are intervals, an interval arithmetic has

to be used. Both infimum-supremum and midpoint-radius arithmetics were implemented in the present work using the programming language C.

5.2.1 Infimum-Supremum Approach

This algorithm is based on the infimum-supremum interval arithmetic. The operation and properties were implemented as explained in section 2.3.

The main difference between the infimum-supremum and midpoint radius arithmetic is that for the infimum-supremum we used the idea of Bohlender [9] to avoid the switching of rounding modes. The rounding mode is switched to rounding down in the beginning, and the operations are a little bit changed so that no other rounding mode is necessary. In this way, we use just one rounding mode during all algorithm.

Avoiding the switching of rounding modes is a little more complicated if we are dealing with an expression. If we have more than one operation to evaluate, the operations must be carefully changed taking into account possible negations that would invert the rounding mode as explained in section 2.3.2. The calculation of z, for exemple, involves 3 operations. Algorithm 19 shows the calculation and rounding approach used to guarantee a lower and upper bound for z.

Another important remark is that since infimum-supremum arithmetic operations need case distinctions, vector and matrix operations cannot use BLAS routines internally. That means these operations were implemented in C with loops and conditions.

Algorithm 19 Calculation of z.

```
1: $setround = down$
2: $aux1 = (b + A \cdot (-x))$
3: $aux2 = (-(-b + A \cdot x))$
4: for $(i = 0; i < SIZE; i++)$ do
5:     $accuu = 0.0$
6:     $accud = 0.0$
7:     for $(j = 0; j < SIZE; j++)$ do
8:         if $(r[i,j] > 0.0)$ then
9:             $accud = accud + (r[i,j] * aux2[j])$
10:            $accuu = accuu - (r[i,j] * aux1[j])$
11:        else
12:            $accuu = accuu - (r[i,j] * aux2[j])$
13:            $accud = accud + (r[i,j] * aux1[j])$
14:        end if
15:    end for
16:    $zd[i] = accud;$
17:    $zu[i] = -accuu;$
18: end for
```

5.2.2 Midpoint-Radius Approach

In this case, the interval arithmetic used in the steps 3, 4 and 5 is the midpoint-radius arithmetic. Its operations and properties were implemented in the programming language C as defined in section 2.3.

The idea of avoiding the switching of rounding mode is not implemented for this arithmetic. Since we have no case distinctions in this arithmetic, we can switch the rounding mode before a matrix operation, do all the basic operations involved in this matrix operation, and change the rounding back. That means that for midpoint-radius arithmetic the switching of rounding mode is only a few times necessary.

The major difference between this approach and the infimum-supremum approach can be seen when dealing with interval matrix and vector multiplication. In Step 5, the operation $[w] := [z] + [C][y]$ needs to be evaluated. Using midpoint-radius, it is possible to implement it using the optimized BLAS routine *dgemv*. On the other hand, for infimum-supremum, this operation has to be programmed with loops and case distinctions, which will lead to a much lower performance. This difference plays a bigger role when using interval input data. It will be shown in Section 5.3.2.

5.2.3 Experimental Results for Point Input Data

The following tests were done in a processor of the cluster named XC1, composed of Intel Itanium2 processors with a frequency of 1.5 GHz, with 12 GB memory and 6 MB of level 3 cache. The basic operating system is HP XC Linux for High Performance Computing (HPC) and the compiler used was gcc Version 3.4.6.

The tools used to implement the algorithms and to compare the results are the following:

- C-XSC release 2.1.1.
  - Using hardware arithmetic;
  - During the writing of this thesis, a new version of the C-XSC solver for dense linear systems of equations was developed, applying these ideas to improve its performance. In Chapter 9, some considerations will be done concerning this new implementation, but the comparisons presented in these experiments use the C-XSC solver implementation presented in [31].
- MATLAB version 7.2.0.283;
  - INTLAB version 5.3.
- New Algorithm:

- Programming language C;
- MKL 10.0.011 in XC1 for BLAS and LAPACK.

Accuracy

The accuracy of both new implementations seems to be very similar to the accuracy provided by C-XSC.

$$\text{Let } A = \begin{pmatrix} 83 & 86 & 77 & 15 & 93 & 35 & 86 & 92 & 49 & 21 \\ 62 & 27 & 90 & 59 & 63 & 26 & 40 & 26 & 72 & 36 \\ 11 & 68 & 67 & 29 & 82 & 30 & 62 & 23 & 67 & 35 \\ 29 & 2 & 22 & 58 & 69 & 67 & 93 & 56 & 11 & 42 \\ 29 & 73 & 21 & 19 & 84 & 37 & 98 & 24 & 15 & 70 \\ 13 & 26 & 91 & 80 & 56 & 73 & 62 & 70 & 96 & 81 \\ 5 & 25 & 84 & 27 & 36 & 5 & 46 & 29 & 13 & 57 \\ 24 & 95 & 82 & 45 & 14 & 67 & 34 & 64 & 43 & 50 \\ 87 & 8 & 76 & 78 & 88 & 84 & 3 & 51 & 54 & 99 \\ 32 & 60 & 76 & 68 & 39 & 12 & 26 & 86 & 94 & 39 \end{pmatrix} \text{ and } b = \begin{pmatrix} 95 \\ 70 \\ 34 \\ 78 \\ 67 \\ 1 \\ 97 \\ 2 \\ 17 \\ 92 \end{pmatrix}$$

This systems has a condition number of $5.24 \cdot 10^{+1}$. The solution of this system $Ax = b$ on XC1 using infimum-supremum and midpoint-radius arithmetic and also the C-XSC solution can be seen in Table 6. The midpoint-radius result was transformed into infimum-supremum representation to make easy a comparison among the results.

In Table 6 is possible to observe the result found by C-XSC. It is contained in the result found using infimum-supremum arithmetic and both are contained in the midpoint radius result. That means that the infimum-supremum result is a little more accurate than the midpoint-radius result. However, both results can be considered very satisfactory, since they are an inclusion of the correct result, and the diameter of the interval is less than one digit larger than the one found by C-XSC.

Table 7 shows the conditional number of several ill conditioned matrices with dimension 4, 8 and 10. It also compares the ability of C-XSC and both new algorithms to solve it.

The new implementation could find a verified result for very ill conditioned systems, while C-XSC could find a verified solution for systems until condition number 10^{15}. It is important to mention that C-XSC has other functionalities to solve very ill conditioned systems, but using just the method on which we based the new implementation, C-XSC could not present a verified result for condition numbers larger than 10^{15}. It is important to mention that these functionalities to solve very ill conditioned problems in C-XSC would, naturally, increase the execution time even more.

Table 6 – Results on XC1.

$x[0]$	c-xsc	$1.33740578923458641 \cdot 10^{-1}$	$1.33740578923458670 \cdot 10^{-1}$
	inf-sup	$1.33740578923455561 \cdot 10^{-1}$	$1.33740578923461528 \cdot 10^{-1}$
	mid-rad	$1.33740578923406184 \cdot 10^{-1}$	$1.33740578923495723 \cdot 10^{-1}$
$x[1]$	c-xsc	$2.84689874808221943 \cdot 10^{-1}$	$2.84689874808221999 \cdot 10^{-1}$
	inf-sup	$2.84689874808219723 \cdot 10^{-1}$	$2.84689874808224497 \cdot 10^{-1}$
	mid-rad	$2.84689874808194798 \cdot 10^{-1}$	$2.84689874808252641 \cdot 10^{-1}$
$x[2]$	c-xsc	$3.00418307730955624 \cdot 10^{-1}$	$3.00418307730955681 \cdot 10^{-1}$
	inf-sup	$3.00418307730953849 \cdot 10^{-1}$	$3.00418307730957623 \cdot 10^{-1}$
	mid-rad	$3.00418307730926315 \cdot 10^{-1}$	$3.00418307730981271 \cdot 10^{-1}$
$x[3]$	c-xsc	$1.75682233712527402 \cdot 10^{0}$	$1.75682233712527425 \cdot 10^{0}$
	inf-sup	$1.75682233712526892 \cdot 10^{0}$	$1.75682233712527980 \cdot 10^{0}$
	mid-rad	$1.75682233712519520 \cdot 10^{0}$	$1.75682233712533908 \cdot 10^{0}$
$x[4]$	c-xsc	$6.62495542349974653 \cdot 10^{-1}$	$6.62495542349974765 \cdot 10^{-1}$
	inf-sup	$6.62495542349969990 \cdot 10^{-1}$	$6.62495542349979649 \cdot 10^{-1}$
	mid-rad	$6.62495542349922806 \cdot 10^{-1}$	$6.62495542350044042 \cdot 10^{-1}$
$x[5]$	c-xsc	$-1.78075378166349841 \cdot 10^{0}$	$-1.78075378166349818 \cdot 10^{0}$
	inf-sup	$-1.78075378166350129 \cdot 10^{0}$	$-1.78075378166349507 \cdot 10^{0}$
	mid-rad	$-1.78075378166353304 \cdot 10^{0}$	$-1.78075378166345577 \cdot 10^{0}$
$x[6]$	c-xsc	$3.72909197413613724 \cdot 10^{-1}$	$3.72909197413613780 \cdot 10^{-1}$
	inf-sup	$3.72909197413610005 \cdot 10^{-1}$	$3.72909197413617277 \cdot 10^{-1}$
	mid-rad	$3.72909197413559379 \cdot 10^{-1}$	$3.72909197413651305 \cdot 10^{-1}$
$x[7]$	c-xsc	$4.97570637450006791 \cdot 10^{-1}$	$4.97570637450006848 \cdot 10^{-1}$
	inf-sup	$4.97570637450004016 \cdot 10^{-1}$	$4.97570637450009456 \cdot 10^{-1}$
	mid-rad	$4.97570637449973818 \cdot 10^{-1}$	$4.97570637450054754 \cdot 10^{-1}$
$x[8]$	c-xsc	$-1.27014384494394195 \cdot 10^{0}$	$-1.27014384494394172 \cdot 10^{0}$
	inf-sup	$-1.27014384494394617 \cdot 10^{0}$	$-1.27014384494393817 \cdot 10^{0}$
	mid-rad	$-1.27014384494399257 \cdot 10^{0}$	$-1.27014384494389354 \cdot 10^{0}$
$x[9]$	c-xsc	$-2.36366016064777252 \cdot 10^{-1}$	$-2.36366016064777223 \cdot 10^{-1}$
	inf-sup	$-2.36366016064780471 \cdot 10^{-1}$	$-2.36366016064774059 \cdot 10^{-1}$
	mid-rad	$-2.36366016064817525 \cdot 10^{-1}$	$-2.36366016064734979 \cdot 10^{-1}$

Table 7 – Accuracy on XC1 Compared with C-XSC.

Cond Num	C-XSC	New Algorithm inf-sup	New Algorithm mid-rad
10^{14}	enclosure found	enclosure found	enclosure found
10^{15}	$2.9 \cdot 10^{15}$ enclosure found but $3.9 \cdot 10^{15}$ enclosure not found	enclosure found	enclosure found
10^{16}	enclosure not found	enclosure found	enclosure found
10^{17}	enclosure not found	large interval (diameter 0.3)	large interval (diameter 0.3)
10^{18}	enclosure not found	enclosure not found	enclosure not found

Performance

Before starting a detailed comparison between the two new implementations, it is important to compare their performances with the well known libraries. Table 8 presents the performance for solving a dense linear system with a random point matrix with dimension $1\,000$. The time is given in seconds.

Table 8 – Performance Comparison (in Seconds) on XC1.

Tool	time
C-XSC	666.50
NEW inf-sup	2.13
NEW mid-rad	2.01
LAPACK	0.20

This table shows that the execution time of both new implementations are very similar and much better than the runtime using C-XSC. The new solver presented a runtime larger than LAPACK runtime. However, it is important to remark that the result given by LAPACK is not a verified solution. LAPACK performance was just used as a comparison since it is a well known tool for its performance.

The idea of this new implementation is to provide a fast solution for large matrices. Table 9 shows the performance of the sequential algorithm in each phase of the solution for several dimensions on XC1 cluster. The largest dimension tested was $10\,000$, since larger dimensions could not be tested in this computer due to memory limitations.

Table 9 – Execution Time (in Seconds) for Point Input Data on XC1.

	Dim	1 000	2 000	3 000	4 000	5 000	6 000	7 000	8 000	9 000	10 000
A^{-1}	inf-sup	0.48	5.87	20.15	46.97	91.46	155.68	248.65	366.33	522.72	712.52
	mid-rad	0.47	5.92	20.14	47.08	91.84	154.83	247.64	369.26	521.56	713.12
x	inf-sup	0.01	0.03	0.07	0.11	0.20	0.27	0.36	0.45	0.56	0.68
	mid-rad	0.01	0.03	0.07	0.11	0.21	0.27	0.35	0.45	0.57	0.68
z	inf-sup	0.11	0.48	1.11	1.98	3.09	4.46	6.11	8.13	10.08	12.45
	mid-rad	0.08	0.40	0.92	1.60	2.49	3.60	4.90	17.22	8.21	10.24
C	inf-sup	0.80	6.07	19.69	46.17	88.49	152.09	240.68	369.36	505.46	693.31
	mid-rad	1.15	7.69	23.46	52.85	99.24	166.90	260.22	442.93	539.98	736.55
$Verif$	inf-sup	0.46	1.88	4.25	7.59	11.83	17.06	23.40	41.52	38.58	47.62
	mid-rad	0.44	1.90	4.31	7.82	12.12	17.41	23.85	73.07	40.18	50.29
$Total$	inf-sup	2.13	16.06	50.21	113.60	215.33	361.26	566.87	855.80	1185.73	1598.51
	mid-rad	2.34	16.98	52.31	117.52	221.22	368.95	577.76	963.68	1196.72	1628.29

Table 9 shows the execution time of each algorithm step. Steps 1 (computation of A^{-1}) and 4 (computation of $[C]$) proved to be the most time consuming steps, as mentioned before. All steps but step 3 have a very similar execution time for both arithmetics. Step 4 presents a higher execution time for midpoint-radius arithmetic. This is a consequence of its arithmetic definitions: to ensure verified results, it needs to perform some other operations that are not needed by infimum-supremum arithmetic.

This table also shows a similar time for calculating z using infimum-supremum and using midpoint-radius arithmetic. As explained in Section 5.2.1, the calculation using infimum-supremum arithmetic has operations involving a negation, that generates case distinctions to define which rounding mode should be used in each case, like in Algorithm 19. The evaluation using midpoint radius do not depends on case distinctions, and is done using the operations defined in Section 2.3.2. The operations for midpoint-radius involves no case distinctions, but more calls to the BLAS routine *dgemv*. This fact equilibrates the time consumed using midpoint-radius with the time for infimum-supremum.

The test case with dimension $8\,000$ presents a cache conflict. It occurs when two blocks (arrays) have a base address alignment and a coherent access pattern [66]. This effect causes an unexpected growth in the execution time of some operations due to bank conflict penalties. Table 10 presents the execution time (in seconds) found for dimension $8\,001$. This result is similar to the one that was expected for dimension $8\,000$, without any unexpected growth.

Table 10 – Execution Time for Point Input Data with Dimension 8 001.

Dim 8 001	Arithm	XC1
A^{-1}	inf-sup	364.24
	mid-rad	365.40
x	inf-sup	0.48
	mid-rad	0.48
z	inf-sup	7.94
	mid-rad	6.42
C	inf-sup	357.16
	mid-rad	384.01
$Verif$	inf-sup	29.78
	mid-rad	31.16
$Total$	inf-sup	819.75
	mid-rad	848.96

In general, the solution using infimum-supremum is a little bit faster than using midpoint-radius. Next section will show the behavior of these two arithmetics when dealing with interval input data.

5.3 Interval Data

When the input matrix is an interval matrix, some considerations and modifications should be done in the algorithm:

- Steps 1 and 2: since the approximation of the inverse matrix A^{-1} and the approximate solution x have no restrictions about their accuracy, aiming at a good performance these operations were implemented using the midpoint value of each interval in the matrix, i.e. $mid([A])$ (the midpoint matrix of [A]);

- Steps 3 and 4: since $[A]$ and $[b]$ are now interval matrix and vector, the evaluation of $[C]$ and $[z]$ should use interval arithmetic;

- Step 5: the verification step uses $[C]$ and $[z]$, two enclosures that were also intervals in the algorithm 18. Therefore, this part of the algorithm remains the same.

The implementation for interval input data is presented in Algorithm 20. The main changes can be seen in step 3 and 4.

Algorithm 20 Enclosure of an Interval Linear System Using LAPACK, BLAS.

```
1: {STEP 1}
2: $R \approx Amid^{-1}${approximation of the inverse using the LAPACK methods: dgetrf_ and dgetri_}
3: {STEP 2}
4: $\tilde{x} \approx R \cdot bmid$ {approximation of the solution}
5: $res = 0;$
6: $cont = 0;$
7: **while** $cont < 15$ or $norm < 10^{-1} \cdot oldnorm$ **do**
8:    $cont++;$
9:    $oldnorm = norm;$
10:   $res = R \cdot (Amid \cdot x + bmid)$ {using BLAS method: dgemv_}
11:   **for** $(i = 0; i < SIZE; i++)$ **do**
12:      $norm = norm + abs(res[i]);$
13:   **end for**
14:   **if** $norm < normold$ **then**
15:      $x = x - res$
16:   **end if**
17: **end while**
18: {STEP 3}
19: $[z] \supseteq R([b] - [A]\tilde{x})$ {enclosure for the residuum using interval arithmetic and directed roundings}
20: {STEP 4}
21: $[C] \supseteq (I - R[A])$ {enclosure for the iteration matrix using interval arithmetic and directed roundings}
22: {STEP 5}
23: $[w] := [z]$, $k := 0$ {initialize machine interval vector}
24: **while not** $([w] \subseteq int[y]$ or $k > 10)$ **do**
25:   $[y] := [w]$ {using interval arithmetic}
26:   $[w] := [z] + [C][y]$ {using interval arithmetic}
27:   $k++$
28: **end while**
29: **if** $[w] \subseteq int[y]$ **then**
30:   $\Sigma(A, b) \subseteq \tilde{x} + [w]$ {using directed roundings} {the solution set ($\Sigma$) is contained in the solution found by the method}
31: **else**
32:   no verification
33: **end if**
```

5.3.1 Infimum-Supremum Approach

This approach presents some modifications compared with the algorithm for point input data that can increase the computation time considerably.

- Steps 1 and 2 will compute the approximation of $mid([A])^{-1}$ and x using the midpoint matrix $mid([A])$ and the midpoint vector $mid([b])$. Therefore the midpoint matrix of $[A]$ and the midpoint vector of $[b]$ must be computed;
- Since $[A]$ and $[b]$ are now interval matrix and vector, the evaluation of $[z]$ (Step 3) and $[C]$ (Step 4) should use interval arithmetic and cannot use BLAS routines. Therefore, the multiplication of 2 matrices and the multiplication of a matrix and a vector were implemented in C, what caused a loss of performance;
- Step 5 stays the same as for point input data, since in both approaches it uses interval arithmetic and directed roundings.

Important remark: Algorithm 20 has much more interval operations than Algorithm 18, and therefore it is even more important to avoid the switching of rounding mode as often as possible.

5.3.2 Midpoint-Radius Approach

The modifications performed at the midpoint-radius algorithm to be able to accept interval input data are the following:

- Steps 1 and 2 for midpoint-radius arithmetic are the same as for infimum supremum arithmetic. With the advantage that no special computation need to be done to find the midpoint matrix, since it already exists for this arithmetic;
- Step 3, the calculation of $Z = R(b - A \cdot x)$, uses the BLAS routine *dgemv* twice and directed roundings to find the direct bounds, and after that, normal floating-point operations with rounding up to evaluate the *zmid* and *zrad* as presented in the section 2.3;
- The calculation of $C = (I - R \cdot A)$ uses the BLAS routine *dgemm* for matrix multiplication and directed roundings twice to find a lower and upper bound. Then, floating-point operations with rounding up for the evaluation of *Crad* and *Cmid* are performed;
- The last step, the verification step, uses midpoint-radius arithmetic also with the BLAS routine *dgemv*.

Another important remark is that it is possible to consider a point matrix as a midpoint matrix with a radius matrix where all elements are null. This makes it very easy to develop an algorithm that can accept both point and interval input data with no big changes.

5.3.3 Tests and Results for Interval Input Data

The following tests were done in the same platform as the test described in Section 5.2.3. Again, we will compare the accuracy and the performance of the implemented interval solutions.

Accuracy

Suppose we want to solve the same system presented in the Section 5.2.3 introducing a radius of 0.0000000001. This interval system has $mid([A])x = mid([b])$ as midpoint system, which has the condition number of $5.24 \cdot 10^{+1}$. The solution of this system $[A]x = [b]$ on XC1 using infimum-supremum and midpoint-radius arithmetic can be seen in Table 11. The midpoint-radius result was transformed into infimum-supremum to make easy a comparison among the results.

Table 11 – Results on XC1.

	inf-sup	$1.33740578898804474 \cdot 10^{-01}$	$1.33740578948044336 \cdot 10^{-01}$
$x[0]$	mid-rad	$1.33740578821250677 \cdot 10^{-01}$	$1.33740579025665607 \cdot 10^{-01}$
	inf-sup	$2.84689874791444586 \cdot 10^{-01}$	$2.84689874825150346 \cdot 10^{-01}$
$x[1]$	mid-rad	$2.84689874738264959 \cdot 10^{-01}$	$2.84689874878184812 \cdot 10^{-01}$
	inf-sup	$3.00418307714720612 \cdot 10^{-01}$	$3.00418307747332636 \cdot 10^{-01}$
$x[2]$	mid-rad	$3.00418307663263551 \cdot 10^{-01}$	$3.00418307798646700 \cdot 10^{-01}$
	inf-sup	$1.75682233708829627 \cdot 10^{+00}$	$1.75682233716261949 \cdot 10^{+00}$
$x[3]$	mid-rad	$1.75682233697100787 \cdot 10^{+00}$	$1.75682233727955306 \cdot 10^{+00}$
	inf-sup	$6.62495542316513530 \cdot 10^{-01}$	$6.62495542383602531 \cdot 10^{-01}$
$x[4]$	mid-rad	$6.62495542210719712 \cdot 10^{-01}$	$6.62495542489227152 \cdot 10^{-01}$
	inf-sup	$-1.78075378168493637 \cdot 10^{+00}$	$-1.78075378164214526 \cdot 10^{+00}$
$x[5]$	mid-rad	$-1.78075378175232002 \cdot 10^{+00}$	$-1.78075378157467945 \cdot 10^{+00}$
	inf-sup	$3.72909197390050628 \cdot 10^{-01}$	$3.72909197437211015 \cdot 10^{-01}$
$x[6]$	mid-rad	$3.72909197315721475 \cdot 10^{-01}$	$3.72909197511502644 \cdot 10^{-01}$
	inf-sup	$4.97570637450004016 \cdot 10^{-1}$	$4.97570637450009456 \cdot 10^{-1}$
$x[7]$	mid-rad	$4.97570637429699592 \cdot 10^{-01}$	$4.97570637470307053 \cdot 10^{-01}$
	inf-sup	$-1.27014384497236565 \cdot 10^{+00}$	$-1.27014384491572496 \cdot 10^{+00}$
$x[8]$	mid-rad	$-1.27014384506151079 \cdot 10^{+00}$	$-1.27014384482638154 \cdot 10^{+00}$
	inf-sup	$-2.36366016089740505 \cdot 10^{-01}$	$-2.36366016039844168 \cdot 10^{-01}$
$x[9]$	mid-rad	$-2.36366016168344129 \cdot 10^{-01}$	$-2.36366015961212594 \cdot 10^{-01}$

In Table 11 it is possible to observe that the result found using infimum-supremum arithmetic and midpoint-radius arithmetic are very similar. It differs from each other only in the 10th place of the mantissa. And most important of all, both presented an inclusion of the correct result.

Performance

To measure the performance, tests were made from dimension $1\,000$ to $10\,000$ with random midpoint matrices and radius 0.0000000001 for all intervals of $[A]$ and $[b]$. Table 12 shows the performance of the sequential algorithms in each phase of the solution for several dimensions on XC1 for both infimum-supremum and midpoint-radius arithmetic. In this table, CTLE means CPU time limit exceeded.

Table 12 – Execution Time (in Seconds) for Interval Input Data on XC1.

	Dim	1 000	2 000	3 00	4 000	5 000	6 000	7 000	8 000	9 000	10 000
A^{-1}	inf-sup	0.49	6.09	19.97	45.86	CTLE	CTLE	CTLE	CTLE	CTLE	CTLE
	mid-rad	0.47	5.82	20.06	46.54	142.86	154.54	246.55	573.33	781.92	1064.2414
x	inf-sup	0.02	0.05	0.06	0.13	CTLE	CTLE	CTLE	CTLE	CTLE	CTLE
	mid-rad	0.02	0.03	0.09	0.11	0.17	0.24	0.32	0.42	0.54	0.91
z	inf-sup	0.26	1.13	2.60	4.57	CTLE	CTLE	CTLE	CTLE	CTLE	CTLE
	mid-rad	0.08	0.40	0.90	1.65	2.79	3.64	4.95	27.90	9.52	11.54
C	inf-sup	168.37	1451.22	4924.72	11692.87	CTLE	CTLE	CTLE	CTLE	CTLE	CTLE
	mid-rad	1.54	10.67	33.05	77.68	142.03	242.30	381.21	710.03	819.79	1130.79
$Verif$	inf-sup	0.45	1.86	4.21	8.03	CTLE	CTLE	CTLE	CTLE	CTLE	CTLE
	mid-rad	0.44	1.90	4.09	9.38	11.74	16.51	23.84	106.49	58.23	57.33
$Total$	inf-sup	169.79	1461.50	4955.33	11760.03	CTLE	CTLE	CTLE	CTLE	CTLE	CTLE
	mid-rad	2.71	19.90	61.62	143.26	315.00	443.14	697.69	1478.28	1754.58	2379.56

From dimension $5\,000$, the execution time of the the infimum-supremum approach exceeded the cpu time limit, and therefore it was not possible to compute the execution time for those dimensions. As said before, the computation of interval matrix multiplication using this arithmetic cannot be implemented using BLAS routines since its definition requires the implementation in C according with its case distinctions. This operation has complexity $O(n^3)$ and therefore its execution time grows considerably from one dimension to another.

This effect can be easily seen when comparing the execution time of the two arithmetics for Step 4, the computation of $[C]$. It has very different execution times for each arithmetic. Decisions about the parallel version were taken because of this experiment. The computation of Step 3 present also a growth in the execution time due to this effect, but since this step deals with operations with complexity $O(n^2)$, it does not affect the total time with such an impact as Step 4.

The test case with dimension $8\,000$ using midpoint-radius arithmetic presents the same cache conflict as we noticed in the algorithms for point data. Table 13 presents the execution time (in seconds) found for dimension $8\,001$. This result is similar to the one that was expected to dimension $8\,000$, without any unexpected growth.

Table 13 – Execution Time for Interval Input Data with Dimension 8 001.

Dim 8 001	XC1
A^{-1}	555.69
x	0.44
z	7.24
C	569.63
$Verif$	34.23
$Total$	1227.55

5.4 Conclusions

This section presented the new sequential solver using infimum-supremum and midpoint-radius arithmetic, for point and interval input data. Table 14 shows a summary of the approach we used to implement the sequential version for each step.

Table 14 – Sequential Approach Used in the New Solver.

dimension		Infimum-supremum	Midpoint-radius	Type of operation	Comments
Step 1	point data	LAPACK	LAPACK	floating-point	complexity $O(n^3)$ (50% total time)
	interval data	LAPACK	LAPACK	floating-point	use the midpoint matrix
Step 2	point data	BLAS	BLAS	floating-point	approximation
	interval data	BLAS	BLAS	floating-point	use the midpoint matrix
Step 3	point data	BLAS	BLAS	interval	use of directed roundings
	interval data	interval routines	interval routines (BLAS)	interval	use of directed roundings
Step 4	point data	BLAS	BLAS	interval	use of directed roundings
	interval data	interval routines	interval routines (BLAS)	interval	complexity $O(n^3)$ (40% total time)
Step 5	point data	interval routines	interval routines (BLAS)	interval	use of directed roundings
	interval data	interval routines	interval routines (BLAS)	interval	use of directed roundings

There are some advantages of using midpoint-radius representation instead of the traditional infimum-supremum representation. The main advantage is that no case distinctions or switching of rounding mode in inner loops of vector and matrix multiplication are needed. In this case, the multiplication of a vector or matrix could be implemented using pure floating-point operations and highly optimizes libraries like BLAS (Basic Linear Algebra Subprograms) [7] could be used to implement it in a much faster way. However, when using infimum-supremum arithmetic, this is not possible. In this case, each interval multiplication has to be executed differently depending if the interval is positive, negative or if it contains zero. Therefore, optimized libraries cannot be used.

This difference between infimum-supremum and midpoint-radius algorithms can be seen in steps 3, 4 and 5. Step 3 and 4 are a problem for infimum-supremum arithmetic only if the input data is of type interval. For point data it is possible to use BLAS, like in midpoint-radius algorithm. It is important to mention that step 3, 4 and 5 of the midpoint-radius algorithm use also interval arithmetic, however its arithmetic can be implemented using BLAS routines. For that reason, there is an improvement of the performance in algorithms based on the midpoint-radius arithmetic.

This effect was presented in Section $5.3.3$ and can be seen when comparing the time for solving a $1\,000 \times 1\,000$ system:

- Point input data:
  - Infimum-supremum: 2.13 seconds;
  - Midpoint-radius: 2.34 seconds.
- Interval input data:
  - Infimum-supremum: 169.79 seconds;
  - Midpoint-radius: 2.71 seconds.

The execution time for interval input data using infimum-supremum arithmetic was much larger than the execution time using midpoint-radius arithmetic. On the other hand, the infimum-supremum numerical result was similar, but a little bit more accurate than the midpoint-radius result.

The midpoint-radius approach seems to be a compromise between performance and accuracy. The result was not so accurate as the one provided when using infimum-supremum, differing from each other in the 10^{th} place of the mantissa, however the execution time is much faster. For an interval system with dimension $4\,000$, the midpoint-radius algorithm needed 143.26 seconds against 3.24 hours needed by the infimum-supremum algorithm. Therefore, we decided to go on with the midpoint-radius implementation.

Aiming at a better performance and to be able to solve larger systems, we decided to implement a parallel version of the new sequential solver using midpoint-radius arithmetic. The approach used for this parallel implementation is described in Chapter 6.

6 Highly Optimized Distributed Memory Solver

This Chapter presents the approach used in the parallel solver. The idea of this parallel implementation is similar to the one used in the sequential solver: high performance and reliability. The first approach was meant to solve point linear systems, and the second to solve interval linear systems. Since we decided to use midpoint-radius arithmetic, this second solver was implemented in such a way that it can be used for both point and interval data.

Section 6.1 presents the parallel optimized libraries which were used in this implementation to achieve high performance, with a description about the data distribution chosen to take advantage of the LAPACK block-partitioned methods. Section 6.2 describes the platform used for the experiments. Sections 6.3 and 6.4 discuss the parallel implementation for point and interval input data respectively. And finally, Section 6.5 presents the conclusions of this Chapter.

6.1 Parallel Optimized Libraries

ScaLAPACK (or Scalable LAPACK) is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of workstations. ScaLAPACK is portable on any computer that supports MPI [79]. It includes a subset of LAPACK routines redesigned for distributed memory. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication. It assumes matrices are laid out in a two-dimensional block cyclic decomposition [5].

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. For such machines, the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor. The fundamental building blocks of the ScaLAPACK library are the parallel Basic Linear Algebra Subprograms (PBLAS) and Basic Linear Algebra Communication Subprograms (BLACS). Figure 6 illustrates the relationship between ScaLAPACK and its components.

PBLAS [15] implements distributed vector-vector, matrix-vector and matrix-matrix operations with the aim of simplifying the parallelization of linear algebra codes, especially when they are implemented on top of the sequential BLAS. It is a distributed memory version of the Level 1, 2 and 3 BLAS, with similar function names and parameters. The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic

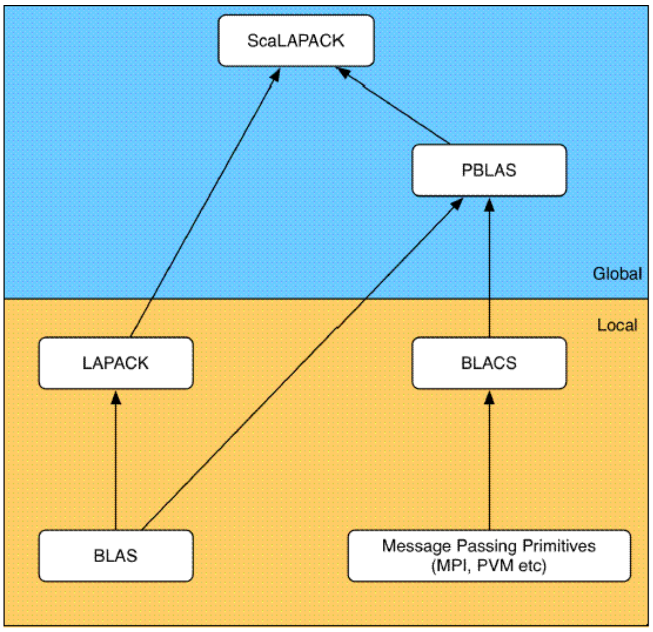


Figure 6 – ScaLAPACK Software Hierarchy.

vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software, LAPACK for example.

The BLACS (Basic Linear Algebra Communication Subprograms) is a set of Basic Linear Algebra Communication Subprograms for communication tasks that arise frequently in parallel linear algebra computations. Its purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms, making linear algebra applications both easier to program and more portable. The BLACS exist in order to make linear algebra applications both easier to program and more portable [23].

There are several implementations of those libraries for different platforms. Since we conducted the tests in a cluster composed of Intel Itanium2, we used the Intel® Math Kernel Library (Intel® MKL) [37]. Intel® MKL is a thread-safe library that offers highly optimized, extensively threaded math routines for scientific, engineering, and financial applications that require maximum performance.

The BLAS and LAPACK libraries are well-known for solving a large variety of linear algebra problems. The Intel® MKL contains an implementation of BLAS and LAPACK that is highly optimized for Intel® processors and provide significant performance improvements over alternative implementations of BLAS and LAPACK. Intel MKL 10.0 is compliant with the new 3.1 release of LAPACK. Intel compared MKL with ATLAS, and the tests showed both an impressive scaling of Intel MKL BLAS (dgemm) and LAPACK (dgetrf) as well as the performance advantage over ATLAS that can be obtained [37].

The Intel MKL library provides the underlying components of ScaLAPACK (Scalable Linear Algebra Package), including a distributed memory version of BLAS (PBLAS or Parallel BLAS) and a set of Basic Linear Algebra Communication Subprograms (BLACS) for interprocessor communication. The Intel MKL implementation of the ScaLAPACK library is specially tuned for Itanium® , Intel® Xeon®, and Intel® Pentium® processor-based systems. Again the tests made by Intel [37] showed that Intel MKL ScaLAPACK significantly outperforms NETLIB ScaLAPACK and that Intel MKL is even more impressive when compared to NETLIB ScaLAPACK using ATLAS BLAS.

The Intel MKL libraries are highly optimized for Intel processors and can significantly increase the performance of applications compared to alternative implementations of them.

6.1.1 Data Distribution

On a distributed memory computer the application programmer is responsible for decomposing the data over the processors of the computer. The way in which a matrix is distributed over the processors has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability [6].

In this implementation, matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution [22] used by ScaLAPACK. In this distribution, an M by N matrix is first decomposed into MB by NB blocks starting at its upper left corner. Suppose we have the following 10×10 matrix with MB and NB equal to 3. In this case, we would have the following blocks:

$$\left(\begin{array}{ccc|ccc|ccc|c}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} & A_{0,5} & A_{0,6} & A_{0,7} & A_{0,8} & A_{0,9} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} & A_{1,6} & A_{1,7} & A_{1,8} & A_{1,9} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & A_{2,5} & A_{2,6} & A_{2,7} & A_{2,8} & A_{2,9} \\
\hline
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{3,5} & A_{3,6} & A_{3,7} & A_{3,8} & A_{3,9} \\
A_{4,0} & A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} & A_{4,5} & A_{4,6} & A_{4,7} & A_{4,8} & A_{4,9} \\
A_{5,0} & A_{5,1} & A_{5,2} & A_{5,3} & A_{5,4} & A_{5,5} & A_{5,6} & A_{5,7} & A_{5,8} & A_{5,9} \\
\hline
A_{6,0} & A_{6,1} & A_{6,2} & A_{6,3} & A_{6,4} & A_{6,5} & A_{6,6} & A_{6,7} & A_{6,8} & A_{6,9} \\
A_{7,0} & A_{7,1} & A_{7,2} & A_{7,3} & A_{7,4} & A_{7,5} & A_{7,6} & A_{7,7} & A_{7,8} & A_{7,9} \\
A_{8,0} & A_{8,1} & A_{8,2} & A_{8,3} & A_{8,4} & A_{8,5} & A_{8,6} & A_{8,7} & A_{8,8} & A_{8,9} \\
\hline
A_{9,0} & A_{9,1} & A_{9,2} & A_{9,3} & A_{9,4} & A_{9,5} & A_{9,6} & A_{9,7} & A_{9,8} & A_{9,9}
\end{array}\right)$$

Suppose we have 4 processors. The process grid would be a 2x2 grid as follows:

$$\left(\begin{array}{c|c}
P_0 & P_1 \\
\hline
P_2 & P_3
\end{array}\right)$$

The blocks are then uniformly distributed across the process grid. Thus, every processor owns a collection of blocks [6]. The first row of blocks would be distributed among the first row of the processor grid, that means among P_0 and P_1, while the second row would be distributed among P_2 and P_3, and so on. For this example, we would have:

$$\left(\begin{array}{c|c|c|c} P_0 & P_1 & P_0 & P_1 \\ \hline P_2 & P_3 & P_2 & P_3 \\ \hline P_0 & P_1 & P_0 & P_1 \\ \hline P_2 & P_3 & P_2 & P_3 \end{array}\right)$$

According with this distribution, each processor would have the following data:

$$P_0 = \left(\begin{array}{ccc|ccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,6} & A_{0,7} & A_{0,8} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,6} & A_{1,7} & A_{1,8} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,6} & A_{2,7} & A_{2,8} \\ \hline A_{6,0} & A_{6,1} & A_{6,2} & A_{6,6} & A_{6,7} & A_{6,8} \\ A_{7,0} & A_{7,1} & A_{7,2} & A_{7,6} & A_{7,7} & A_{7,8} \\ A_{8,0} & A_{8,1} & A_{8,2} & A_{8,6} & A_{8,7} & A_{8,8} \end{array}\right) \quad P_1 = \left(\begin{array}{ccc|c} A_{0,3} & A_{0,4} & A_{0,5} & A_{0,9} \\ A_{1,3} & A_{1,4} & A_{1,5} & A_{1,9} \\ A_{2,3} & A_{2,4} & A_{2,5} & A_{2,9} \\ \hline A_{6,3} & A_{6,4} & A_{6,5} & A_{6,9} \\ A_{7,3} & A_{7,4} & A_{7,5} & A_{7,9} \\ A_{8,3} & A_{8,4} & A_{8,5} & A_{8,9} \end{array}\right)$$

$$P_2 = \left(\begin{array}{ccc|ccc} A_{3,0} & A_{3,1} & A_{3,2} & A_{3,6} & A_{3,7} & A_{3,8} \\ A_{4,0} & A_{4,1} & A_{4,2} & A_{4,6} & A_{4,7} & A_{4,8} \\ A_{5,0} & A_{5,1} & A_{5,2} & A_{5,6} & A_{5,7} & A_{5,8} \\ \hline A_{9,0} & A_{9,1} & A_{9,2} & A_{9,6} & A_{9,7} & A_{9,8} \end{array}\right) \quad P_3 = \left(\begin{array}{ccc|c} A_{3,3} & A_{3,4} & A_{3,5} & A_{3,9} \\ A_{4,3} & A_{4,4} & A_{4,5} & A_{4,9} \\ A_{5,3} & A_{5,4} & A_{5,5} & A_{5,9} \\ \hline A_{9,3} & A_{9,4} & A_{9,5} & A_{9,9} \end{array}\right)$$

The distribution of a vector is done considering the vector as a column of the matrix. This approach implies that only the processors of the first column in the process grid would contain the vector data. Suppose we have the same process grid as before, and we have a vector with 10 elements:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \hline b_3 \\ b_4 \\ b_5 \\ \hline b_6 \\ b_7 \\ b_8 \\ \hline b_9 \end{pmatrix} \text{ distributed among the processors as follows: } \begin{pmatrix} P_0 \\ \hline P_2 \\ \hline P_0 \\ \hline P_2 \end{pmatrix}$$

In this case, processors 1 and 3 would not contain any vector data.

6.2 Platform

The software platform adopted to implement the proposed solution is composed of optimized versions of libraries ScaLAPACK and PBLAS (MKL 10.0.011) plus the standard Message Passing Interface (MPI), more specifically the mpich implementation (version 1.2). The basic operating system on each node is HP XC Linux for High Performance Computing and the compiler used was the gcc version 3.4.6.

The hardware platform is the HP XC6000 cluster, also called XC1, located at the Computing Center of the University of Karlsruhe. The HP XC6000 is a distributed memory parallel computer with three different sets of nodes all in all. In this chapter, all experiments were carried out over the set composed by 108 2-way HP Integrity rx2620 nodes with 12 Gb memory, each one containing 2 Intel Itanium2 processors (1.5 GHz) with a 6 MB of level 3 cache. The theoretical peak performance of the system is 1.9 TFLOPS. All nodes are connected to the Quadrics QsNet II interconnection which shows a bandwidth over 800 MB/s and a low latency.

6.3 Approach for Point Input Data

To implement the parallel version of Algorithm 18 (Section 5.2), we used an approach for cluster architectures with message passing programming model (MPI [79]) and the highly optimized library PBLAS and ScaLAPACK. Clusters of computers are considered a good option to achieve better performance without using parallel programming models oriented to very expensive (but not frequently used) machines. A parallel version for this algorithm runs on distributed processors, requiring communication among the processors connected by a fast network and the communication library.

The self-verified method previously presented is divided in several steps. By tests, the computation of R, the approximate inverse of matrix A, takes more than 50% of the total processing time. Similarly, the computation of the interval matrix $[C]$ that contains the exact value of $I - RA$ (iterative refinement) takes more than 40% of the total time, since matrix multiplication requires O(n^3) execution time, and the other operations are mostly vector or matrix-vector operations which require at most O(n^2). Both operations could be implemented using ScaLAPACK (R calculation) and PBLAS (C calculation).

A parallel version of the self-verified method for solving linear systems was presented in [46, 47]. We now propose a new parallel version with the following improvements aiming at a better performance:

- Calculate R using just floating-point operations;
- Avoid the use of C-XSC elements that could slow down the execution;
- Use of the fast and highly optimized libraries: PBLAS and ScaLAPACK;
- Matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution [22] used by ScaLAPACK;
- Use midpoint-radius interval arithmetics (as proposed by Rump [77]).

As presented in Chapter 5, sequential algorithms for point and interval input data were written using both infimum-supremum and midpoint-radius arithmetic. The performance tests showed that the midpoint-radius algorithm needs approximately the same time to solve a linear system with point or interval input data, while the infimum-supremum algorithm needs much more time in the interval case, since the interval multiplication must be implemented with case distinctions (each interval multiplication has to be executed differently depending if the interval is positive, negative or if it contains zero) and optimized functions from BLAS/PBLAS cannot be used. Therefore, midpoint-radius arithmetic was chosen for the parallel implementation.

The parallel implementation uses the following ScaLAPACK/PBLAS routines to implement Algorithm 18:

- ScaLAPACK
  - *pdgetrf*: for the LU decomposition (matrix R on step 1);
  - *pdgetri*: to find the approximation of the inverse matrix (matrix R on step 1).
- PBLAS
  - *pdgemm*: matrix-matrix multiplication (matrix C on step 4);
  - *pdgemv*: matrix-vector multiplication (many steps: 2, 3 and 5 to find the vectors x, z and w).

It is important to remember that behind every midpoint-radius interval operation more than one floating-point operation should be done using the appropriate rounding mode. The matrix multiplication $R \cdot A$ from step 4 needs actually 4 operations, like defined in Algorithm 14. In this algorithm, the routine *pdgemm* of PBLAS would be called three times: one for step 1, one for step 2, and one for step 4.

6.3.1 Experimental Results

This section presents some experimental results of the new parallel implementation. Three different tests were executed: accuracy, performance and a real problem test. This set of experiments represents the range of aspects that should be tested in parallel verified algorithms. These experiments were conducted in the platform described in Section 6.2.

Accuracy

The accuracy depends on the condition number of the matrix A. For well conditioned problems, the new algorithm may deliver a very accurate result with up to 16 correct digits.

For example supposing A is a 10×10 point matrix with condition number $4.06 \cdot 10^{+02}$ and b is a vector where each position has the value 1.0. The solution for x delivered by the developed algorithm is presented in Table 15 in both the midpoint-radius and infimum-supremum representation. Analyzing this results, it is possible to see that the radius of the solution is very small, and therefore it is a very accurate solution.

Table 15 – Midpoint-Radius and the Equivalent Infimum-Supremum Result.

Res	Midpoint	Radius
x[0] =	-1.81645396742261731e-05	3.30189340310308432e-18
x[1] =	-1.81103764097278691e-06	1.27042988556363316e-18
x[2] =	-3.50284396008609789e-06	8.65422361039543574e-19
x[...] =	...	...
x[8] =	-2.65213537238245186e-06	1.12671706774209183e-18
x[9] =	3.01161008622528871e-05	4.81485187791373545e-18

Res	Infimum	Supremum
x[0] =	-0.00001816453967423	-0.00001816453967422
x[1] =	-0.00000181103764097	-0.00000181103764097
x[2] =	-0.00000350284396009	-0.00000350284396009
x[...] =	...	...
x[8] =	-0.00000265213537238	-0.00000265213537238
x[9] =	0.00003011610086225	0.00003011610086226

The new implementation also finds an inclusion for ill conditioned dense matrices, but the accuracy may vary depending on the condition number as presented in Table 16.

It is important to mention that this relation between condition number and diameter of the resulting interval was found for a special class of matrix: square, dense with random numbers.

Table 16 – Relation Between Condition Number and Diameter.

Condition number	diameter
10^1	10^{-14}
10^2	10^{-13}
10^3	10^{-12}
10^4	10^{-11}
10^5	10^{-10}
10^6	10^{-9}
10^7	10^{-8}
10^8	10^{-7}
10^9	10^{-6}

A well-known example of ill conditioned matrix are the Boothroyd/Dekker matrices (see Section 3.4.2), which for dimension 10 has a condition number of $1.09 \cdot 10^{+15}$. The result found for this system by this parallel solver is presented in Table 17.

Table 17 – Results for the Boothroyd/Dekker 10x10 Matrix.

Res	Midpoint	Radius	Infimum	Supremum
x[0]	5.4711703e-07	2.8569585e-06	-0.0000023	0.0000034
x[1]	9.9999473e-01	2.7454332e-05	0.9999672	1.0000221
x[2]	-1.9999718e+00	1.4633209e-04	-2.0001182	-1.9998255
x[3]	2.9998902e+00	5.7081064e-04	2.9993194	3.0004611
x[4]	-3.9996506e+00	1.8174619e-03	-4.0014680	-3.9978331
x[5]	4.9990383e+00	5.0008707e-03	4.9940374	5.0040392
x[6]	-5.9976307e+00	1.2322380e-02	-6.0099531	-5.9853083
x[7]	6.9946526e+00	2.7799205e-02	6.9668534	7.0224518
x[8]	-7.9887617e+00	5.8434700e-02	-8.0471964	-7.9303270
x[9]	8.9777416e+00	1.1572349e-01	8.8620181	9.0934651

As expected for an ill conditioned problem, the accuracy of the results is not the same as for a well-conditioned problem. It is important to remark that even if the result has an average diameter of $4.436911 \cdot 10^{-02}$, it is an inclusion. In other words, it is a verified result.

Performance

Performance analysis of this parallel solver was carried out varying the order of input matrix A. Matrices with three different orders were used as test cases: $2\,500 \times 2\,500$, $5\,000 \times 5\,000$, and $10\,000 \times 10\,000$. For each of those matrices, executions with the number of processors varying from 1 to 64 were performed. All matrices are dense and were specifically generated for these experiments and are composed of random numbers. These random numbers were generated with the function *rand()*.

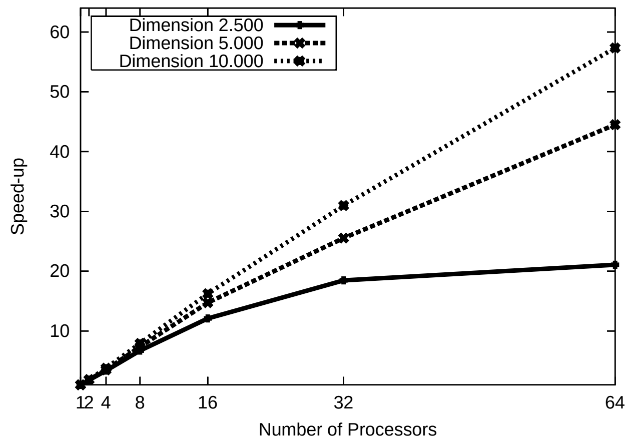


Figure 7 – Experimental Results - Speed-up.

Figure 7 presents a comparison of the scalable speed-ups achieved for the tested matrices. The proposed parallel solution presents a good scalability and improves the performance of the application. As expected, the larger the input matrix, the better is the speed-up achieved.

For larger dimensions, the speed-up is almost linear. In some cases, like for dimension $10\,000$ and 16 processors, we found a super linear speed-up. This is possible due to cache effects resulting from the different memory hierarchies of a modern computer. In this case, the size of accumulated caches from different processors can also change, and sometimes all data can fit into caches and the memory access time reduces dramatically, leading to a drastic speed-up. It is understandable that this effect occurs in this implementation since PBLAS and ScaLAPACK were optimized to make the best possible use of cache hierarchies.

Results for more than 64 processors were not presented for these test cases since beyond this threshold the performance started to drop down. As can be seen in Figure 8, for 64 processors, the efficiency drops significatively, for the test case with dimension $2\,500$, it drops under 35%.

Real Problem

For a real problem test, the used matrix is from the application of the Hydro-Quebec power systems' small-signal model, used for power systems simulations [4]. This problem uses a square $1\,484$ x $1\,484$ real unsymmetric matrix, with $6\,110$ entries ($1\,126$ diagonals, $2\,492$ below diagonal, $2\,492$ above diagonal) as can be seen in Figure 9.

The presented solver was written for dense systems, therefore, this sparse systems will be treated as a dense system. No special method or data storage was used/done concerning the sparsity of this systems.

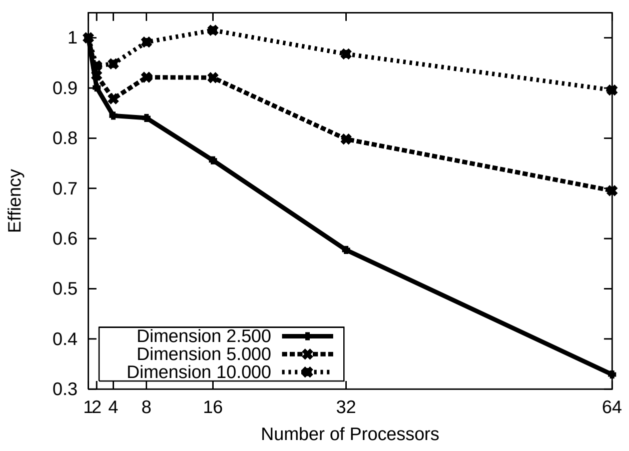


Figure 8 – Experimental Results - Efficiency.

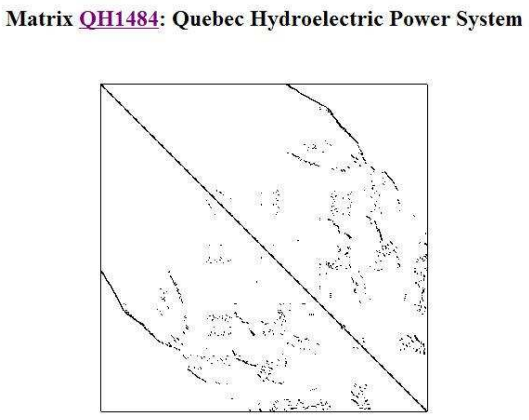


Figure 9 – Structure View of Matrix QH1484.

The first elements of the result vector found for this problem with condition number $5.57 \cdot 10^{17}$ is presented in Table 18.

Table 18 – Results for QH1484: Quebec Hydroelectric Power System.

Res	Midpoint	Radius	Infimum	Supremum
x[0]	$-4.1415310 \cdot 10^{+00}$	$2.0242898 \cdot 10^{-07}$	$-4.1415312 \cdot 10^{+00}$	$-4.1415308 \cdot 10^{+00}$
x[1]	$-2.1936961 \cdot 10^{+00}$	$2.3526014 \cdot 10^{-07}$	$-2.1936964 \cdot 10^{+00}$	$-2.1936959 \cdot 10^{+00}$
x[2]	$-4.1417322 \cdot 10^{+00}$	$2.0242898 \cdot 10^{-07}$	$-4.1417324 \cdot 10^{+00}$	$-4.1417320 \cdot 10^{+00}$
x[3]	$-2.1954030 \cdot 10^{+00}$	$2.3526014 \cdot 10^{-07}$	$-2.1954032 \cdot 10^{+00}$	$-2.1954028 \cdot 10^{+00}$
x[...]	...	...	...	...

Despite it is an ill conditioned problem, the average diameter of the interval results found by this solver was $1.26 \cdot 10^{-8}$. This is a very accurate result for such an ill conditioned problem.

As shown before, for dimension $2\,500$ the efficiency drops very fast for small matrices.

Therefore we measure the execution time of this matrix until 8 processors. The execution time and speed-up for solving this systems of linear equations is presented in Table 19. This performance can be considered very fast to find a verified solution of a linear system.

Table 19 – Real Problem Execution Time and Speed-up.

Number of proc.	Execution Time (sec)	Speed-up
1	8.728	1.000
2	4.307	2.026
4	2.186	3.992
8	1.419	6.150

6.4 Approach for Interval Input Data

Many real problems are simulated and modeled using dense linear systems of equations. In many problems the input data is obtained from measurements that are usually done using tools that are not always precise. This fact, among other reasons that generate uncertainties (see Section 2.2), can cause errors in the evaluation of the system, and this can be specially bad for problems that need an accurate result. One possible way to handle the uncertainty of input data is using intervals.

Many different numerical algorithms contain this kind of task as a sub-problem [3, 81, 89]. A large number of these problems can be solved through a dense interval linear system of equations like

$$[A]x = [b] \tag{6.1}$$

with a $n \times n$ interval matrix $[A] \in \mathbb{IR}^{n\times n}$ and a right hand side interval vector $[b] \in \mathbb{IR}^n$.

To solve an interval linear systems means that an infinite number of matrices contained in the interval should be solved. The exact solution set of an interval linear system has a complicated non-convex shape which is difficult to compute [73]. The only possible way to find a solution is to compute a narrow interval that contains this set.

Each of these matrices has a different condition number, and the larger the dimension of the matrix and the radius of the matrix are, the larger is the probability that it will contain an ill conditioned or a singular matrix.

With the advent of large clusters composed by hundreds or thousands of nodes connected through a gigabit network, huge linear systems can be computed in parallel [24]. Well-known libraries such as PBLAS and ScaLAPACK alone are not suited for interval systems based on verified methods once these methods introduce several steps to deliver a guaranteed result.

This section proposes a new cluster solution for solving interval linear systems using high performance computing with MPI and combining the previous mentioned libraries with verified computing techniques in order to treat uncertain data represented by intervals.

The idea of this approach for interval systems is the same as for point systems: to use popular and highly optimized libraries to gain performance and verified methods to have guaranteed results. In other words, the new implementations try to join the best aspects of each library:

- ScaLAPACK and PBLAS: performance;
- Verified method like the one used in C-XSC toolbox: guaranteed results.

This text is organized as follows. Section 6.4.1 describes the implementation issues and Section 6.4.2 shows some experimental results.

6.4.1 Implementation Issues

The idea of this implementation is to increase the performance of a solver for dense linear systems using a verified method (but no special library for verification, since those libraries can increase the computational time significantly [46]), interval arithmetic, directed roundings and optimized numerical libraries like PBLAS and ScaLAPACK aiming to provide both self-verification and speed-up at the same time.

To implement the parallel version of Algorithm 20 (Section 5.3), we used an approach for cluster architectures with message passing programming model (MPI) and wherever possible the highly optimized libraries PBLAS and ScaLAPACK. Matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution used by ScaLAPACK. Next sections present the approach used for each step of this parallel algorithm for solving interval systems.

Approximate Solution

The first two steps of the algorithm compute R, the approximate inverse matrix of $mid([A])$, and x, the approximate solution of the linear system.

Since the input matrix is an interval matrix, a decision should be made about how the steps 1 and 2 of the Algorithm 20 will be implemented. In both, the mathematical and performance point of views, the approximate inverse matrix R should be computed using just the midpoint matrix, i.e. a point matrix. In this case, the result will be much more accurate than using the interval input data, because the interval inverse matrix $[A]^{-1}$ would contain huge intervals with almost no information.

Aiming at a good performance, the idea is to find the approximate inverse R and the approximate solution x using just the midpoint matrix and floating-point operations. Later on, the computation of the residuum will use interval arithmetic and the original interval matrix $[A]$ and the interval vector $[b]$ to guarantee that we will find the most accurate result possible.

For the computation of the approximation R in step 1, the following ScaLAPACK routines are used:

- *pdgetrf* - computes in parallel a LU factorization of a general matrix using partial pivoting (with row interchanges) using double precision;
- *pdgetri* - computes in parallel the inverse of a matrix using the LU factorization computed by *pdgetrf* using double precision.

For the computation of the approximative solution x in step 2, the following PBLAS routine is used:

- *pdgemv* - performs the matrix-vector operation in parallel using double precision.

Enclosures for the Verification Iteration

Steps 3 and 4 from algorithm 20 compute the enclosure needed to start the verification iteration. Step 3 computes the enclosure for the residuum and step 4 the enclosure for the iteration matrix.

Since $[A]$ and $[b]$ are respectively an interval matrix and vector, for the evaluation of $[C]$ and $[z]$, it is essential to use midpoint-radius interval arithmetic.

As discussed in Section 5.4, the major advantage of midpoint-radius arithmetic is that the matrix operations use exclusively pure floating-point matrix operations and there is no need for case distinctions. Therefore, Step 4 (the calculation of $z = R(b - A \cdot x)$) is implemented using the PBLAS routine *pdgemv* and directed roundings.

The calculation of $C = (I - R \cdot [A])$ uses the PBLAS routine *pdgemm* (for double precision matrix multiplication) twice: once rounded up and once rounded down to find lower and upper bounds of the interval matrix $[C]$. Then, floating-point operations are performed with rounding up for the evaluation of midpoint of C ($\tilde{c}$) and radius of C ($\tilde{\gamma}$) as presented in algorithm 14.

Verification Iteration

The verification iteration is performed by step 5. It implements a Newton-like iteration to find the enclosure of the correct solution.

These steps use the midpoint-radius arithmetic with directed roundings. In order to implement it, the PBLAS routine *pdgemv* was used to compute the interval vector/matrix multiplication. This iteration does not represent a large amount of computation time, since its operation is of complexity O(n^2). Even so, the parallelization introduced by the PBLAS routine helps to improve the overall performance.

The while statement checks if the new result is contained in the interior of the previous result. If it is true, the while loop is finished, and the enclosure was found. If not, it tries for 10 iterations to find the enclosure. It is an empirical fact that the inner inclusion is satisfied nearly after a few steps or never [31]. In this part of the algorithm, the processors have to communicate to check if the enclosure was found (since the result vectors containing the midpoint and the radius are split among the processors).

6.4.2 Experimental Results

The experiments performed to test the method proposed have been carried out in the same environment described in Section 6.2. After that, analysis about the performance and accuracy of the obtained results are presented at the end of this section.

Input Data

The input interval matrix $[A]$ and vector $[b]$ were generated as follows:

- The midpoint matrix $mid([A])$ and midpoint vector $mid([b])$ were generated with random numbers.
- The radius matrix $rad([A])$ and the radius vector $rad([b])$ were filled with the value $0.1 \cdot 10^{-10}$.

The tests were performed using different matrix dimensions from $1\,000$ to $9\,000$. Larger dimensions were not tested due to memory limitations. This algorithm generates the matrices and vectors in one processor and distributes it among the others. Therefore no larger matrices could be generated. Chapter 7 shows a version that generates the blocks of the matrices and vectors in each processor, making possible to solve very large linear systems until dimension $100\,000$.

Accuracy

The accuracy depends on the condition number of all possible matrices in $[A]$. For well conditioned problems, the new algorithm may deliver a very accurate result with up to 16 correct digits.

For example, let $[A]$ be an $1\,000$ x $1\,000$ interval matrix and $[b]$ an $1\,000$ interval vector, both generated as described in subsection 7.3.1. The generated midpoint matrix $mid([A])$ has a condition number $5.77 \cdot 10^{+04}$. The sequential algorithm and the new parallel algorithm with 4 processors will deliver the solution for the first 10 positions of vector $[x]$ presented in table 20. The parallel results are in fact more accurate than the sequential result in almost all cases.

As can be seen in tables 20, the radius of the parallel solution is smaller than the sequential version (possibly by changes in the sequence of operations). Since we divided the problem in smaller parts, it is possible that the condition of the scalar product of smaller parts is better

Table 20 – Sequential and Parallel Midpoint-Radius Result.

res	Sequential midpoint	Sequential Radius	Parallel midpoint	Parallel radius
x[0] =	$-1.3287445 \cdot 10^{+00}$	$1.34971880 \cdot 10^{-06}$	$-1.3287445 \cdot 10^{+00}$	$6.63489228 \cdot 10^{-07}$
x[1] =	$1.48876460 \cdot 10^{+00}$	$1.93288065 \cdot 10^{-06}$	$1.48876460 \cdot 10^{+00}$	$9.50282020 \cdot 10^{-07}$
x[2] =	$-1.4043083 \cdot 10^{+00}$	$1.07750578 \cdot 10^{-06}$	$-1.4043083 \cdot 10^{+00}$	$5.29588854 \cdot 10^{-07}$
x[3] =	$3.77801931 \cdot 10^{+00}$	$2.42986973 \cdot 10^{-06}$	$3.77801931 \cdot 10^{+00}$	$1.19457841 \cdot 10^{-06}$
x[4] =	$1.51834157 \cdot 10^{+00}$	$1.57890684 \cdot 10^{-06}$	$1.51834157 \cdot 10^{+00}$	$7.76385827 \cdot 10^{-07}$
x[5] =	$-1.0618383 \cdot 10^{+00}$	$1.14296727 \cdot 10^{-06}$	$-1.0618383 \cdot 10^{+00}$	$5.61832108 \cdot 10^{-07}$
x[6] =	$4.75233990 \cdot 10^{-01}$	$1.21834560 \cdot 10^{-06}$	$4.75233990 \cdot 10^{-01}$	$5.98872205 \cdot 10^{-07}$
x[7] =	$-2.0933523 \cdot 10^{+00}$	$1.48170529 \cdot 10^{-06}$	$-2.0933523 \cdot 10^{+00}$	$7.28515143 \cdot 10^{-07}$
x[8] =	$1.08921106 \cdot 10^{-01}$	$9.41392800 \cdot 10^{-07}$	$1.08921106 \cdot 10^{-01}$	$4.62543062 \cdot 10^{-07}$
x[9] =	$-9.1790106 \cdot 10^{-02}$	$9.33610885 \cdot 10^{-07}$	$-9.1790106 \cdot 10^{-02}$	$4.58689372 \cdot 10^{-07}$

than the condition of the complete scalar product. This effect was already studied in [14, 58]. In these papers, it was shown that if the summation is done in pairs, the final result is more accurate than doing the whole summation at once. This effect can be a reason why we found a more accurate result using the parallel algorithm. Theoretically, it would also be possible to have a worse conditioned scalar product when dividing it into parts. In these tests, we did not find any case where the parallel solution was less accurate than the sequential. Despite of this small difference, there was no loss of accuracy in the results. It is important to mention that as required by the algorithm, both results contain the exact result.

Performance

The results presented in this section were obtained through a mean of 10 executions discarding the lowest and the highest times. This procedure is required in order to minimize the influence of environment fluctuations in the final results.

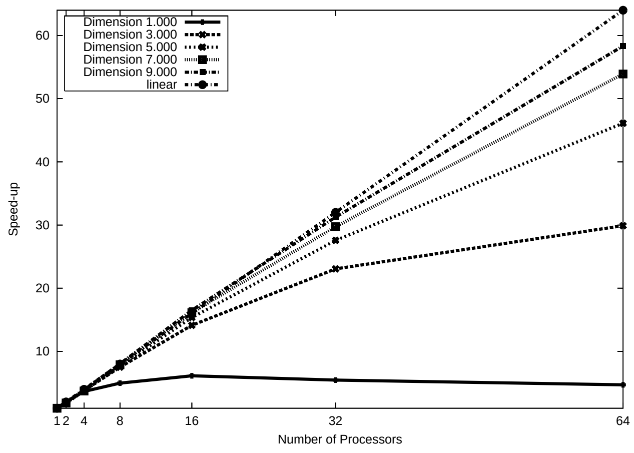


Figure 10 – Speed-up for Interval Input Data.

Figure 10 shows the speed-up curves for matrix dimensions varying from dimension 1 000 to dimension 9 000. As expected, the speed-up of matrix dimension 1 000 drops down very quickly after the execution over few processors (8 in this case, since from 8 processors the amount of computation is not so large compared with the cost of communication among the processors). After that dimension, the speed-ups follow the growth of the amount of data to be computed. For matrix dimension 9 000, the speed-up factor is around 58.3 for 64 processors, indicating that the proposed solution scales well. Experiments with matrices with dimension larger than 9 000 presented irregular behaviors, certainly due to memory allocation problems since the matrices are not allocated in each node, but generated in one node and then distributed.

Table 21 – Summary of the Test Cases: Execution Times and Efficiency.

dimension		number of processors						
		1	2	4	8	16	32	64
1 000	T	2.74	1.34	0.74	0.55	0.44	0.50	0.58
	E	1.00	1.01	0.91	0.62	0.38	0.17	0.07
3 000	T	62.46	31.61	16.35	8.31	4.42	2.70	2.08
	E	1.00	0.98	0,95	0.93	0.88	0.72	0.46
5 000	T	265.72	144.94	71.43	34.46	17.28	9.63	5.76
	E	1.00	0.91	0.93	0.96	0.96	0.86	0.72
7 000	T	700.79	381.85	187.91	88.80	43.44	23.56	13.00
	E	1.00	0.91	0.93	0.98	1.00	0.92	0.84
9 000	T	1467.71	803.37	375.27	182.34	88.92	46.95	25.15
	E	1.00	0.91	0.97	1.00	1.03	0.97	0.91

Looking closer at the results (see Table 21), it is possible to identify four points where superlinear speed-ups appear: i) matrix dimension 1 000 with 2 processors; ii) matrix dimension 7 000 with 16 processors; iii) matrix dimension 9 000 with 8 processors and iv) matrix dimension 9 000 with 16 processors.

The first case can be easily explained by the fact that the matrix is too small and both MPI processes were allocated to processors sharing the same node and cache memory causing a cache effect. The communication time needed to exchange data between the MPI processes decreases dramatically when compared with the situation in which the processors are in different nodes.

For the remaining cases, the superlinear speed-ups can be explained by the memory needed to run the sequential program. The amount of memory in that case is more than ten times larger than the memory needed to run experiments with 8 or 16 processors. For instance, it is necessary to use approximately 5.18 GB[*] of one processor's memory to run the sequential program against 324 MB per processor when executing with 16 processors [†]. This situation leads to a much smaller number of page faults for the executions with 8 or 16 processors. In the case of a larger number of processors (e.g., 32 or 64), this effect fades down because the communication cost becomes comparatively more significant when the amount of data to be processed decreases.

[*] 8 bytes (double) * 9 000 (rows) * 9 000 (columns) * 8 (number of matrices needed to solve the problem)

[†] 5.18 GB divided by 16

6.5 Conclusions

This Chapter introduced a new parallel implementation based on the new sequential algorithms presented in Chapter 5. Aiming at a better performance, the algorithm was parallelized using the libraries ScaLAPACK and PBLAS. The performance results showed that the parallel implementation leads to very good speed-ups in a wide range of processor numbers for large dimensions. This is a very significant result for clusters of computers.

Table 22 shows a comparison of the speed-up achieved for dimension $5\,000$ by both approaches: point and interval input data.

Table 22 – Speed-up for Point and Interval Input Data for Dimension $5\,000$.

p	point speed-up	interval speed-up
1	1	1
2	1.85	1.83
4	3.51	3.72
8	7.37	7.71
16	14.73	15.40
32	25.54	27.58
64	44.50	46.11

Since the approach for interval input data deals with more operations, this approach means more work to be distributed among the processors. This fact has an impact in the speed-up. As can be seen in Table 22, the point input data approach presented a better speed-up than the other approach only in the case of 2 processors. For all other number of processors, the speed-up presented by the interval input data approach is better.

One important advantage of the presented algorithm is the ability to find a solution even for ill conditioned problems while most algorithms may lead to an incorrect result when it is too ill conditioned (above condition number 10^8). The accuracy of the results in many cases depends on the condition number. However, the result of this method is always an inclusion, given the guarantee that the correct result is inside the interval.

7 Solving Very Large Matrices in Clusters

So far, the focus of this work was mainly to find alternatives to the use of C-XSC, keeping the verification of the results and working properly in parallel to improve the overall performance. Once this goal was achieved, the next problem to face is the input matrix dimension barrier, which until now is around $10\,000$. This limitation is a result of the matrix generation strategy adopted, which concentrates the creation of the input matrix in one node leading very quickly to memory allocation problems. The natural option would be to generate sub-matrices of the input matrix individually on each available node, allowing a better use of the global memory since the original matrix would be partitioned using several local memories.

This Chapter presents the approach used to optimize the parallel solver in such a way that it will be able to solve very large dense linear systems. The idea of this optimization is to generate the matrix A directly in each node according to the two dimensional cyclic block distribution. This seems to be an easy task, but since this distribution has particularities, there are some aspects we have to consider. A mapping function was used to map the global position of matrix A into the local position in matrix MyA. This is specially important to generate matrices that are defined by a function deppending on the indexes of its rows and columns (i and j).

Section 7.1 compares the two platforms used for these experiments. Its configuration has an important impact in the performance achieved by this implementation. Section 7.2 presents the approach used to optimize the parallel implementation and its particularities concerning the mapping function and matrix creation. Section 7.3 shows the experimental results achieved for very large dense systems. And finally, Section 7.4 discusses the conclusions of this Chapter.

7.1 Platform

This section presents a comparison between the clusters. Many of the features presented in Tables 23 and 24 will be key aspects to a better understanding of the results achieved by this implementation.

Table 23 shows the hardware configuration of each cluster. XC1, also called XC6000, was installed in 2005, and in June 2005 was in the position 310 of the top500 list[*]. IC1 is a newer cluster, that was installed in 2007 and in November 2007 was in the position 104 of the top500 list. As can be seen in the table, XC1 is less powerful than IC1. The theoretical peak perfor-

[*]list of the top500 supercomputing sites [82]

Table 23 – Comparison of Clusters - Hardware.

	XC1	IC1
Location	Karlsruhe	Karlsruhe
CPU	Intel Itanium2	Intel Xeon Quad Core
Clock Frequency	1.5 GHz	2.66 GHz
Number of Nodes	108	200
Number of Cores	2	2 x 4
RAM per Node	12 GB	16 GB
Cache per Node	6Mb of level 3 cache	2 x 4 MB of level 2 cache
Hard Disk	146 GB	4 x 250 GB
Network	Quadrics QsNet II interconnect	InfiniBand 4X DDR
point to point bandwidth	800 MB/s	1300 MB/s
latency	low	very low (below 2 microseconds)
Theoretical peak performance	1.9 TFLOPS	17.57 TFLOPS

mance that could be achieved by XC1 is 1.9 TFLOPS, while IC1 could achieve 17.57 TFLOPS (almost 10 times more).

Table 24 – Comparison of Clusters - Software.

	XC1	IC1
OS	HP XC Linux	Suse Linux Enterprise 10 SP
BLAS Library	Intel Math Kernel Library 10.0.011	Intel Math Kernel Library 10.0.2
Compiler	gcc version 3.4.6	gcc version 4.1
MPI	MPICH (version 1.2)	OPENMPI (version 1.2.8)

The software comparison presented in Table 24 shows also some differences between the two clusters, e.g. the different versions of MKL, which is an implementation of the optimized libraries BLAS, LAPACK, PBLAS and ScaLAPACK. This and other factors will also affect the performance of the parallel solver for very large dense systems.

7.2 Approach for Very Large Systems

The main goal of this approach is to be able to generate a specific matrix locally in each node according to the 2-dimensional cyclic block distribution, which is not trivial. In this case, matrix elements $A[i, j]$, which are accessible by a given process, are not accessed as elements of the global matrix A, but rather accessed as elements of the sub-matrix MyA of that specific process. Thus, the sub-matrix MyA is not addressed using the global $[i, j]$ address directly. Instead, the global address is converted to a local address in the sub-matrix MyA (which is stored in the form of a vector) in order to access global positions from local positions of the distributed matrix, and vice-versa. The address conversion is accomplished via a mapping function (see Algorithm 21).

Algorithm 21 Mapping Global Position of A to Local Positions in MyA.

```
1: mapping(int i, int i, int MB, int NB, int LLD, int NPROW, int NPCOL, int RESULT)
2: int lauxi, lauxj
3: lauxi = i/MB/NPROW
4: lauxj = j/NB/NPCOL
5: RESULT = (LLD*((NB*lauxj) + (j%NB))) + (MB*lauxi) + (i%MB)
```

This function requires in addition to i (row in the global matrix) and j (column in the global matrix), the number of columns of a block (NB), the number of lines of a block (MB), the number of rows of the sub-matrix (LLD), the number of rows in the processor grid ($NPROW$) and the number of columns in the processor grid ($NPCOL$). The output result variable will contain the local address in the sub-matrix MyA and represents the local vector position of the respective global matrix element.

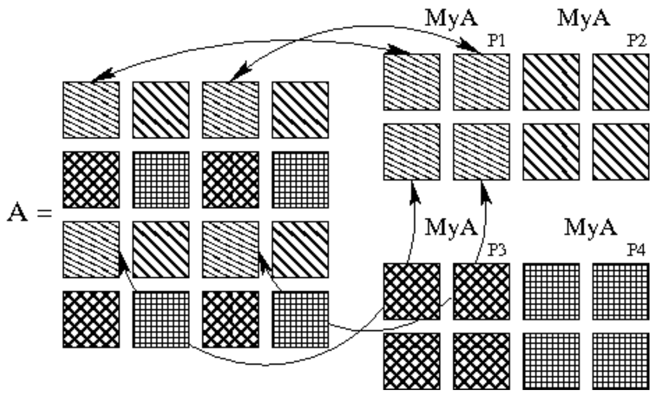


Figure 11 – Graphic Representation of the Mapping Function.

Figure 11 presents the graphical representation of the mapping function. The arrows represent the mapping function, that make possible to generate the data locally on each processor, according to the logical location in matrix A. In this figure, P1, P2, P3, P4 denote the processors.

Accessing global positions of A in the local-matrix is very useful when generating specific matrices in each processor, specially when extra care must be taken to avoid the generation of singular matrices that do not have an inverse.

7.3 Experimental Results

The experiments performed to test the method proposed have been carried out in the two clusters described in Section 7.1. In this section we will present some experiments concerning the accuracy and performance achieved by the solver for very large dense linear systems.

7.3.1 Input Data

The input matrix $mid(A)$ was generated according to 2 test matrices [27]:

- Matrix 1: $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} \frac{i}{j} & \text{if } i \leq j \\ \frac{j}{i} & \text{if } i > j. \end{cases}$$

- Matrix 2: $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} max(i,j) & i,j = 1,2,3...n. \end{cases}$$

Each node generates its own $mid(A)$, also called $mid(MyA) \in mid(A)$. The function mapping, described in Section 7.2, is used to locally generate these pieces of matrix $mid(A)$ through a global view of $mid(A)$. The use of this function makes possible to identify the local pieces that should be generated in each node locally.

The midpoint vector $mid(b)$ was generated with random numbers. Since the condition number gets worse as the matrix dimension grows, we tested this approach only for point input data, and thus the radius matrix $rad(A)$ and the radius vector $rad(b)$ was filled with the value 0.0.

The tests were performed using four different matrix dimensions: $25\,000$, $50\,000$, $75\,000$ and $100\,000$. It is important to remember that a matrix with dimension $100\,000$ has $10\,000\,000\,000$ elements, i.e. 80 GB of memory. This can be considered a very large or even huge dense linear system.

7.3.2 Accuracy

In this section, we will do a qualitative comparison considering the result accuracy found in each cluster. An experiment was conducted using as midpoint matrix the matrix 1 (that will be generated in each node according to the mapping function). The local midpoint vector ($mid(Myb)$) was generated as follows: $mid(Myb)[i] = i$, for i=$0, 1, 2...$(NUMROW-1), where NUMROW is the number of elements of the local vector. The dimension of the problem is $100\,000$ and the tests were performed using 128 processors on XC1 and IC1. The numerical results found were exactly the same in both clusters and are presented in Table 25.

It is important to notice that the result found by this solver is verified and contains the correct result. This is the main difference in comparison with the traditional algorithms and which makes this solver an important contribution.

Some experiments using matrix 2 were also performed in both clusters. The numerical results are, nevertheless, not so accurate as the ones found for matrix 1, what could be explained

Table 25 – Result on Both Clusters.

res	Midpoint	Radius	Infimum-Supremum Representation
x[0] =	$0.00 \cdot 10^{+00}$	$0.000000 \cdot 10^{+00}$	$[0.00000000000000000, 0.00000000000000000]$
x[1] =	$1.00 \cdot 10^{+00}$	$1.110223 \cdot 10^{-16}$	$[9.99999999999999889, 1.00000000000000000]$
x[2] =	$2.00 \cdot 10^{+00}$	$2.220446 \cdot 10^{-16}$	$[1.99999999999999978, 2.00000000000000000]$
x[3] =	$3.00 \cdot 10^{+00}$	$3.330669 \cdot 10^{-16}$	$[2.99999999999999956, 3.00000000000000044]$
x[4] =	$4.00 \cdot 10^{+00}$	$4.440892 \cdot 10^{-16}$	$[3.99999999999999956, 4.00000000000000000]$
x[5] =	$5.00 \cdot 10^{+00}$	$5.551115 \cdot 10^{-16}$	$[4.99999999999999911, 5.00000000000000089]$
x[6] =	$6.00 \cdot 10^{+00}$	$6.661338 \cdot 10^{-16}$	$[5.99999999999999911, 6.00000000000000089]$
x[7] =	$7.00 \cdot 10^{+00}$	$7.771561 \cdot 10^{-16}$	$[6.99999999999999911, 7.00000000000000089]$
x[8] =	$8.00 \cdot 10^{+00}$	$8.881784 \cdot 10^{-16}$	$[7.99999999999999911, 8.00000000000000000]$
x[9] =	$9.00 \cdot 10^{+00}$	$9.992007 \cdot 10^{-16}$	$[8.99999999999999822, 9.00000000000000178]$

by the condition number of the matrix. Matrix 1 has a condition number of $1.00000 \cdot 10^{+00}$ and presents a very small radius of 10^{-17}. Matrix 2, however, has a worse condition number of $3.99996 \cdot 10^{+10}$, and presents an accuracy of around 10^{-03}. Again it can be observed that the accuracy is related to the matrix condition number.

7.3.3 Performance

Performance analysis of this parallel solver was carried out varying the order of input matrix A. Matrices with large dimension were used as test cases. For each of those matrices, executions with the number of processors varying from 8 to 256 were performed. Table 26 shows approximately the amount of memory needed for one execution of the solver according to the matrix dimension. Since the greatest part of the memory is used to allocate matrices, we are considering for this calculation only the memory needed by them. During the computation, the maximal number of matrices simultaneously allocated is 7. That means, for a matrix with dimension n, using a number of processors p and considering that each double variable uses 8 bytes, we have:

$$\text{Memory} = \frac{n \cdot n \cdot 7 \cdot 8}{p}. \tag{7.1}$$

.

Since the memory available in each node of XC1 and IC1 is respectively 12 GB and 16 GB, it is clear that some of these executions are not possible, as can be seen in Table 26.

Figure 12 shows the execution time needed for matrix 1 to be solved for all dimensions using different numbers of processors on XC1. Since this cluster has only 108 nodes with 2 cores, it was not possible to run experiments with 256 processors.

As can be seen in this figure, the execution times drop very quickly each time we double the

Table 26 – Memory per Processor.

	8	16	32	64	128	256
25 000	4.37GB	2.18GB	1.09GB	546MB	273MB	136.5MB
50 000	17.5GB	8.75GB	4.47GB	2.18GB	1.09GB	546MB
75 000	39.4GB	19.7GB	9.84GB	4.9GB	2.45GB	1.23GB
100 000	70.0GB	35GB	17.5GB	8.75GB	4.37GB	2.18GB

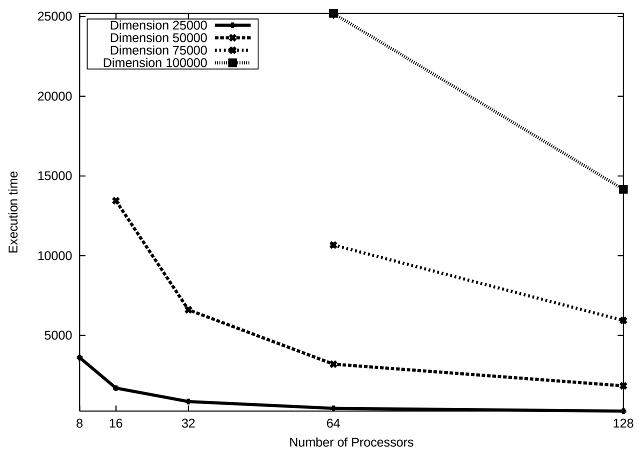


Figure 12 – Execution Time for Solving Matrix 1 on XC1.

number of processors. This fact indicates that the problem scales well even when using a large number of nodes. If we compare this figure with Table 26, we can see that all executions that need more than 12GB of memory do not appear in the figure. It is clear that for these cases, we do not have enough memory to solve the linear system. Except the case with dimension 75 000 using 32 processors, all other cases are represented in Figure 12. This exception can be explained because in the calculation of memory we did not consider the following aspects:

- the memory needed by vectors and other variables needed by the solver;
- the memory that MPI needs for its buffer;
- and the memory needed for the execution of the program and the operating system.

For this case, only the matrices themselves need almost 10 GB. Considering that we have to add to this 10 GB the memory necessary for the previously mentioned items, and since we have only 12 GB available per node, it is possible that this memory was not enough to solve the problem.

Figure 13 shows the execution times for all dimensions and different numbers of processors using IC1. This figure shows a very uniform behavior. However, considering each node of this cluster has 16 GB of memory available, there are some executions missing in this figure.

Comparing the execution times represented in Figure 13 with its respective memory need presented in Table 26, we can see that all the executions that needed more than 8 GB could

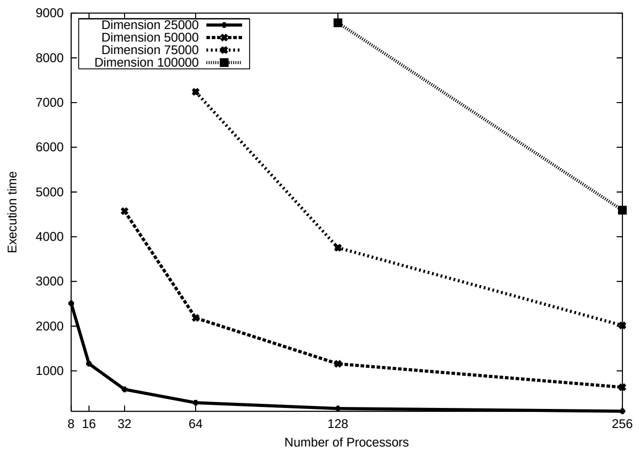


Figure 13 – Execution Time for Solving Matrix 1 on IC1.

not be solved. Since this cluster is still in an experimental phase, it is possible that we cannot address all 16 GB of the memory as stated in its user´s guide, and therefore, we are using just 8 GB instead of 16 GB, what would explain why some of the executions could not be completed.

Since we cannot solve such a large problem using just one processor, we cannot calculate the speed-up achieved in parallel executions. Therefore we calculated a speed-up relative to the number of processors involved. That means if we have 2 times more processors, we expect the relative speed-up to be somewhere near 2. Figure 14 shows the speed-up related to the execution using 64 and 128 processors on XC1. As can be seen in Figure 14, the relative speed-up is near 2 for both matrices.

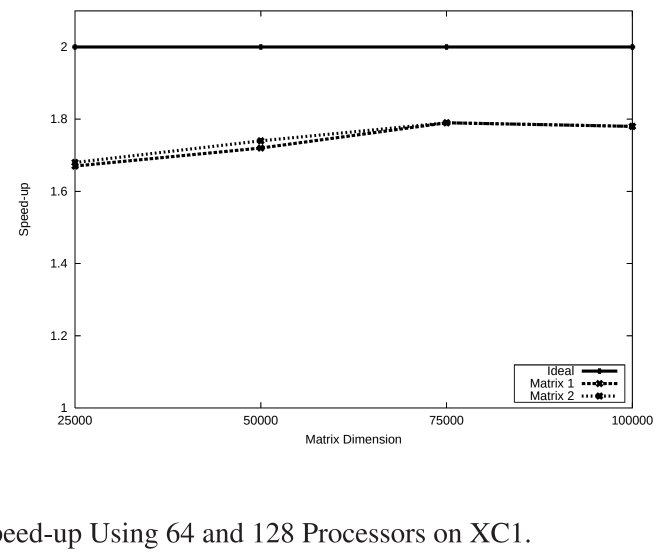


Figure 14 – Relative Speed-up Using 64 and 128 Processors on XC1.

Figure 15 shows the speed-up related to the execution using 128 and 256 processors on IC1. The relative speed-up found in this case was even better. For larger matrices it was near the ideal speed-up, meaning that the problem scales well.

If we compare the figures 14 and 15, we can notice that the execution on IC1 presented

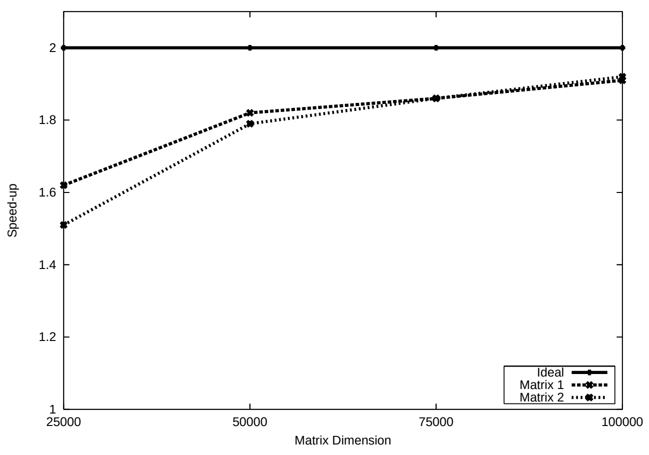


Figure 15 – Relative Speed-up Using 128 and 256 Processors on IC1.

a better speed-up, that grows even for very large dimensions, while on XC1, for dimension $100\,000$, the speed-up slows down a little. This behavior can be possibly explained by the interconnection networks. IC1 has a very fast interconnection network using an Infiniband 4X DDR, while XC1 uses a Quadrics QsNet II interconnection. IC1 interconnection has almost 2 times the bandwidth of XC1 (1 300 MB/s against 800 MB/s), with a very low latency. On XC1, as the grain gets smaller, the communication time represents a larger slice of the execution time, what slows down the speed-up. On the other hand, this fact does not happen on IC1 since its network is very fast and suitable to communication intensive applications. Specifically the very short latency of IC1 makes it ideal for applications doing a lot of collective MPI communications [30].

Table 27 shows the execution times needed to compute each step of Algorithm 18 for matrix 1 with dimension $100\,000$ using 128 processors on both XC1 and IC1.

Table 27 – Execution time needed for each step of the algorithm.

Step	XC1	IC1
R	3 973.52	1 469.73
x	1.13	1.33
z	1.13	1.33
C	10 109.63	3 097.65
verif	52.79	15.37
total	14 159.78	4 594.09

As can be seen in this table, the most expensive operations are steps 1 and 4, which have complexity $O(n^3)$ and represent around 99.4% of the total time. These steps are implemented using PBLAS and ScaLAPACK. Table 27 presents a much higher execution time on XC1 than IC1. This fact can be justified mainly for 3 reasons:

- IC1 has a much faster network interconnection as discussed above;

- The frequency of each core is much higher on IC1 (2.66GHz) than on XC1 (1.5GHz);
- The version of MKL is newer on IC1 (10.0.2) than on XC1 (10.0.011), offering perhaps more optimized routines.

We believe that together these three reasons explain why steps R and C (1 and 4) are around three times faster on IC1 than on XC1.

7.4 Conclusions

This chapter presented optimizations which were added to the parallel solver previously described in chapter 6. These modifications, which are related to the allocation of matrices among the memories of processors, allowed the solver to computes very large dense systems of equations. Indeed, they make it possible to overcome the previous input matrix dimension limit from $10\,000$ to $100\,000$.

In addition to that, in order to investigate the portability of the proposed solution, tests were performed using two clusters with distinct configurations. The results indicated that the proposed optimizations allowed a clever exploitation of the higher computational power of IC1 maintaining a good scalability. On XC1, however, the scalability was not so good, possibly due to its slower interconnection network.

Finally, it was possible to identify some restrictions on the memory allocation when few processors were used. This only reinforces the importance of a scalable solution, since with the use of a large number of processors, the problems of memory allocation disappear. On IC1, for example, it was possible to carry out performance experiments using 256 processors for all matrix dimensions tested.

8 Shared Memory Approach for Dual-Cores

Parallel computing has become a trend for the future. In the last few years, expensive multiprocessor systems have become commodity hardware. Today many users already have dual-core or even quad-core processors at home. Intel has also a prototype of an 80-core processor [86], that will be available in 5 years. Very soon multi processing capacities will be widely available.

The new technology of dual-core processors allows the execution of parallel programs in any modern computer. The available applications, however, are not fully ready to use these new technologies. Even well-studied scientific applications must be adapted in order to fully use all available processors.

Therefore we decided to investigates the use of dedicated threads for speed-up the solver of dense linear systems on duo-core processors [49]. This chapter presents a new multithreaded approach for the problem of solving dense linear systems with verified results through a new method that allows our algorithm to run in a dual-core system without making any changes in the floating-point rounding mode used during the computations, i.e., each processor independently uses its own floating-point rounding strategy to do the computations. This strategy avoids the cost of switching of rounding mode during the computation, which, as said in Section 2.3.2, can be more than 10 times larger than a floating-point operation. The algorithm distributes the computational tasks among the processors based on the floating-point rounding mode required by the task.

Section 8.1 describes the implementation issues, Section 8.2 presents some experimental results and finally the conclusions can be found in Section 8.3.

8.1 Implementation Issues

The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) [35] is the *de facto* standard for floating-point computation and is adopted by all the major microprocessor manufacturers [39]. The standard defines both the representation and operations in floating-point numbers.

At runtime, it is possible to configure a microprocessor to perform all the floating-point operations using one of the following rounding modes: round to nearest, round toward zero, round toward $+\infty$ (rounding-up), and rounding toward $-\infty$ (rounding-down). Using the right round mode is very important to some scientific computations, although it is well-known that changes in the rounding mode are expensive and can impact the performance [10].

The main idea of this implementation is to use threads to explore the benefits of dual-core processors to improve the performance. The algorithm is based on interval arithmetic which needs to change the rounding mode frequently to find the enclosing solution. Using dual-core processors, it is possible to divide the method in two parts: one with rounding-up and another with rounding-down. The natural solution was to use dedicated threads.

This parallelization approach divides the execution of the algorithm in five super-steps, following the Bulk Synchronous Parallel model [85], and utilizes different threads to execute the operations in each rounding mode. All operations that need a particular rounding mode are executed in the same thread.

In order to improve the performance of this implementation, each worker is statically attributed to one available processor. Defining the processor affinity [61] instructs the operating system kernel scheduler to not change the processor used by one particular thread:

- ensuring that no thread dynamic attribution will take place and change the rounding mode expected to be used in the processor, and
- minimizing the number of changes in the rounding mode of each CPU.

Algorithms based on this approach were implemented using the new sequential solver (Chapter 5) as a model and libraries like BLAS and LAPACK to achieve better performance. To ensure that an enclosure will be found, interval arithmetic and directed rounding were used. Algorithms were written for point and interval input data using midpoint-radius arithmetic.

Inter-thread synchronization is done using POSIX shared memory and semaphores primitives. Threads are created and managed using the standard POSIX threads library [13].

8.2 Experimental Results

In order to verify the benefits of these optimizations, three different experiments were performed. The first concerns the correctness of the result. The second experiment was done to evaluate the speed-up improvement brought by the proposed method. The last test uses a real problem to compare both accuracy and execution time.

We used one node of XC1, that means a dedicated computer with 2 Intel Itanium2 processors of 1.6 GHz. The operating system is HP XC Linux for High Performance Computing (HPC), the compiler used was the gnu gcc 3.4.6 and the MKL 10.0.011 was used for an optimized version of libraries LAPACK and BLAS.

8.2.1 Accuracy

Once modifications were done in the algorithm, we conducted some experiments with random well-conditioned and with special ill conditioned matrices to confirm that there were no changes on the accuracy of the result. For well-conditioned matrices, like the sequential version, the parallel implementation delivers a very accurate result.

The Boothroyd/Dekker matrices (see Section 3.4.2) were used for the ill conditioned tests. For $n = 10$ this matrix has a condition number of $1.09 \cdot 10^{+15}$.

The result found by this parallel solver compared with the sequential and with the exact result is presented in Table 28.

Table 28 – Results for the Boothroyd/Dekker 10×10 Matrix.

	exact result	solution with threads	sequential solution
x[0]	0.0	[-0.0000023,0.0000022]	[-0.0000023,0.0000034]
x[1]	1.0	[0.9999785,1.0000223]	[0.9999672,1.0000221]
x[2]	-2.0	[-2.0001191,-1.9998557]	[-2.0001182,-1.9998255]
x[3]	3.0	[2.9995540,3.0004640]	[2.9993194,3.0004611]
x[4]	-4.0	[-4.0014757,-3.9985808]	[-4.0014680,-3.9978331]
x[5]	5.0	[4.9960976,5.0040557]	[4.9940374,5.0040392]
x[6]	-6.0	[-6.0099871,-5.9903856]	[-6.0099531,-5.9853083]
x[7]	7.0	[6.9783176,7.0225133]	[6.9668534,7.0224518]
x[8]	-8.0	[-8.0472766,-7.9544496]	[-8.0471964,-7.9303270]
x[9]	9.0	[8.9097973,9.0935856]	[8.8620181,9.0934651]

As expected for an ill conditioned problem, the accuracy of the results is not as good as the results for well-conditioned problems. It is important to remark that even if the result has an average diameter of $4.436911 \cdot 10^{-02}$, it is an inclusion. In other words, it is a verified result.

The tests generated by the Boothroyd/Dekker formula presented almost the same accuracy on both versions (sequential and multithreaded). Actually, for this example, the result of the parallel version is a narrower interval than the result of the sequential version. As required by the algorithm, both results contain the exact result. This indicates that the rounding was affected by the use of threads (possibly by changes in the sequence of operations), however there was no loss of accuracy of the results.

8.2.2 Performance

This section compares the execution times of both sequential and multithreaded algorithms. Tests were executed for matrix dimensions from $1\,000$ to $10\,000$. As we had a very small standard deviation (less than 1.82 in the worst case; the error bars did not even appear) we just run 10 simulations for each dimension.

In order to validate the proposed parallelization schema, only the computation of the enclosure for the iteration matrix (the C component shown in Algorithm 15) was parallelized. The computation of C requires two matrix multiplications, that is one of the most costly operation performed by the algorithm (complexity $O(n^3)$), so this operation is the first hot spot to be considered in order to improve the performance for this method. Also, first experiments showed that, for this particular case, the parallelization of the other parts of this algorithm suffered from poor performance mainly due to the overhead introduced by lock contention. The fine-grained access to the memory performed by these operations caused, very often, a dispute for the locks that protect the same region of memory between the rounding threads.

A more detailed study on where the use of multi-threaded parallelism can effectively improve the execution time of other parts still needs to be done. There is a clear trade-off between the overhead incurred by thread synchronization and the performance gain. This chapter presents a proof-of-concept on the use of threads to avoid context changes.

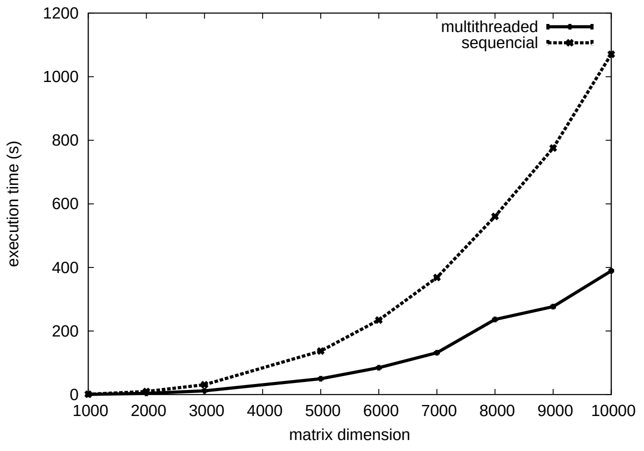


Figure 16 – Time for the Computation of C (Multithreaded x Sequential).

Figure 16 shows the execution times for the C computation (enclosure for the iteration matrix). In this figure, it is possible to see a significant reduction in the execution time. The speed-ups are even super linear on the C computation (the heaviest part of the computation), with speed-ups up to 2.80. This occurs probably due to cache effects as with two processors there are less cache misses. In the sequential version, all matrix elements must be loaded in the cache to compute C with rounding-up, and after that, again, to compute it with rounding-down. If the entire matrix does not fit in the cache, there will be many cache misses for each rounding mode. Since both threads use the same data at the same time, the multithreaded version allows a more effective utilization of the available cache memory, resulting in a better (and even super linear) speed-up as expected.

Despite the fact of threads are being used only in the C computation, it was possible to obtain a reasonable global speed-up that can be seen in Figure 17. The gain for solving a

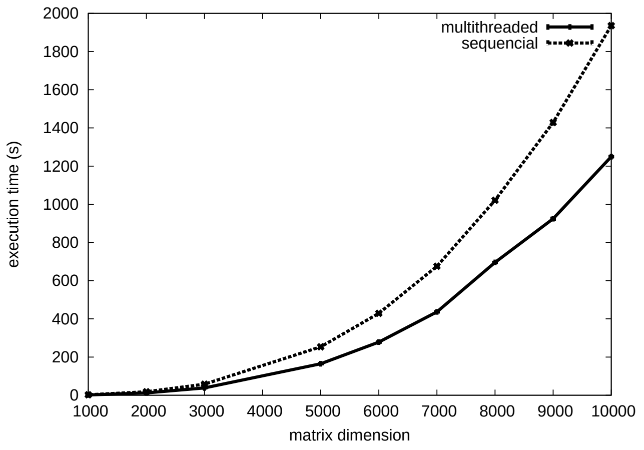


Figure 17 – Total Execution Time for the Multithreaded Version.

system with dimension $10\,000$ was approximately 40%, which is a considerable result for this kind of platform.

8.2.3 Real Problem

The real problem used in this test is part of the application of Alfven Spectra in Magnetohydrodynamics [12], used for Plasma physics. Large nonsymmetric generalized matrix eigenvalue problems arise in the modal analysis of dissipative magnetohydrodynamics (MHD). The MHD system combines Maxwell's and fluid flow equations. The physical objective of these MHD systems is to derive nuclear energy from the fusion of light nuclei. The plasmas generated exhibit both the characteristics of an ordinary fluid and special features caused by the magnetic field. The study of linearized motion in MHD has contributed significantly to the understanding of resistive and nonadiabatic MHD plasma phenomena such as plasma stability, wave propagation and heating.

This problem uses a square $3\,200$ x $3\,200$ real symmetric indefinite matrix, with $18\,316$ entries ($3\,200$ diagonals, $7\,558$ below diagonal, $7\,558$ above diagonal). The presented solver was written for dense systems, therefore, this sparse systems will be treated as a dense system. No special method or data storage was used/done concerning the sparsity of this system.

The first elements of the result vector found for this problem with condition number $2.02 \cdot 10^{+13}$ are presented in Table 29. This table also presents the equivalent infimum-supremum result.

Despite it is an ill conditioned problem, the average diameter of the interval results found by this threaded solver was $1.87 \cdot 10^{-5}$. This is a very accurate result for such an ill conditioned problem.

The execution time for solving this systems of linear equations using the sequential version was 90.80 seconds and with the new multithreaded solution it was 70.03 seconds. The time gain

Table 29 – Results for MHD3200B: Alfven Spectra in Magnetohydrodynamics.

res	Midpoint	Radius	Infimum	Supremum
x[0]	$6.1031675 \cdot 10^{-01}$	$6.7758771 \cdot 10^{-17}$	$6.10316758737 \cdot 10^{-01}$	$6.10316758738 \cdot 10^{-01}$
x[1]	$1.0954139 \cdot 10^{-01}$	$5.3302866 \cdot 10^{-14}$	$1.095413995491 \cdot 10^{-01}$	$1.095413995492 \cdot 10^{-01}$
x[2]	$5.4993982 \cdot 10^{-02}$	$6.1055585 \cdot 10^{-18}$	$5.499398270479 \cdot 10^{-02}$	$5.499398270480 \cdot 10^{-02}$
x[3]	$7.0470230 \cdot 10^{+05}$	$4.221512 \cdot 10^{-07}$	$7.04702304574 \cdot 10^{+05}$	$7.04702304575 \cdot 10^{+05}$

in C computation was around 45% from 48.53 to 26.41 seconds.

8.3 Conclusions

This chapter investigated the use of dedicated threads for speed-up the solver of dense linear systems on dual-core processors. One possible use of dual-core computers when rounding is needed was shown. The idea of separating the calculation with rounding up and rounding down on two separated threads was introduced in order to speed up the verified computation. The same idea can be used for other problems of the same kind.

Experiments showed that it is possible to minimize the performance overhead of changes in the floating-point rounding mode using multi-core machines. Although the inter-thread synchronization primitives could potentially downgrade the overall performance, coarse-grained operations combined with this minimization in the number of changes in the rounding mode can lead to a significantly better performance.

The fine control in the scheduling of threads provided by the Linux kernel (through the processor affinity mechanism described in Section 8.1) allows the use of any number of threads and still guarantees that each available processor will use the minimum number possible of changes in the floating-point rounding mode. We believe that this makes our parallelization method flexible enough to be used in the parallelization of other numerical problems.

9 SIAM 100-Digits Challenge

The idea of this thesis of using optimized libraries for achieving a better performance was also implemented in the C-XSC solver in a joint work with Michael Zimmer [52, 90].

This chapter discusses the usage of a reliable linear system solver to compute the solution of problem 7 of the SIAM 100-digit challenge [1, 83]. Of all the 100-digit challenge problems from SIAM, this is the problem most suitable for our parallel solver since to find the result we have to solve a $20\,000 \times 20\,000$ system of linear equations using interval computations. To perform this task we run our software on the advanced Linux cluster ALiCEnext located at the University of Wuppertal and on the high performance computer HP XC6000 at the computing center of the University of Karlsruhe (XC1).

The main purpose of this research is to demonstrate the power/weakness of the approach of using optimized libraries with C-XSC to solve linear interval systems with a large dense system matrix.

Section 9.1 presents problem 7 of the SIAM 100-Digits Challenge. Section 9.2 describes the parallel verified solution using C-XSC with an accuracy of 16 digits. The results of this implementation are shown in Section 9.3. Section 9.4 presents an extension of the first solution which allows the computation of all the 100 digits for problem 7. Section 9.5 contains some concluding remarks.

9.1 The Problem 7 of the Siam 100-Digit Challenge

Here comes the original problem [83]: Let $A = (a_{ij})$ be the $20\,000 \times 20\,000$ matrix whose entries are zero everywhere except for the primes $2, 3, 5, 7, \ldots, 224\,737$ along the main diagonal and the number 1 in all the positions a_{ij} with $|i - j| = 1, 2, 4, 8, \ldots, 16\,384$. What is the (1, 1) entry of A^{-1}?

Later on we also solve this problem with dimension $n = 50\,000$ and $n = 100\,000$. In the $n = 50\,000$ case the primes from 2 to $611\,953$ are the diagonal entries and the number 1 is in all the positions $|i - j| = 1, 2, 4, 8, \ldots, 32\,768$ and in the $n = 100\,000$ case the primes from 2 to $1\,299\,709$ are the diagonal entries and the number 1 is in all the positions $|i - j| = 1, 2, 4, 8, \ldots, 65\,536$.

9.2 Implementation

Up to now there are no special linear system solvers for sparse system matrices available in C-XSC [32, 44] (but see e.g. [33, 56] for some first approaches). Thus we will solve the Problem 7 of the Siam 100-digit Challenge for dimensions $20\,000$, $50\,000$ and $100\,000$ with brute force using our parallel linear system solver for dense matrices [28, 53, 90]. In case of dimension $n = 20\,000$ the matrix has $400\,000\,000$, in case of $n = 50\,000$ it has $2\,500\,000\,000$ and in case of $n = 100\,000$ it has $10\,000\,000\,000$ entries. We do not store all these entries in one (master) process. Each process generates a matrix MyA, which stores only its part of the matrix, with the help of the function $generateElementOfA$ (a pointer to this function is the first parameter when calling our new solver LSS). Matrices are stored according to the two-dimensional block cyclic distribution used by ScaLAPACK [5]. Internally, the solver creates an approximate inverse R of A and also computes an enclosure (an interval matrix [C]) for $I - R \times A$. Because the solver does not exploit the sparse structure of the problem, these matrices will be dense. Thus, a lot of memory is needed. Nevertheless, using e.g. the cluster computer ALiCEnext at the University of Wuppertal or the high performance computer XC1 at the computing center of the University of Karlsruhe we are able to solve the linear system $Ax = e_1$, where $e_1 = (1, 0 \ldots 0)^T$ denotes the first unit vector. Then, the first component of the solution vector x gives an enclosure of the element at position $(1, 1)$ of A^{-1}, which is the solution of the described problem.

The details on the algorithm used to solve linear systems with interval matrices can be found in Section 2.5.4 [31, 74]. The use of BLAS and LAPACK for computing the verified solution of a linear system is described in Chapter 5 [48]. The parallelization is described in [53, 90]. A different approach (some kind of master-slave) for the parallelization has been used in Chapter 4 [28, 48, 51] storing the complete system matrix (and/or some auxiliary matrices) on a single node (memory bottleneck). The new solver [53, 90] overcomes this difficulty using a two dimensional block cyclic distribution of the matrices according to the idea presented in Chapter 7. It is based on BLAS, MPI [29, 40], ScaLAPACK [5], so called error free transformations [11, 19, 45, 70, 88] and allows very large matrices.

Please note that until now we are only interested in an accurate enclosure of the true result within an interval with floating-point numbers as bounds (i.e. roughly 15 decimal digits accuracy).

9.3 Interval Results and Some Timing Information

Our software has been developed and tested on two very different high performance cluster computers at the Universities of Wuppertal and Karlsruhe (both in Germany), on which also the final computations have been done.

The configuration of these clusters are presented in Table 30.

Table 30 – Cluster Comparison.

	AliceNext	XC1
Location	Wuppertal	Karlsruhe
CPU	AMD Opteron	Intel Itanium2
Clock Frequency	1.8 GHz	1.5 GHz
Number of Nodes	512	128
Number of Cores	2	2
RAM per Node	1 GB	12 GB
Hard Disk	2 x 250 GB	146 GB
Network	6 x Gigabit-Ethernet	Quadrics QsNet II interconnect
BLAS Library	AMD Core Math Library	Intel Math Kernel Library 10.0.011
Compiler	Intel Compiler 9.0	Intel Compiler 10.0

Let us first discuss the results obtained for the original problem ($n = 20\,000$). We find the interval enclosure $A^{-1}_{11} = 0.725078346268401167468687719251160968869180594479508957878164769\ldots$ $\in [0.7250783462684010; 0.7250783462684012]$. This enclosure was reproduced in all cases on both machines even when using different numbers of processors. The value of A^{-1}_{11} is known exactly as a rational number in which the numerator and denominator each have 97 389 decimal digits, see [1] or http://www-m3.ma.tum.de/m3old/bornemann/challengebook/Chapter7.

On AliceNext, solving the problem using 20 processors took about 1 800 seconds and using 50 processors, it took about 870 seconds. On the super computer XC1 in Karlsruhe the corresponding timings were 620 and 280 seconds, respectively.

Table 31 – Execution Time.

	AliceNext	XC1
Data distribution and initialization	27.6	23.7
Compute matrix R	430.1	94.8
Defect iteration	3.8	3.4
Computation of [z]	3.6	4.4
Computation of [C]	402.0	149.2
Verification	2.1	2.3
Overall time needed	869.7	278.2
Average number of exact digits	$1.581288580850802E + 001$	$1.581288580850802E + 001$
Result $A^{-1}[1,1]$	$[7.250783462684010 \cdot 10^{-001}, 7.250783462684012 \cdot 10^{-001}]$	$[7.250783462684010 \cdot 10^{-001}, 7.250783462684012 \cdot 10^{-001}]$

Table 31 presents the execution time of the solver in Wuppertal and Karlsruhe, when executing the linear system solver in case of a system matrix with dimension 20 000 and using 50 processors. Comparing these data gives deeper insight into which operations contribute most to the overall computing times. These main contributions are (as expected) on both machines

- computing the approximation of the inverse R;
- and computing the enclosure [C] of $I - R \times A$.

Computations on the XC1 cluster were a lot faster than on ALICEnext, due to the faster Itanium2 CPUs. However, in the smaller parts $O(n^2)$ ALICEnext is often a little faster than the XC1. The performance of these parts relies heavily on the inlining capabilities of the compiler. Since the Itanium2 processors require a very different compiler architecture, inlining might not yet work as efficient on these systems. Further investigations have to be done in that matter.

We also solved the problem for dimension $50\,000$ and $100\,000$ on the high performance computer in Karlsruhe. For the problem with dimension $50\,000$, we found the interval enclosure:

$A_{11}^{-1} \in [7.250799024199662 \cdot 10^{-001}, 7.250799024199664 \cdot 10^{-001}]$.

Again the computed result does not depend on the number of processors used. The execution times using 50 and 100 processes are $3\,603$ and $1\,958$ sec, respectively.

For the problem with dimension $100\,000$, we found the interval enclosure:

$A_{11}^{-1} \in [7.250806337086472 \cdot 10^{-001}, 7.250806337086474 \cdot 10^{-001}]$.

The time taken to solve this problem with 128 processes was $12\,118$ seconds.

9.4 Solution with 100 Digits

With some small modifications we were also able to compute 100 digits of the exact result, as demanded in the original challenge. To achieve this, we solve the system seven times, setting the right hand side to

$$[b^{(i+1)}] = [b^{(i)}] - A \cdot \text{mid}([x^{(i)}]), \quad i = 1, \dots, 7$$

where $[b^{(1)}]$ is the first unity vector and $[x^{(i)}]$ is an enclosure of the solution of the system with right hand side $[b^{(i)}]$. Since the solver is already able to compute systems with multiple right hand sides, only small modifications were necessary for this approach and only the necessary computations were performed in each step (R and $[C]$ were computed only once). To compute 100 digits of the exact result, the sum

$$\sum_{i=1}^{6} \text{mid}([x_1^{(i)}]) + [x_1^{(7)}]$$

is computed inside the long fixed-point accumulator of C-XSC. We then get the exact result for the first 100 digits of A_{11}^{-1} and dimension $20\,000$:

$$0.72507834626840116746868771925116096886918059447950\\ 8957878164769207773189994596283573592392786478202 0.$$

This computation took 990 seconds on AliceNext using 50 processors, compared to 870 seconds for the computation of the normal interval enclosure.

9.5 Conclusions

Solving the Hundred-Dollar, Hundred-Digit Challenge Problem 7 was not very hard when using our parallel C-XSC solver. Even storing the sparse matrix A as a full matrix, because no special solver for sparse interval systems is available, the method leads to quite acceptable execution times.

The idea described in this thesis of using optimized libraries for improving the verified libraries performance used together with C-XSC presented very promising results. Previously in Chapter 4, we concluded that C-XSC presented significant limitations to achieve high performance in comparison with other possibilities. However, the experiments described in this chapter allying C-XSC with the ideas defended by this thesis, indicate that this strategy allows C-XSC to be used in parallel implementations with very interesting results.

We got bounds for the correct mathematical results. The correctness of these bounds was proved automatically by the computer using (machine) interval calculations. The open source parallel C-XSC solver for linear interval systems is a very powerful tool. It is also able to handle system matrices and right hand sides with real or even with complex interval entries. The numerical results are always reliable.

10 Conclusions

This Chapter concludes this thesis presenting a summary in Section 10.1, followed by Section 10.2 that describes the contributions. Finally, Section 10.3 introduces some ideas of future work.

10.1 Summary

In this thesis, a parallel implementation of a verified method for solving dense linear systems is presented. The idea was to implement an algorithm that uses technologies as MPI communication primitives associated to libraries as LAPACK, BLAS and C-XSC, aiming to provide both self-verification and speed-up at the same time. The algorithms should find a solution even for very ill conditioned problems. This is possible due to the use of extended precision (new algorithm) or scalar product (C-XSC library) when finding the approximation of the inverse.

A first parallel implementation was presented using the C-XSC library [46, 47]. Two main parts of this method, which demand a higher computational cost, were studied, parallelized and optimized. Tests with five granularities were conducted and presented interesting speed-ups. The correctness tests also point out a good implementation, where the changes in the method did not affect the correctness of the verified result. Thus, the gains provided by the verified computing could be kept with a significant decrease of the execution time of the most time consuming steps by using parallelization techniques.

However, the C-XSC parallel method did not achieve the expected overall performance since the solver was not 100% parallelized due to its special variables and optimal scalar product. For large matrices, the execution time of such an algorithm was too large, specially in comparison with the execution time of INTLAB. Therefore, C-XSC did not seem to be the most efficient tool for time critical applications.

A comparison of different numerical tools for solving dense linear systems with (C-XSC and INTLAB) and without (LAPACK) high accuracy was presented in this work. This study was very useful for the choice of which libraries and which method should be used in a new implementation [48].

Sequential algorithms for point and interval input data were written using both infimum-supremum and midpoint-radius arithmetic. The idea was to implement a verified method for solving linear systems using a similar method compared to the C-XSC method, but using the libraries BLAS and LAPACK to achieve better performance. An optimization of the residuum was also implemented based on the INTLAB method. In other words, the new implementations try to join the best aspects of each library avoiding to increase the computational time [48].

Performance tests showed that the midpoint-radius algorithm needs approximately the same time to solve a linear system with point or interval input data, while the infimum-supremum algorithm needs much more time for interval data. This occurs because the interval multiplication must be implemented with case distinctions (each interval multiplication has to be executed differently depending if the interval is positive, negative or if it contains zero) and therefore optimized functions from BLAS/PBLAS cannot be used. Considering that, midpoint-radius arithmetic was the natural choice for the next step of this work: the parallel implementation.

The idea of this parallel implementation was similar to the one used in the sequential solver: high performance and reliability. Thus, the parallel implementation was designed based on the following ideas [48, 50]:

- Compute R using just floating-point operations;
- Input matrix composed by point or interval elements;
- Avoid the use of C-XSC elements that could slow down the execution;
- Use midpoint-radius interval arithmetics and directed roundings;
- Use the fast and highly optimized libraries: PBLAS and ScaLAPACK;
- Matrices stored in the distributed memory according to the two-dimensional block cyclic distribution used by ScaLAPACK.

The performance results showed that it was possible to achieve very good speed-ups in a wide range of processor numbers for large matrix dimensions for both point and interval input data.

The next problem to face was the input matrix dimension barrier, which until that point was around $10\,000$. This limitation was a result of the matrix generation strategy adopted, which concentrates the creation of the input matrix in one node leading very quickly to memory allocation problems. The natural option was to generate sub-matrices of the input matrix individually on each available node, allowing a better use of the global memory since the original matrix was partitioned over several local memories.

These modifications made it possible to overcome the previous input matrix dimension limit from $10\,000$ to $100\,000$. In addition to that, in order to investigate the portability of the proposed solution, tests were performed using two clusters with distinct configurations presenting significant results, indicating that the parallel solver scales well even for very large dense systems over many processors. This is a strong result considering the natural limitations imposed by a distributed memory architecture like clusters, i.e. the very high cost of communication primitives.

Once the main goals of this thesis were achieved, some further investigations were done in two directions:

- Study of the use of dedicated threads for speed-up the solver of dense linear systems on shared memory, specially dual-core processors [49];
- The use of the ideas presented in this thesis to speed-up C-XSC library [52, 91].

The first investigation showed one possible use of dual-core computers when rounding is needed. The idea of separating the calculation with rounding up and rounding down on two separated threads was introduced in order to speed up the verified computation. Experiments showed that it is possible to minimize the performance overhead of changes in the floating-point rounding mode using multi-core machines.

The idea of this thesis of using optimized libraries for achieving a better performance was also implemented in the C-XSC solver and used to compute the solution of problem 7 of the SIAM 100-digit challenge. Tests showed that solving Problem 7 was not very hard when using the parallel C-XSC solver. Even when storing the sparse matrix A as a full matrix (because no special solver for sparse interval systems is available) the method leads to quite acceptable execution times.

10.2 Contributions

The main contribution of this thesis was to provide a free, fast, reliable and accurate solver for point and interval dense linear systems increasing the use of verified computing through its optimization and parallelization. The idea was to implement a solver for dense linear systems using a verified method, interval arithmetic and directed roundings based on MPI communication primitives associated to optimized libraries, aiming to provide both self-verification and speed-up at the same time. This is an important contribution since without parallel techniques verified computing becomes the bottleneck of an application.

Additionally, other contributions can be enumerated:

1. The development of a verified parallel solver for dense linear systems using C-XSC;
2. The proposal and implementation of a new sequential verified solver for dense linear systems for point and interval input data using both infimum-supremum and midpoint-radius arithmetic based on highly optimized libraries (BLAS/LAPACK);
3. The development of a new parallel verified solver for point and interval dense linear systems using midpoint-radius arithmetic, directed roundings and optimized libraries (PBLAS/ScaLAPACK);

4. Investigation of the portability of the solution over 3 different clusters in Germany (ALiCEnext, XC1 and IC1);
5. Solution of very large dense linear systems, exploiting the scalability of the solution;
6. Experimental implementation of the solver for dense linear systems on dual-core processors;
7. Integration of the ideas presented in this thesis to speed up C-XSC library, using it to compute the solution of problem 7 of the SIAM 100-digit challenge;
8. Publication of many papers in conferences [46–50, 52, 91];
9. Publication of a paper in the International Journal of Parallel Programming [51].

10.3 Future Work

Among the ideas for future work we will concentrate in the following two main investigations:

- Parallelization of a verified method for solving sparse matrices;
- Join the concepts of verified computing and test of software in two different approaches:
  - Development of a test strategy for verified libraries;
  - Use of a verified library as a tool for testing numerical softwares.

Since many real problems are modeled using sparse systems, the parallelization of a verified method for solving sparse matrices appears as a natural future work. It would be interesting to implement methods that take advantage of matrix sparsity, making possible to solve problems with huge dimensions.

Another future work would be to join the concepts of verified computing and test of software. This could be done in two different approaches. The first is to propose and implement a method to perform automated tests in a verified library for solving linear systems in both sequential and parallel environments to ensure that it will be working correctly in different platforms and operating systems. The idea is to be able to provide a method that can be also used for creating tests for numerical software in general.

The second approach would be to use this library in a semi automatic way, to verify the proper functioning of programs that use mathematical libraries without verification. The main idea is to get a sample of calculation parts, and recalculate them using a verified algorithm which allows the verification of the previous calculated results. This sampling will be done very carefully, because in one hand checked computing is much more costly in terms of processing, but in the other hand it provides the guarantee that the correct result will be found.

References

[1] *The SIAM 100-Digit Challenge - A Study in High-Accuracy Numerical Computing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.

[2] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983.

[3] M. Baboulin, L. Giraud, and S. Gratton. A Parallel Distributed Solver for Large Dense Symmetric Systems: Applications to Geodesy and Electromagnetism Problems. *International Journal of High Speed Computing*, 19(4):353–363, 2005.

[4] Z. Bai, D. Day, J. Demmel, and J. Dongarra. A Test Matrix Collection for Non-Hermitian Eigenvalue Problems. Technical Report UT-CS-97-355, Knoxville, TN, USA, 1997.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[6] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers Design Issues and Performance. In *SUPERCOMPUTING '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 5.1–5.22, Pittsburgh, USA, 1996. IEEE Computer Society.

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

[8] G. Bohlender. What Do We Need Beyond IEEE Arithmetic? *Computer Arithmetic and Self-validating Numerical Methods*, pages 1–32, 1990.

[9] G. Bohlender. Faster Interval Computations Through Minimized Switching of Rounding Modes. Technical Report UKA-01-2002, Universität Karlsruhe, Karlsruhe, Germany, 2002.

[10] G. Bohlender, M. Kolberg, and D. Claudio. Modifications to Expression Evaluation in C-XSC. Technical Report BUW-WRSWT 2005/5, Universität Wuppertal, Wuppertal, Germany, 2005. Presented at SCAN04, Fukuoka, Japan.

[11] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. Semantics for Exact Floating Point Operations. In *ARITH ´91: Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 26–28, Grenoble, France, 1991. IEEE Society Press.

[12] J. G. L. Booten, P. M. Meijer, H. J. J. te Riele, and H. A. van der Vorst. Parallel Arnoldi Method for the Construction of a Krylov Subspace Basis: An Application in Magnetohydrodynamics. In *High-Performance Computing and Networking*, volume 797 of *Lecture Notes in Computer Science*, pages 196–201, Berlin, 1994. Springer-Verlag.

[13] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] O. Caprani. Implementation of a Low Round-off Summation Method. *BIT Numerical Mathematics*, 11(3):271–275, 1971.

[15] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, and D. Walker. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, volume 1041 of *Lecture Notes in Computer Science*, pages 107–114, Germany, 1996. Springer Berlin/Heidelberg.

[16] D. M. Claudio and J. M. Marins. *Cálculo Numérico Computacional: Teoria e Prática*. Editora Atlas S. A., São Paulo, 2000.

[17] Program Committee. Description and Rationale of Validated Computing 2002. http://interval.louisiana.edu/conferences/Validated_computing_2002/html_notice.html. visited in 09th February 2009.

[18] P. W. de Oliveira, T. A. Divério, and D. M. Claudio. *Fundamentos da Matemática Intervalar*. Editora Sagra Luzzatto, Porto Alegre, 2005.

[19] T. J. Dekker. A Floating-point Technique for Extending the Available Precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[20] J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[21] J. Dongarra, R. Pozo, and D.W. Walker. LAPACK++: A Design Overview of Object-oriented Extensions for High Performance Linear Algebra. In *SUPERCOMPUTING ’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 162–171, Portland, USA, 1993. IEEE Computer Society Press.

[22] J. Dongarra and D. Walker. LAPACK Working Note 58: The Design of Linear Algebra Libraries for High Performance Computers. Technical Report UT-CS-93-188, Knoxville, TN, USA, 1993.

[23] J. Dongarra and R. Whaley. BLACS User´s Guide V1.1. Technical Report UT-CS-95-281, Knoxville, TN, USA, 1995.

[24] I. S. Duff and H. A. van der Vorst. Developments and Trends in the Parallel Solution of Linear Systems. Technical Report RAL TR-1999-027, CERFACS, Toulouse, France, 1999.

[25] A. Facius. *Iterative Solution of Linear Systems with Improved Arithmetic and Result Verification*. PhD thesis, University of Karlsruhe, Germany, 2000.

[26] P. Gonzalez, J. C. Cabaleiro, and T. F. Pena. Solving Sparse Triangular Systems on Distributed Memory Multicomputers. In *PDP '98: Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 470–478, Madrid, Spain, 1998. IEEE Computer Society Press.

[27] R. T. Gregory and D. L. Karney. *A Collection of Matrices for Testing Computational Algorithms*. Wiley-Interscience, New York, 1969.

[28] M. Grimmer. *Selbstverifizierende Mathematische Softwarewerkzeuge im High Performance Computing. Konzeption, Entwicklung und Analyse am Beispiel der Parallelen Verifizierten Lösung linearer Fredholmscher Integralgleichungen Zweiter Art*. PhD thesis, University of Wuppertal, Germany, 2007.

[29] M. Grimmer and W. Krämer. An MPI Extension for Verified Numerical Computations in Parallel Environments. In *CSC'07: Proceedings of the International Conference on Scientific Computing*, pages 111–117, Las Vegas, USA, 2007.

[30] H. Haefner. Institutscluster User Guide. http://www.rz.uni-karlsruhe.de/rz/docs/IC/ug/ugic.pdf. visited in 09th February 2009.

[31] R. Hammer, D. Ratz, U. Kulisch, and M. Hocks. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Springer-Verlag New York, Secaucus, NJ, USA, 1997.

[32] W. Hofschuster and W. Krämer. C-XSC 2.0: A C++ Library for Extended Scientific Computing. In *Numerical Software with Result Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 15–35, Germany, 2004. Springer Berlin/Heidelberg.

[33] C. Hölbig, W. Krämer, and T. Diverio. An Accurate and Efficient Selfverifying Solver for Systems with Banded Coefficient Matrix. In *PARCO 2003: Proceedings of Parallel*

Computing: Software Technology, Algorithms, Architectures and Applications, pages 283–290, Germany, 2003. Elsevier Science B.V. Amsterdam.

[34] C. A. Hölbig, P. S. Morandi Júnior, B. F. K. Alcalde, and T. A. Diverio. Selfverifying Solvers for Linear Systems of Equations in C-XSC. In *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer Science*, pages 292–297, Germany, 2004. Springer Berlin / Heidelberg.

[35] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1985.

[36] IEEE-SA. IEEE to Create Standard for Interval Arithmetic. http://standards.ieee.org/announcements/intervalarith.html. visited in 09th February 2009.

[37] Intel. Intel® Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm. visited in 09th February 2009.

[38] INTLAB. INTerval LABoratory. http://www.ti3.tu-harburg.de/ rump/intlab/. visited in 09th February 2009.

[39] W. Kahan. Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic. http://www.cs.berkeley.edu/ wkahan/ieee754status/ieee754.ps. visited in 09th February 2009.

[40] G. Karniadakis and R. M. Kirby II. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, 2003.

[41] D. Kaya and K. Wright. Parallel Algorithms for LU Decomposition on a Shared Memory Multiprocessor. *Applied Mathematics and Computation*, 163(1):179–191, 2005.

[42] R. B. Kearfott. Interval Computations: Introduction, Uses and Resources. *Euromath Bulletin*, 2(1), 1996.

[43] T. Kersten. *Verifizierende Rechnerinvariante Numerikmodule*. PhD thesis, University of Karlsruhe, Germany, 1998.

[44] R. Klatte, U. Kulisch, C. Lawo, R. Rauch, and A. Wiethoff. *C-XSC- A C++ Class Library for Extended Scientific Computing*. Springer-Verlag Berlin, 1993.

[45] D. E. Knuth. *The Art of Computer Programming Vol.2 - Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1997.

[46] M. Kolberg, L. Baldo, P. Velho, L. G. Fernandes, and D. Claudio. Optimizing a Parallel Self-verified Method for Solving Linear Systems. In *PARA 2006: Applied Parallel*

Computing. State of the Art in Scientific Computing, volume 4699 of *Lecture Notes in Computer Science*, pages 949–955, Germany, 2008. Springer Berlin / Heidelberg.

[47] M. Kolberg, L. Baldo, P. Velho, T. Webber, L. G. Fernandes, P. Fernandes, and D. Claudio. Parallel Selfverified Method for Solving Linear Systems. In *VECPAR 2006: 7th International Meeting on High Performance Computing for Computational Science*, pages 179–190, Rio de Janeiro, Brazil, 2006.

[48] M. Kolberg, G. Bohlender, and D. Claudio. Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computation. In *VECPAR 2008: High Performance Computing for Computational Science*, volume 5336 of *Lecture Notes in Computer Science*, pages 13–26, Germany, 2008. Springer Berlin / Heidelberg.

[49] M. Kolberg, D. Cordeiro, G. Bohlender, L. G. Fernandes, and A. Goldman. A Multithreaded Verified Method for Solving Linear Systems in Dual-Core Processors. In *PARA - 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing* , To be published, Lecture Notes in Computer Science, Trondheim, Norway, 2008. Springer Berlin / Heidelberg.

[50] M. Kolberg, M. Dorn, G. Bohlender, and L. G. Fernandes. Parallel Verified Linear System Solver for Uncertain Input Data. In *SBAC-PAD: 20th International Symposium on Computer Architecture and High Performance Computing*, pages 89–96, Campo Grande, Brazil, 2008. IEEE Computer Society Press.

[51] M. Kolberg, L. G. Fernandes, and D. Claudio. Dense Linear System: A Parallel Selfverified Solver . *International Journal of Parallel Programming*, 36:412–425, 2008.

[52] M. Kolberg, W. Krämer, and M. Zimmer. A Note on Solving Problem 7 of the SIAM 100-Digit Challenge Using C-XSC. In *Numerical Validation in Current Hardware Architectures*, to be published, Lecture Notes in Computer Science, Germany, 2008. Springer Berlin / Heidelberg.

[53] W. Krämer and M. Zimmer. Fast (Parallel) Dense Linear Interval Systems Solving in C-XSC using Error Free Transformations and BLAS. In *Numerical Validation in Current Hardware Architectures*, to be published, Lecture Notes in Computer Science, Germany, 2008. Springer Berlin / Heidelberg.

[54] U. Kulisch and W. L. Miranker (Eds.). A New Approach to Scientific Computation. *SIAM Review*, 27(2):267–268, 1985.

[55] U. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, New York, 1981.

[56] W. Krämer, U. Kulisch and R. Lohner. *Numerical Toolbox for Verified Computing II: Advanced Numerical Problems*. SpringerVerlag, 2006.

[57] LAPACK. Linear Algebra Package. http://www.cs.colorado.edu/ jessup/lapack/. visited in 09th February 2009.

[58] P. Linz. Accurate Floating-point Summation. *Communications of the ACM*, 13(6):361–362, 1970.

[59] Z. Liu and D. W. Cheung. Efficient Parallel Algorithm for Dense Matrix LU Decomposition with Pivoting on Hypercubes. *Computers & Mathematics with Applications*, 33(8):39–50, 1997.

[60] G. C. Lo and Y. Saad. Iterative Solution of General Sparse Linear Systems on Clusters of Workstations. Technical Report UMSI-96-117, University of Minnesota, Minneapolis, USA, 1996.

[61] R. Love. *Linux Kernel Development*. SAMS, Developer Library Series, 2003.

[62] P. Markstein. The New IEEE-754 Standard for Floating Point Arithmetic. In *Numerical Validation in Current Hardware Architectures*, to be published, Lecture Notes in Computer Science, Germany, 2008. Springer Berlin / Heidelberg.

[63] MathWorks. *MATLAB, The Language of Technical Computing*. The MathWorks, Inc., 2001.

[64] C. R. Milani. Estudo Sobre a Aplicação da Computação Paralela na Resolução de Sistemas Lineares. Monografia de Introdução à Pesquisa I, Pontifícia Universidade Católica do Rio Grande do Sul, 2008.

[65] W. L. Miranker and R. A. Toupin, editors. *Accurate Scientific Computations*, volume 235 of *Lecture Notes in Computer Science*, Germany, 1986. Springer-Verlag Berlin.

[66] Intel Software Network. Introduction to Microarchitectural Optimization for Itanium Processors. Technical Report 251464-001, Intel Corporation, USA, 2002.

[67] American National Standards Institute / Institute of Electrical and Eletronics Engineers. A Standard for Binary Floating-point Arithmetic. *ANSI/IEEE*, Std.754-1985, 1985.

[68] Microprocessor Standards Committee of the IEEE Computer Society. *IEEE Std 754-2008 (Revision of IEEE Std 754-1985)*. IEEE Computer Society Press, New York, USA, 2008.

[69] T. Ogita, S. M. Rump, and S. Oishi. Accurate Sum and Dot Product with Applications. In *CACSD 2004: IEEE International Symposium on Computer Aided Control Systems Design*, pages 152–155, Taipei, Taiwan, 2004. IEEE Computer Society Press.

[70] T. Ogita, S. M. Rump, and S. Oishi. Accurate Sum and Dot Product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[71] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc, San Francisco, USA, 1997.

[72] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Companies, Inc, New York, USA, 2004.

[73] J. Rohn. Systems of Linear Interval Equations. *Linear Algebra and its Applications*, 126:39–78, 1989.

[74] S. M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, University of Karlsruhe, Germany, 1980.

[75] S. M. Rump. Solving Algebraic Problems with High Accuracy. In *Proceedings of the Symposium on a New Approach to Scientific Computation*, pages 51–120, San Diego, CA, USA, 1983. Academic Press Professional, Inc.

[76] S. M. Rump. Convergence Properties of Iterations Using Sets. Technical Report 15(6):427-432, TU Leipzig, 1991.

[77] S. M. Rump. Fast and Parallel Interval Arithmetic. *Bit Numerical Mathematics*, 39(3):534–554, 1999.

[78] S. M. Rump. INTLAB - INterval LABoratory. In T. Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publisher, 1999.

[79] M. Snir, S. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[80] S. Stark and A. N. Beris. LU Decomposition Optimized for a Parallel Computer with a Hierarchical Distributed Memory. *Parallel Computing*, 18(9):959–971, 1992.

[81] P. Stpiczynski and M. Paprzycki. Numerical Software for Solving Dense Linear Algebra Problems on High Performance Computers. In *APLIMAT 2005: Proceedings of the 4th International Conference on Applied Mathematics*, pages 207–218, Bratislava, Slovak Republic, 2005.

[82] TOP500 supercomputing site. TOP500 List of the Worlds Most Powerful Supercomputers. http://www.top500.org. visited in 09th February 2009.

[83] N. Trefethen. *A Hundred-Dollar, Hundred-Digit Challenge*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[84] N. K. Tsao. The Accuracy of a Parallel LU Decomposition Algorithm. *Computers & Mathematics with Applications*, 20(7):25–30, 1990.

[85] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[86] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, T. Jacob, S. Jain, C.and Hoskote Y. Erraguntla, V.and Roberts, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[87] A. Wiethoff. *Verifizierte Globale Optimierung auf Parallelrechnern*. PhD thesis, University of Karlsruhe, Germany, 1997.

[88] N. Yamanaka, T. Ogita, S. M. Rump, and S. Oishi. A Parallel Algorithm of Accurate Dot Product. *Parallel Computing*, 34(6-8):392–410, 2008.

[89] J. Zhang and C. Maple. Parallel Solutions of Large Dense Linear Systems Using MPI. In *PARELEC '02: Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, pages 312–317. IEEE Computer Society Press, 2002.

[90] M. Zimmer. Laufzeiteffiziente, Parallele Löser für Lineare Intervallgleichungssysteme in C-XSC. Master's thesis, Universität Wuppertal, 2007.

[91] M. Zimmer, M. Kolberg, and W. Krämer. Efficient Parallel Solvers for Large Dense Systems of Linear Interval Equations. In *SCAN 2008: 13th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations*, El Paso, Texas, USA. To be published.