Parallel Implementation of Data Balancing Algorithms

Nelson Filipe Lopes da Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Dr. José Carlos Alves Pereira Monteiro
Prof. Dr. João Filipe Dias Rodrigues

Examination Committee
Chairperson: Prof. Dr. João António Madeiras Pereira
Supervisor: Prof. Dr. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Dr. Juan Antonio Acebrón Torres

May 2016
Acknowledgments

Since the very beginning, this work counted with the help of various people. It is possible to say that, without those people, this work would not be possible. It is also important to note that, help did not always came from a technical perspective, but also from a word of encouragement and motivation.

I would like to thank professors José Monteiro and João Rodrigues, for the guidance, continuous help, advice and availability. During the development of this work their enthusiasm and ideas were very motivating and challenging.

Similarly, I would like to show my gratitude towards my course and work colleagues, for the exchange of ideas and motivational words in frustration times.

Last, but not least, I would like to express my gratitude towards my dear parents, brother and girlfriend. They were tireless during all my academic progress, providing an excellent environment so I could achieve this milestone and be prepared for future challenges.
Abstract

The goal of this thesis is to describe our work on the development of a parallel implementation for the Input-Output data balancing algorithm of Rodrigues. This application aims to out-perform sequential methods, providing quicker solutions. An analysis is made to find the most computational challenge operations of this algorithm, and discusses possible optimization points.

To support our implementation we use PETSc, a parallel library which provides highly optimized routines for linear algebra using vectors and matrices. We describe in detail how this tool is used to fulfill our requirements.

This document presents the principal problems found during development, addressing the different solutions. Scalability was the main issue in this project. To overcome this problem we experimented with processor load balancing and matrix permutation.

The performance was evaluated in a computer cluster, logging and comparing execution times for various test problems. Real-life, as well as artificial problems were used. The input data is provided in Octave files. The application loads this input file, creating and distributing the data across all machines.

Various matrix permutations are considered and computational scalability are tested for different matrix structures. To do this we executed our application, activating or deactivating such optimizations at run-time.

Keywords: Input-Output Analysis; data balancing; parallel computing; PETSc
Resumo

Esta tese descreve a análise e implementação em paralelo do algoritmo de balanceamento de informação Input-Output de Rodrigues. Serão apresentados os desafios computacionais deste algoritmo, através da discussão de possíveis pontos de otimização.

A implementação é feita utilizando PETSc, uma biblioteca computacional que oferece implementações paralelas das várias operações algébricas, entre matrizes e vetores. É feita igualmente uma descrição detalhada da utilização deste pacote.

Este documento apresenta as principais dificuldades no desenvolvimento da solução, com especial atenção à escalabilidade da mesma. A solução encontrada experimenta uma melhor distribuição de carga e permutação de matrizes, no sentido de obter melhor resultados computacionais.

Os testes decorreram através da utilização de um cluster de computadores, calculando tempos de execução para diferentes casos de teste. Adicionalmente, para uma melhor abrangência, foram utilizados casos de estudo reais e artificiais. Estes testes são carregados de ficheiros Octave existentes, sendo a informação posteriormente distribuída pelas diferentes máquinas.

São consideradas várias permutações, de forma a testar diferentes estruturas de matriz. Neste sentido, diferentes optimizações são ativas ou desativas durante sua execução.

Palavras-Chave: Analise de Input-Output; balanceamento de informação, computação paralela; PETSc
## Contents

**List of Tables** xi

**List of Figures** xiii

**Acronyms** xv

1 **Introduction** 3
   1.1 Project Objectives ........................................ 3
   1.2 Proposed Solution ........................................ 4
   1.3 Challenges ............................................. 4
   1.4 Dissertation Outline .................................... 4

2 **Problem Definition** 5
   2.1 Input-Output Analysis ...................................... 5
   2.2 Data Balancing ........................................... 6
   2.3 Computational Challenges ................................ 8
   2.4 Summary .................................................. 9

3 **Related Work** 11
   3.1 Data Storage .............................................. 11
   3.2 Storage Formats for Sparse Matrices .................... 11
       3.2.1 Coordinate Format (COO) .......................... 12
       3.2.2 Compressed Sparse Rows/Columns (CSR/CSC) ..... 12
   3.3 Solving a Linear System .................................. 13
       3.3.1 Rate of Convergence ............................... 14
       3.3.2 Jacobi Method ...................................... 14
       3.3.3 Gauss-Seidel ....................................... 15
       3.3.4 GMRES ............................................. 15
       3.3.5 Conjugate Gradient ................................. 15
   3.4 Software Packages .......................................... 16
   3.5 Summary .................................................. 17

4 **Materials and Methods** 19
   4.1 Operations Required ....................................... 19
       4.1.1 Computing Matrix A .................................. 20
       4.1.2 Computing Left Hand and Error ................. 20
       4.1.3 Hierarchical Cycle ................................. 21
   4.2 Object Distribution and Assembly ....................... 22
       4.2.1 Matrix construction ............................... 22
List of Tables

2.1 Input-Output (IO) table for a two sector economy. ............................................. 6
3.1 Comparative view between libraries. ................................................................. 17
6.1 Comparative view between problem topologies. .............................................. 37
List of Figures

4.1 Sparse matrix distributed across four processors. ........................................ 23
4.2 Sparse matrix distributed across two processors. ...................................... 23
4.3 Vector divided across four machines. ......................................................... 23
4.4 Vector divided across two machines. ......................................................... 24
5.1 Sparse bi-diagonal rectangle matrix. ............................................................ 30
5.2 Sparse bi-diagonal rectangle matrix divided across two processors, A and B. .. 30
5.3 Permuted sparse bi-diagonal rectangle matrix. .............................................. 30
6.1 Different problems topologies used. ............................................................ 36
Acronyms

BEA  Bureau of Economic Analysis. 36
BLAS  Basic Linear Algebra Subroutines. 16
CG  Conjugate Gradient. 14, 16, 17, 19, 25, 29, 30
COO  Coordinate Format. 12
CSC  Compressed Sparse Columns. 12, 13
CSR  Compressed Sparse Rows. 12, 13
EEIO  Environmentally-extended input-output. 3, 5, 6
GMRES  Generalized Minimal RESidual. 14, 15, 17, 19, 25, 29, 30
GS  Gauss-Seidel. 15
GSL  GNU Scientific Library. 16
IO  Input-Output. XI, 3–8, 11, 37, 39, 41
IOA  Input-Output Analysis. 3, 5, 9
IWLS  Iterative Weighted Least-Squares. 7–9, 20, 21, 29
JOR  Jacobi Overrelaxation. 14, 15
LAMA  Library of Accelerated Math Applications. 17
LCA  Life Cycle Assessment. 9
MINRES  Minimal RESidual. 15
MIT  Massachusetts Institute of Technology. 5
MPI  Message Passing Interface. 4, 17, 35
PETSc  Portable Extensible Toolkit for Scientific Computation. 4, 17, 19, 24, 26–29, 31, 41
PSBLAS  Parallel Sparse Basic Linear Algebra Subroutines. 16
QCEW  Quarterly Census of Employment and Wage. 8, 36
RCM Reverse Cuthill-McKee. 31

SAM Social-Accounting Matrix. 5 6

SOR Successive Over-Relaxation. 15

spBLAS Sparse Basic Linear Algebra Subprograms. 16

WLS Weighted Least-Squares. 7 41
Chapter 1

Introduction

The need for information affects multiple areas in modern society. Disciplines, such as policy analysis, consumes vast quantities of data to study location patterns. In industry, researchers process large chunks of data to find competitive advantages. In environment, as global population continues to grow, Environmentally-extended input-output (EEIO) creates a need for data to evaluate environment impacts, and economic drivers for global carbon and nitrogen emissions, and natural resources footprints can be identified. Input-Output Analysis (IOA) is yet another area demanding large data-sets to update regional IO tables, predicting how goods are used in different economic sectors.

Problems rise when some of these values are suppressed, due to confidentiality or surveys issues. Moreover, data may also contain value inconsistencies, making these analysis more difficult. In response to this issue, methods for values estimation were developed. Indeed, using available values constraints in faulty data, such methods are capable of estimate missing values, and reconcile inconsistencies. However, these techniques often requires dealing with large amounts of information, which naturally result in an increase of computational time.

In response to this issue, computer clusters and multi-processor machines become attractive, as these provide elevated computational power, compared with ordinary machines. Indeed, parallel computing takes advantage of such architectures in an attempt to provide fast solutions to problems demanding high computational power.

Hence, the present work is concerned with implementation, in the context of parallel/distributed computing, of algorithms that deal with uncertainty in balancing IOA data. This field deals with the compilation of data which describes the interactions among different components of the economy and the environment and, among other applications, with the assessment of the impact of a demand stimulus on carbon emissions. The guiding line connecting the algorithms implemented in this thesis is that they deal with the estimation and propagation of uncertainty, in large sparse systems.

1.1 Project Objectives

The main objective of this work is to speedup the computational process, finding quicker solutions, comparing with sequential methods. That said, in a nutshell, the goal of this work is to provide parallel implementations of algorithms to solve problems of such dimensions. Moreover, the new parallel algorithms should be portable across computer clusters, and properly scale for real life IO data.

Also important is the group of problems which can be solved by our method. In other words, it is necessary to perform an analysis to find which problems work best in the parallelization technique. Since different topologies offer different matrix structures, data division and parallelization will change.
1.2 Proposed Solution

Currently there exists computer software capable of balancing data, implemented in high level languages, such as Octave. However, these solutions are limited to small systems, since it shows slow results for bigger systems. Furthermore, current solutions do not implement parallel operations, making it impossible to run many operations at the same time.

We propose a solution based on distributed memory architectures, as these provide the needed computational resources for this project. Implementation is done using Message Passing Interface (MPI) sending data across nodes within the cluster.

Objects, such as matrices and vectors, are divided by chunks of contiguous rows and sub-vectors, respectively. The algorithm proceeds by balancing input data, saving the final solution in an output file. To accomplish this, we use Portable Extensible Toolkit for Scientific Computation (PETSc) [1], an open-source library which provides parallel implementation of basic operations between matrices and vectors. Due to its dimensions, matrices are stored in sparse format, reducing memory requirements and entries to process.

Furthermore, our system is capable of permuting matrix rows. Some input matrices may present non-zero pattern, that impede parallel operations or demand high number of messages to be exchanged. Hence, reshaping matrix structures can potentially increase performance in some cases.

1.3 Challenges

Parallelization of algorithms raises problems sequential implementations do not face. For instance, consider memory allocation and object creation. In traditional sequential solutions, large chunks of contiguous memory may be used, storing complete matrices and vectors. However, in parallel environments, data must be divided and distributed across machines. This task is crucial since all processors should have identical workloads for an optimal performance. Finding the best object division is a challenge, as different problems may require different approaches.

Another interesting aspect of parallel computing is the communication between machines. Since algorithms run in multiple machines, these require messages to be exchanged across the network. Considering that communication costs are considerable higher, comparing with processing times, this task introduces overhead to the overall process. Algorithms must aim for homogeneous object distribution and low communication overheads.

1.4 Dissertation Outline

The present thesis is organized in the following structure: Chapter 2 exposes the problem being addressed in this work, providing the necessary background for a better understanding of future chapters. In Chapter 3 we propose theoretical state-of-the-art solutions for the problems presented in Chapter 2. Chapter 4 summaries the materials and methods used to implement such solutions, discussing potential optimization algorithm points. Next, Chapter 5 presents our issues during implementation, discussing respective causes and solutions. Lastly, Chapters 6 and 7 discuss evaluation and final conclusions respectively.
Chapter 2

Problem Definition

In this chapter we introduce to the problem of balancing IO data, starting by presenting the Leontief Model of IOA and its purpose in economic analysis. After, we present its applications and different application areas. Further, we propose a data balancing algorithm, mathematically defining how it operates and how data is manipulated in computation.

Finally, the chapter ends discussing the computational challenges in balancing such data structures. We analyze possible optimization points and computational heavy operations in the proposed algorithm.

2.1 Input-Output Analysis

IOA was developed by Professor Wassily Leontief in the 1930s, as a framework to analyze interdependence of economic sectors [2]. The IO model is often referred as Leontief Model for this same reason. In its core, an IO model is a system of Linear Equations where, each represents the distribution of a given product through the economic sectors. In other words, it consists in a table showing how economy sectors’s products (outputs) is used in other economy sectors (inputs).

Leontief attempted to manually perform the first IOA using a table representation of the US Economy in 1919. However, since it consisted in a 42-sector IO table, the required equation solving was performed by a calculation machine from John Wilbur at the Massachusetts Institute of Technology (MIT) [3].

Leontief model also been extended to other frameworks, dealing with employment and social account data, regarding industrial production, international and inter-regional flow of products and services. Furthermore it is also used in frameworks, analyzing energy consumption and environment pollution [2].

IOA may be extended to problems of different nature. For instance, consider the conciliation and update of Social-Accounting Matrix (SAM) matrices, which require coherent data. These tables often suffer from deficient row and column sums, i.e, these values do not add [4]. Moreover, IO tables may also present incomplete data, as some values may be missing. This may be due to various reasons, varying from inefficient industry surveys, as these may be costly, or suppression of confidential data [5]. In these cases, IOA may be used as a technique for both, estimate suppressed values and balance rows and columns sums in SAM.

EEIO is yet another field in which IOA is used to study relations between economic consumption activities and environment impacts [6]. Consider the consumption of beef in the modern society. In order to produce beef, inputs such as water and animal food must be consumed. Additionally, to provide water and food to, other activities must take place, such as the construction of water delivery structures, and food plantations. Continuing the same reasoning, it is possible to estimate global impacts, such as carbon, water and nitrogen footprints involved in beef production [6].
Table 2.1: IO table for a two sector economy.

<table>
<thead>
<tr>
<th></th>
<th>Ag</th>
<th>Ma</th>
<th>$x^f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ag</td>
<td>8</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>Ma</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>$x^c$</td>
<td>12</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Data Balancing

The specific problem being addressed in this work is the balancing of IO tables, such as SAM, with arbitrary structure, uncertainty estimates and multiple data source.

To further illustrate this idea, consider the example of the IO Table 2.1 based on EEIO explanation [6]. This table accounts for a two sector economy - Agriculture (Ag) and Manufacture (Ma) - where each sector sells and buys goods from the other.

In this example, business within sector Ag bought $4 worth of goods and services from business in sector Ma, and business in sector Ag sold $8 worth of services and goods for the same sector. Furthermore, $x^f$ and $x^c$ are the rows and columns sums, respectively. Note that the theses values must be coherent with one another, meaning that, column and row sums must add up to matrix values total.

From this example, information can be divided in two vectors and a matrix, as expressed in Equation 2.1:

$$Z = \begin{bmatrix} 8 & 5 \\ 4 & 2 \end{bmatrix}, \quad x_r = \begin{bmatrix} 13 \\ 6 \end{bmatrix}, \quad x_c = \begin{bmatrix} 12 \\ 7 \end{bmatrix}. \quad (2.1)$$

Essentially, to assure data consistency, IO values can be arranged in a single vector, as stated in Vector 2.2.

$$t = \begin{bmatrix} Z_{00} \\ Z_{01} \\ Z_{10} \\ Z_{11} \\ x^r_0 \\ x^r_1 \\ x^c_0 \\ x^c_1 \end{bmatrix} \quad (2.2)$$

where $Z_{ij}$ is the value of $Z$ at row $i$ and column $j$, and $t$ is the IO data.

This representation allows for a better manipulation of data, in a sense that, using the aggregation matrix $G$, it is possible to assure data consistency, by verifying if $G \times t = 0$.

As such, to obtain this effect in our example, $G$ is constructed as follows.

$$G = \begin{bmatrix} 1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix}, \quad t = \begin{bmatrix} 8 \\ 5 \\ 4 \\ 2 \\ 13 \\ 6 \\ 12 \\ 7 \end{bmatrix}, \quad h = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 3 \\ 3 \end{bmatrix}, \quad u = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.01 \\ 0.01 \\ 0.001 \end{bmatrix} \quad (2.3)$$

Furthermore, since $t$ contains data from $Z$, $x^f$ and $x^c$, three data sources are used. Additionally,
using data uncertainty for each data point would result in the data structures presented in Equation 2.3.

In summary, a complete balance problem is a system composed of objects: aggregation matrix $G$; initial IO data vector $\mu$; IO data quality rank vector $h$; IO data uncertainty vector $u$.

According to the literature, an arbitrary structure is formulated by IO data arranged in a vector $t$ of size $n_t$, and $n_k$ accounting identities, described by [4]:

$$0 = \sum_{j=1}^{n_t} G_{ij} t_j + k_i$$ (2.4)

where $G$ is an aggregation matrix, whose entries are either 0, 1 or -1, and $k$ is a vector of numerical constrains. Additionally, accounting identities can be represented in matrix form as follows:

$$0 = G t + k$$ (2.5)

However, in practice, the IO data is unbalanced, meaning that $t$ is unknown, and what is available is a vector $\mu$ such that:

$$0 \neq G \mu + k$$ (2.6)

In a nutshell, the objective is to compute $t$ using $\mu$, so Equation (2.5) is satisfied.

Initially, $G$, $\mu$ and $k$ are known objects, previously computed from IO data tables. Additionally, since IO data is also subject to uncertainty due to empirical measurements. Hence, it is also known an additional vector $u$ of size $n_t$, which describes the relative uncertainty for each IO data unit.

Finally, to account for multiple data sources and conflicting information, a vector of size $n_t$ is also available at input. Named $h$, this vector describes the rank of information quality for each IO data point.

In this project, this problem will be addressed using the Weighted Least-Squares (WLS) algorithm of Rodrigues [4]. The basic idea is to use $h$ to dictate which values will be changed and which will stay constant. The algorithm starts by changing low quality rank values, and use high quality values as constraints. If no solution is found, the algorithm will evolve and adjust high quality data, computing successive result approximations in an Iterative Weighted Least-Squares (IWLS) process. Once the computation starts, the algorithm sets $t^{(0)} = \mu$ and solves the following [4]:

$$t^{(n+1)} = t^{(n)} + \hat{s}^{(n)} G^T \alpha^{(n)}$$ (2.7)

where $G^T$ denotes the transpose of $G$ and $\hat{s}^{(n)}$ denotes a diagonal matrix built from $s^{(n)}$, the reliability index at iteration $n$. Additionally, $s^{(n)}$ is computed from a Hadamard Product $\odot$ (element-wise product), according to:

$$s^{(n)} = t^{(n)} \odot u \quad \text{or} \quad s^{(n)}_i = t^{(n)}_i \times u_i$$ (2.8)

where $s^{(n)}_i$ corresponds to index $i$ of vector $s^{(n)}$.

Finally, $\alpha^{(n)}$, the Lagrange Parameter at iteration $n$, is obtained by computing one of the following methods:

**Method 1.**

$$\alpha^{(n)} = -(G t^{(n)} + k) \odot (G \hat{s}^{(n)} G^T)$$ (2.9)

**Method 2.**

$$(G \hat{s}^{(n)} G^T) \alpha^{(n)} = -(G t^{(n)} + k)$$ (2.10)

**Method 3.**

$$(G \hat{s}^{2(n)} G^T) \alpha^{(n)} = -(G t^{(n)} + k)$$ (2.11)
where $\odot$ denotes element-wise division.

The result from this algorithm is a balanced system, i.e., $t$ is such that Equation 2.5 is verified.

A simplified version of the complete algorithm runs as follows:

Algorithm 1 Simplified balance algorithm

1: function BALANCE_SYSTEM($G$, $t$, $u$, $h$, method, lim, $dm$)
2: quality ← 0
3: for quality < MAX(h) do
4: $G' ← $ remove_unnecessary_columns($G$, quality)
5: $k ← G' \times t$
6: error ← lim
7: while error $\geq$ lim do
8: $s ← t \times u$
9: error ← MAX(ABS($G \times t + k$))
10: if method $= 1$ then
11: $\alpha ←$ compute_alpha_method1($G$, $s$)
12: else if method $= 2$ then
13: $\alpha ←$ compute_alpha_method2($G$, $s$)
14: else if method $= 3$ then
15: $\alpha ←$ compute_alpha_method3($G$, $s$)
16: $m ← 2 \times dm$
17: while $m > dm$ do
18: $t' ← t + \hat{s} \times G^T \times \alpha$
19: $m ←$ MAX(ABS($1 - t' \odot t$))
20: $\alpha = \alpha / 2$
21: error' ← MAX(ABS($G \times t' + k$))
22: $t ← t'$
23: if MAX(error') $\geq$ MAX(error) then
24: Break

It is important to notice that three loops are present in this algorithm: the first starts at Line 3. We refer to this loop as the hierarchy loop, since it loops trough data quality ranks from the lowest to the highest; the second, resent at Line 7 is the main loop, called IWLS loop; and finally the third, at Line 17 is the must inner loop and has the purpose of computing the next solution approximation.

2.3 Computational Challenges

While the mathematical concepts are not complex, performing such algebraic operations on high dimension objects still present considerable computational challenges. Indeed, due to such extensive objects, matrix and vector storage must be considered, since all operations will benefit from an optimized object manipulation. This is a well known optimization since in 1979 Duchin and Szyld [3] introduced several techniques for large sparse models.

Analyzing Equation 2.7, responsible for computing the next solution approximation, presented in Line 18 of Pseudo-Code 1, it is easily understood that a matrix-vector product is computed. Furthermore, the computational error, which describes whenever the algorithm should end, computed from $\epsilon = Gt_s + k$ at Line 9 also requires the computation of a matrix-vector product. Since these operations are executed once IWLS iteration, the algorithm requires the computation of at least two matrix-vector products per IWLS iteration.

To further illustrate this analysis, consider an estimation and harmonization of a Quarterly Census of Employment and Wage (QCEW) database example, discussed in the results chapter. This particular real-life case presents an aggregation matrix with 1 million rows per 2 million columns. Our experiments
showed that, for a database of such dimensions and initial guesses, applying this algorithm requires around 3500 \( \text{IWLS} \) iterations to find a solution. Further, consider a maximum of 10 executions per iteration for Equation 2.7. Hence, the expected number of matrix-vector products in a single execution, \( p \), determined as \( p = 3500 \times (10 + 1) \) is \( p = 38500 \). This indicates a possible optimization challenge as, optimizing matrix-vector products will lead to a better overall performance.

Additionally, depending on the user’s choice, the algorithm will compute the Lagrange Parameter \( \alpha \), following Equations 2.9, 2.10 or 2.11. This difference between methods is visible at Lines 10, 12 and 14 respectively. The first performs an Hadamard division, which is a simple point-wise division of vectors. However, the second and third options will solve a system of linear equations, in which \( \alpha \) is the solution.

Methods for solving linear systems of equations have been developed with extensive research in the past, since this is a well known operation in IOA and other areas, such as Life Cycle Assessment [7]. Considering that solving systems of such dimension demand considerable computational resources, and that \( \alpha \) must be computed every \( \text{IWLS} \) iteration, a conclusion is reached: \( \alpha \) computation is one of the most computationally heavy operation in this algorithm and its optimization will have a large impact on the overall computation. Moreover, the algorithm parallelization and work distribution must be considered, since it is intended to run this algorithm in a cluster of machines.

To conclude, to successfully reduce computational time, we intend to focus on the parallelization of matrix-vector products and linear system solving. Considering the arguments above, the parallelization of such operations will heavily impact the overall algorithm performance and successfully reduce computational times.

2.4 Summary

In this chapter we discussed the foundations of IOA. Professor Wassily Leontief developed this framework to analyze connections between different economic sectors. The first analysis was performed in the 1930s, using a 42-sector table. Due to its dimensions, computations were carried by a machine. IOA can also be extended to other areas, addressing problems of different areas and natures.

Further, we propose the data balancing solution used in this work. The algorithm of Rodrigues evolves in an iterative way, computing better successive solutions each cycle. Additionally, this algorithm has three different methods of computing solution values, each choice requiring different computational resources.

Finally, the chapter ends discussing the computational challenges of this algorithm. Matrix-vector products and linear system solver are the computationally operations, requiring special attention.
Chapter 3

Related Work

This chapter discusses the techniques used to solve the Economical Data Balancing problem specified in Chapter 2. The intention is to discuss state-of-the-art solutions to deal with the computational challenge presented in the last chapter. Theoretical solutions are explored by providing a comparative view, discussing advantages and disadvantages, for their different appliances. Since both fields of IO Analysis and Scientific Computation are closely related, much work has been done in the past, aiming for faster solutions using highly optimized methods.

The analysis in this chapter is made from low to high level of perception, starting by discussing matrix representations and ending by addressing possible software packages to support implementation requirements.

3.1 Data Storage

To compute a solution for this problem, it is required to perform operations on massive matrices and vectors. Both, matrices and vectors, should be represented using the most efficient format, for a specific problem.

Generally, two options of storing data are available:

- Dense Storage;
- Sparse Storage.

Using dense storage, all matrix entries are kept on the machine, resulting in higher memory requirements. Dense storage can only be used when dealing which small matrices or vectors, as algorithms are generally less complex.

However, this is not the correct way of dealing with large amount of information, since some parts can be discarded. For instance, consider a matrix composed by a majority of zero values. To minimize memory requirements, it is possible to drop null values, storing only the non-zero entries.

Sparse storage gains in performance, since this type of storage only keeps the non-zero elements of a structure, less data is processed and kept. Additionally, to better exploit matrices properties, these techniques store all non-zero values in contiguous memory positions, optimizing the cache usage, due to the Principle of Locality.

3.2 Storage Formats for Sparse Matrices

There are different methods for storing sparse structures, however the most commonly used ones are:
• **Coordinate Format (COO)**

• **Compressed Sparse Rows (CSR)**

• **Compressed Sparse Columns (CSC)**

There is no best format for all causes. As such, it is important to evaluate which one is the better for our case.

### 3.2.1 Coordinate Format (COO)

Sometimes referred as Triple Format, this method applies one of the most basic concepts for storing the non-zero values of a matrix [8]. COO stores each non-zero value in a data structure, and for each one of them, the respective row and column index are stored in two auxiliary data structures. The list index of each non-zero element correspond to the index of the two auxiliary structures - the row and column index list. In some software libraries, the COO is the default method for matrix creation for its flexibility, on adding or removing elements, and application to any matrix. Usually, after creation, a matrix in COO format can be converted into any other format, for convenience [8].

### 3.2.2 Compressed Sparse Rows/ Columns (CSR/CSC)

CSR and CSC [8] are formats that can be used to represent any sparse matrix using less memory, compared to the Coordinate Format. In CSR, the value of each non-zero element is stored in a contiguous memory vector. Matrix elements are ordered by row, meaning that elements on row $r$ are placed before $r + 1$, and follow the order they appear in that particular row. Additionally, CSR uses two more vectors: the first one stores the column index for each non-zero element stored in the first vector; the second vector stores a pointer to each row.

For CSC, the representation follows CSR very closely. The difference is that non-zero elements are stored by column. As such, the second vector stores the column index for each non-zero element and the last vector stores a pointer to each column.

The storage requirements are two vectors of size $nnz$ - number of non-zero matrix entries - and a vector of size $k + 1$, $k$ being the number of rows, in CSR or columns, in CSC [8].

CSR is a very popular storage method since it is specially constructed for matrix-vector products [9].

As an example, consider matrix $A$ with 4 rows and 6 columns:

$$A = \begin{bmatrix} 21 & 33 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 6 & 0 & 0 \\ 0 & 0 & 50 & 71 & 44 & 0 \\ 0 & 0 & 0 & 0 & 0 & 66 \end{bmatrix}$$

The structure to represent $A$ in COO format would contain the following data:

$$nnz = 8$$
Values = \[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
21 & 33 & 12 & 6 & 50 & 71 & 44 & 66
\end{array}
\]

ColumnIndexes = \[
\begin{array}{cccc}
0 & 1 & 1 & 3 & 2 & 3 & 4 & 5 \\
\end{array}
\]

RowIndexes = \[
\begin{array}{cccc}
0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 \\
\end{array}
\]

For **CSR** representation, we replace **RowIndexes** vector:

Values = \[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
21 & 33 & 12 & 6 & 50 & 71 & 44 & 66
\end{array}
\]

ColumnIndexes = \[
\begin{array}{cccc}
0 & 1 & 1 & 3 & 2 & 3 & 4 & 5 \\
\end{array}
\]

RowPointers = \[
\begin{array}{cccc}
0 & 2 & 4 & 7 & 8 \\
\end{array}
\]

As for **CSC**, we replace **ColumnIndexes** vector for **ColumnPointers**, and change the order of the values as:

Values = \[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
21 & 33 & 12 & 6 & 50 & 71 & 44 & 66
\end{array}
\]

RowIndexes = \[
\begin{array}{cccc}
0 & 0 & 1 & 2 & 1 & 2 & 2 & 3 \\
\end{array}
\]

ColumnPointers = \[
\begin{array}{cccc}
0 & 1 & 3 & 4 & 6 & 7 & 8 \\
\end{array}
\]

### 3.3 Solving a Linear System

The problem addressed in this thesis is to balance a set of economical data. As explained in Section 2.2, the most computationally heavy step in the algorithm is to find the Lagrange parameter \( \alpha \), as it requires the solution of a linear system.

Similarly to Equations 2.10 and 2.11, solving a linear problem is the act of finding a solution for the following equation:

\[
Ax = b
\]  \( (3.1) \)

where \( A \) is a matrix representing a set of linear equations, \( b \) is a constant vector for the right hand side and \( x \) is a vector of unknowns.

Equations 2.10 and 2.11 and 3.1 have many similarities - (1) \( A \) takes the value of \((G\hat{s}(n)G^T)\); (2) \( b \) takes the value of \(- (Gt(n) + k)\); finally \( x \), the vector of unknowns, is analogous to vector \( \alpha \), of unknown Lagrange parameters.

Since this is the step that consumes most computational resources, its optimization will benefit the overall method.

Large amount of work has been done to optimize linear solvers, with the purpose of making these algorithms more efficient in time and memory consumption. Widely used methods were modified to better exploit multi-computer architectures, in order to make algebraic operations more efficient and reduce the computing time. There are two types of methods for solving linear systems, and each one is
suitable for different kind of data. These are classified as direct and indirect (or iterative) methods.

Direct methods, such as Gaussian Elimination and LU-Factorization [10], are suitable for small to medium systems. These algorithms are an exact solution for a given problem, but tend to use unnecessary memory and consume more time. Moreover, these methods tend to reduce the matrix sparsity, resulting in a gradual performance deterioration.

The best approach for our problem is to use indirect methods for solving systems with such dimensions. In contrast to direct methods, these algorithms are better suited for large sparse systems [10]. Methods such as Conjugate Gradient (CG) and Generalized Minimal RESidual (GMRES) do not provide an exact solution, they only afford an approximation to the correct value. They start with a guess and iterate to a more accurate solution. The result of this approach is a reduction in the number of multiplications, and thus, a reduction in the computational time [8].

Since direct methods are not suited for our problem, in the following sections we focus on existing iterative methods and their properties.

### 3.3.1 Rate of Convergence

The most desirable feature of a linear problem solver is the time it takes to find a solution. Multi-computer architectures aim to provide the power to accelerate these processes.

Along with this, it is important to note that not a single method is suitable for all cases. In some situations, applying the best algorithm will result in a faster solution, providing a faster response [10, 11, 12]. This response speed is called, in numerical analysis, Rate of Convergence [13]. Convergence can be calculated by the difference between successive approximations, that is

$$\epsilon = x_i^{(k+1)} - x_i^{(k)}.$$  

When $\epsilon$ is smaller than the specified residual value, the computation stops [13].

### 3.3.2 Jacobi Method

The Jacobi method belongs to the family of the stationary iterative methods [12]. This particular kind of methods are expressed in the form

$$x^k = F x^{k-1} + c,$$

where $x^k$ is the approximated solution in iteration $k$, and $F$ and $c$ are independent of $k$ [12]. It is a simple way of solving large linear systems when each row and column have its large absolute value on the diagonal element [10]. As all iterative methods, Jacobi starts with an initial guess and iterates until the residual difference is less than a given threshold [10]. In each iteration, the following calculation is performed:

$$x_i^{(k+1)} = a_{ii}^{-1} (b_i - \sum_{j \neq i}^n a_{ij} x_j^{(k-1)})$$  \hspace{1cm} (3.2)

Note that the last calculated value is used, approximating the current solution to the exact one. As stated, an important aspect of iterative algorithms is their solution convergence. The Jacobi Method is sure to converge when the matrix $A$ is diagonally dominant, meaning that, the value on the diagonal of each row of $A$ should be greater than the sum of all remaining values, in that particular row.

A possible optimization to accelerate the converge of Jacobi by adding an over-relaxation parameter, resulting in a method also known as Jacobi Overrelaxation (JOR) [10]. Since Jacobi and JOR only use the old solution for the new approximation calculation, these methods possess a high degree for natural parallelism. However, both of them still show slow convergence [12] [10].
3.3.3 Gauss-Seidel

In general, Gauss-Seidel (GS) converges twice as fast as the Jacobi Method. This is due to the fact that GS uses the most recently available solution approximation. Doing so, some parallelism potential is lost, in contrast to Jacobi Method. Each iteration, the GS method computes the following calculation:

\[ x^{(k+1)}_i = a_{ii}^{-1} (b_i - \sum_{j=i+1}^{i-1} a_{ij} x^{(k+1)}_j - \sum_{j=i+1}^{n} a_{ij} x^{(k)}_j) \]  

(3.3)

As expected, from indirect methods, GS starts too with an initial guess and performs successive approximations to the final value. This process stops when "the relative approximate error is less than pre-specified tolerance" [10].

Gauss-Seidel is sure to converge when matrix \( A \) is diagonal dominant. It is very efficient concerning memory usage, time requirements and error adjusting, for matrices with high sparsity coefficient [10].

An extension to GS method is the Successive Over-Relaxation (SOR) [12]. Analogous to the JOR in the Jacobi Method, the SOR attempts to surpass GS in the speed of convergence. To do so, a relaxation factor \( \omega \in (0, 2) \) is inserted. For the optimal value of \( \omega \) the computation can out-perform GS. However, the pre-calculation of \( \omega \) is not affordable [12].

3.3.4 GMRES

GMRES was developed by Saad and Schultz as a generalized Minimal RESidual (MINRES) method, for solving slightly large number of linear equations [11]. It is better suited for non-symmetric systems and in all types of systems in general. It is known for giving good results in most cases, making it a good choice for large sparse systems [11].

Starting with an initial guess \( x_0 \), the algorithm calculates a new solution approximation, \( x_n \) [14][10]. The goal idea is to minimize "the residual norm of the approximate solution over the Krylov subspace at each iteration until reaching the desired residual tolerance \( \epsilon \)" [11].

A variance of the GMRES is the GMRES with restarts. In the same logic, this method will restrict the Arnoldi process at \( m \ll n \) iterations and restart with the most recent solution as an initial guess for the new approximation calculation.

There may be some cases in which GMRES does not converge or takes too many iterations to provide a solution. To improve the performance in these cases, a preconditioning matrix must be added. This technique modifies the original Equation 3.1 to:

\[ M^{-1} A x = M^{-1} b \]  

(3.4)

\[ A M^{-1} \hat{x} = b, \; x = M^{-1} \hat{x} \]  

(3.5)

where [3.4] and [3.5] represent Left and Right preconditioning, respectively, and \( M \) is the preconditioning matrix.

3.3.5 Conjugate Gradient

Conjugate Gradient is possibly the best-known iterative method for solving large linear systems [10, 9, 13]. Introduced by Hestenes and Stiefe, this technique is applicable when, in Equation 3.1, \( A \) is a positive definite matrix. This algorithm is still the method of choice for systems with such properties [14].
In each iteration, the approximation

\[ x_{k+1} = x_k + \alpha_k p_k \]  

is computed, calculating one matrix-vector product and two inner products. In Expression 3.6, \( p_k \) are known as direction vectors and \( \alpha_k \) are used to minimize the function in the respective direction \( p_k \).

The convergence rate of \( \text{CG} \) is usually good in most cases, but this is not true when matrix \( A \) has undesirable properties, i.e., large condition number. In these cases, a precondition matrix could be used to accelerate the solution convergence [13].

### 3.4 Software Packages

To implement a numeric application, the building blocks are necessary. Operations, such as matrix multiplication and vector dot products, are essential to any scientific application.

Our initial approach was to develop a new library from zero. The language of choice was C because of its flexibility and good performance. This library only supported dense operations, and was constructed with the purpose of testing some of the algorithms explained. Since the dense format is not the goal of our work, a sparse version of this library was constructed. However, because it was not optimized, existing libraries were considered.

The intention was to find a free numerical library that could fit the following requirements:

1. The library must support sparse structures;
2. It must provide a parallel implementation of basic matrix operations;
3. The implementation would be able to run in a cluster of machines using MPI;
4. Support C or C++ programming languages;

From search and advises, five packages were identified and discussed. However, some were excluded because they could not meet a given requirement. The packages were the following:

**BLAS** Basic Linear Algebra Subroutines (BLAS) is probably one of the best known package for low-level algebraic subroutines [15]. This library implements the basic operations for matrices and vectors, providing the basic blocks for any scientific application. It has many implementations in various programming languages, such as C, Fortan and OpenCL. Despite its popularity, BLAS does not support sparse storage nor sparse operations.

An extension to BLAS is Sparse Basic Linear Algebra Subprograms (spBLAS) [16]. As the name suggests, spBLAS provides the same functionality as BLAS but in the sparse domain. However, it does not implement parallel operations.

Yet another good extension is Parallel Sparse Basic Linear Algebra Subroutines (PSBLAS) [17]. This library implements a parallel version of spBLAS better suited for our problem.

**GSL** GNU Scientific Library (GSL) is a powerful numerical library for C and C++ developers [18]. This library provides a set of mathematical operations such as sorting and linear algebra. GSL uses BLAS for its implementation of low-level operations.

Despite being very efficient, this library does not provide parallel functions. Since this work is meant to reduce the computational time using multiprocessor architectures, this library is not suitable.
Trilinos Another C library is Trilinos \[19\]. The Trilinos project is a large collection of packages that provide functionality for various large-scale engineering and scientific problems. One of the key points of this library is that it is scalable and suitable for multi-processor environments. Another good advantage is that one can select the most desirable package to use.

During our experiments, we did not find the basic operations needed for this project.

LAMA Yet another good package is Library of Accelerated Math Applications (LAMA) \[20\]. This library provides sparse matrix storage, such as the ones presented in this paper. Adding to this, LAMA also provides shared and distributed memory support, through OpenMP and MPI, respectively. Even more, this package provides parallel solvers for linear systems. LAMA is a C++ library and fully exploits the language features, providing operator redefinition. LAMA also provides support for CUDA kernels, for better exploit parallel features of the algorithms.

Despite its capability, during our experiments we did not manage to compile the library.

PETSc PETSc is a package of data structures and operations for parallel, large-scale applications \[1\]. It provides all the needed operations for this work, such as linear, nonlinear and matrix algebra, highly optimized and implemented for parallel matrices and parallel vectors. Moreover, PETSc also supports CUDA, OpenCL and hybrid MPI-GPU parallelism.

The suite is configurable, so it fits different requirements. This is accomplished by using different configuration parameters during compilation and installation.

Table 3.1 shows a comparative view between the different libraries. To conclude, PETSc would fit best for our problem. Besides meeting all requirements, and providing the necessary operations to support our implementation, PETSc is also well documented, easy to install, use and port between machines \[1\]. In summary, it is complete, efficient and easy to use.

3.5 Summary

In this chapter we detailed the computational challenges that come from solving the problem formulated in Chapter 2, more precisely, matrix storing formats and linear solving algorithms. For each issue, a set of solutions were considered, performing a comparative view between them.

Two linear solvers seem better suited for our problem: GMRES and CG works best when the input system is a positive definite matrix, whereas GMRES is better suitable for non-symmetric, non-Hermitian systems.

Lastly, the chapter ended by providing a list of possible computational software packages, and their characteristics. After evaluation and experimentation, PETSc would be the best solution to support our implementation.
Chapter 4

Materials and Methods

Following the Chapter 3, PETSc is the best candidate for our implementation, and is the one used in this work. As such, this chapter discusses the materials and methods used in our implementation. It starts by discussing how basic algebraic matrix operations are provided, and how each is used.

Moreover, this chapter has the purpose of describing the ideal usage of basic operations, as a base for the algorithm implementation, starting by focusing on functions of higher importance, and gradually generalize to the complete process of execution.

Much attention is given to which methods are used in each precise step, giving a detailed view of application construction. Furthermore, it is also exposed how parallelization of functions is accomplished, providing a detailed view on matrix and vector structures, explaining how operations are performed.

The chapter ends discussing the input and output files of our program. Data files are provided with concrete problems, while configuration files are used to set thresholds and constants.

4.1 Operations Required

Finding a solution to Equation 2.7, requires the computation of Lagrange Parameter $\alpha$. This computation can be accomplish in two different methods: (1) performing a Hadamard division; or (2) solving a linear system of equations.

Using the first, computation time will be reduced, as computing a Hadamard division. More precisely, instead of using a full matrix to compute $\alpha$, only its diagonal is used, and thus, is faster than solving a system of linear equations. Moreover, using this method, the most computational heavy operation is the calculation of matrix-vector products, occurring often during execution.

On the other hand, using the second method, both operations - linear system solving and matrix-vector products - will dominate execution time. Naturally, optimizing these operations will lead to better execution times in both methods.

Starting with the linear system solution, the process consists in finding a solution for Equation 3.1.

As discussed, PETSc implements all necessary functions for this project. To use a linear solver, the user creates and configures these objects with the desired objectives. This can be accomplished at runtime, by providing configuration flags when starting program execution, or at compile time, by defining in the application code.

The function KSPSetOperators is used to specify the input matrix and left hand for a particular linear problem. Additionally, KSPSetType is used to define the solving method. By default, the program will use GMRES method, however the user can also choose the CG. Finally, the process starts by calling function KSPSolve.
When the process is complete, PETSc provides a solution and, optionally, the number of iterations and convergence ration for that particular execution. These optional values were used in this work to evaluate different solving methods. The developer must use KSPGetIterationNumber and KSPGetConvergedReason functions to find the number of iterations and convergence reason, respectively.

4.1.1 Computing Matrix A

In this work, the linear system of equations is not known from input data. Moreover, the system is not constant. It is up to the application to compute the matrix form of each linear problem, as addressed in Equation 2.10, following:

\[ A(n) = G\hat{s}(n)G^T \]  \hspace{1cm} (4.1)

where \( A(n) \) is the linear system matrix at iteration \( n \), \( G \) is an aggregation matrix from input data, \( G^T \) denotes the transpose of \( G \), and \( \hat{s} \) is a square diagonal matrix, constructed from a vector \( s \) at iteration \( n \). We discuss possible optimizations for this step, in the next chapter.

PETSc provides this operation for parallel sparse matrices. Starting by computing \( G^T \) MatTranspose is called to obtain the distributed transpose of matrix \( G \). Finally, a three matrix multiplication is performed by invoking MatMatMatMult.

Note that this operation is performed once for every IWLS iteration, and thus, should be optimized with respect to computational time.

4.1.2 Computing Left Hand and Error

The left hand vector \( b \) needed to solve the linear system in iteration \( n \) is computed following Equation 2.10, more precisely as:

\[ b(n) = Gt(n) + k \]  \hspace{1cm} (4.2)

where \( G \) is an aggregation matrix, \( t(n) \) and \( k \) are vectors.

Analogously the error \( \epsilon \), used to check for the exit condition, is also calculated using the same expression:

\[ \epsilon(n) = Gt(n) + k \]  \hspace{1cm} (4.3)

From Equations 4.2 and 4.3, it is easily understood that a matrix-vector product must be computed, in which both \( b \) and \( \epsilon \) are represented as vectors distributed across all machines.

To perform this operation, we use PETSc method MatMultAdd, which will multiply a matrix and a vector, adding the result to a second vector.

Additionally, the exit condition is calculated by finding the maximum value in error vector, and comparing the result with the provided threshold. This operation is performed by invoking VecMax PETSc function.

Finally, the rate these operations are executed must be considered. Since both, the left hand and error calculation, require this computation, it could be assumed that a matrix-vector product is computed at least twice per IWLS cycle. This is very important, regarding performance, and will be addressed in more detail ahead in this document.
4.1.3 Hierarchical Cycle

During the course of this paper, only operations within the IWLS algorithm were addressed. In this section, operations in hierarchical cycle are exposed.

In each hierarchical cycle iteration, an IWLS algorithm execution is performed. Additionally, important matrix and vector modifications are made. These modifications consist in removing entries and columns, from the original vectors and matrix, respectively. Depending on each specific iteration, these entries and columns change accordingly to the data quality rank vector \( h \), provided in data input. More specifically, the length of \( h \) is equal to the number of matrix columns, allowing each vector index to correspond to a matrix column. As such, all values of \( h \) greater than the current hierarchical level, should have its corresponding matrix column removed.

This operation is applied to the input matrix \( G \), best guesses \( t \) and relative uncertainty \( u \) vectors, resulting in smaller versions of these objects.

To illustrate this further, consider the following example matrix \( G \), hierarchy information vector \( h \) and best guesses vector \( t \):

\[
G = \begin{bmatrix}
    c0 & c1 & c2 & c3 & c4 & c5 \\
    1 & 1 & 0 & -1 & 0 & 0 \\
    1 & 0 & 1 & 0 & -1 & 0 \\
    0 & 1 & 1 & 0 & 0 & -1 \\
\end{bmatrix}
\]

\[
h = [1 \ 1 \ 1 \ 2 \ 2 \ 2]
\]

\[
t = [1 \ 2 \ 3 \ 4 \ 5 \ 6]
\]

Note that the maximum entry in \( h \) is 2, meaning that this problem requires two hierarchical iterations to be solved. Furthermore, to each value of \( t \) and column of \( G \), a value of the same index in \( h \) is assigned.

Starting at iteration \( n = 1 \), and zeroing all elements of \( h \) greater than \( n \), the result would be:

\[
h = [1 \ 1 \ 1 \ 0 \ 0 \ 0]
\]

Meaning that columns and indexes 3, 4 and 5 should be removed from \( G \) and \( t \), respectively, resulting in the following objects:

\[
G = \begin{bmatrix}
    c0 & c1 & c2 \\
    1 & 1 & 0 \\
    1 & 0 & 1 \\
    0 & 1 & 1 \\
\end{bmatrix}
\]

\[
t = [1 \ 2 \ 3]
\]

To implement this operation, PETSc provides functions to zero complete matrix columns, or vector entries. As such, it is possible for each processor to calculate which entries to remove, and call MatZeroRowsColumns for matrix columns, or VecSetValue for vector indexes. However, since these operations are only replacing each entry by zeros, some problems regarding matrix structure and load balancing are considered. These problems will be addressed in mode detail ahead.

Finally, at the end of every hierarchic iteration, result information must be gathered in a single processor, so a new iteration can start. In our particular case, the data consists in a single vector, since all...
remaining objects can be discarded. Using \texttt{PETSc VecScatterCreateToZero}, the application gathers all vector chunks, creating a single vector on the root machine.

This operation is executed at the end of every hierarchic cycle, thus consuming some additional time. However, since a problem usually requires two to three hierarchic iterations, the overhead is very small.

### 4.2 Object Distribution and Assembly

Until now, despite being a core decision in a parallel application, little to no attention was given to division and distribution of matrices and vectors. The way objects are divided across machines is crucial, regarding performance, as a given distribution could provide a better parallelization. For this matter, the decision is carefully discussed in the following sections.

Despite \texttt{PETSc} providing a good abstraction in data distribution, the user still needs to find the best way to divide data, as each specific implementation is addressing a different problem.

The important aspect of matrix division is one of load balance. Considering that all machines should have identical chunk sizes for optimal performance, objects should be distributed accordantly. This concept is also important when addressing parallel operations, such as matrix-vector products. A good distribution must guarantee that, if two objects are meant to interact, its local sizes must be equal for the same processor.

#### 4.2.1 Matrix construction

In this project we use the \texttt{MPIAIJ} matrix representation of \texttt{PETSc}. Essentially, the matrix is divided by chunks of contiguous rows, scattered across machines. Moreover, \texttt{MPIAIJ} distributed matrices are represented in two sub-matrices: a diagonal values matrix; and a non-diagonal values matrix.

**Diagonal Values Matrix** This matrix will store all entries present on the diagonal part of a matrix. The size of this block varies with the number of processors, since the number of rows per chunk also vary.

**Non-Diagonal Values Matrix** Stores all the remaining values that were not stored on the diagonal values sub-matrix. This block is usually larger than the one with the diagonal values.

This approach provides a better parallelization of operations, such as matrix-vector products, since each processor contains a set of complete matrix rows.

To distribute a matrix, the developer first creates an empty matrix object across all processors, by calling \texttt{MatCreate}. After, it is necessary to define the matrix dimensions and local sizes, using \texttt{MatSetSizes}. In this step is possible to define how the matrix is divided across machines. \texttt{MatSetType} is then called to set the matrix type. For this project distribute sparse matrix were used, however \texttt{PETSc} supports other matrix formats, such as sparse local matrix, sparse block matrix or dense matrices.

In a cluster of four processors, scattering a \texttt{MPIAIJ} matrix, would have a similar aspect to matrix presented in Figure \ref{fig:matrix_distribution}.

The chunk of rows in each processor is presented by letters \texttt{A}, \texttt{B}, \texttt{C} and \texttt{D}. Additionally, each sub-matrix is signed by \texttt{D} and \texttt{ND}, for diagonal and non-diagonal sub-matrix, respectively. Furthermore, dividing the same matrix across a different number of machines would result in different sub-matrices sizes. This effect can be noticed from matrix presented in Figure \ref{fig:matrix_distribution2}.

Comparing both distributions, it is possible to conclude that, as the number of machines grow, values in non-diagonal sub-matrices will also increase and values in diagonal sub-matrix will decrease. This key idea is particularly interesting when discussing application scalability, ahead in this document.
4.2.2 Vector Construction

To better understand the purpose of this matrix representation, it is important to understand division of vectors.

PETSc vectors are created using the same process as matrices: first, the object is created across processors, invoking `VecCreate`; after and the values are distributed.

Analogous to the matrix, all vectors are divided across processors as chunks of contiguous values. This division is set by the programmer, calling `VecSetSizes`, providing the local size for each processor chunk, and the global size. This strategy allows for perfect parallelization of operations, such as vector sums, as each machine only processes its local vector entries.

These creation functions should be called by each processor, guaranteeing all machines create the distributed object.

A distributed dense vector would be scattered according to vector in Figure 4.3 using four processors, A and B.
Finally, distributing the same objects across two processors would result in a vector similar to one in Figure 4.4.

4.2.3 Object Data Distribution and Preallocation

After creating empty structures, the next step is to distribute the data itself. PETSc provides functions to distribute matrix and vector data, sending each value to the correct machine. More precisely, it is possible for a single machine to distribute all values across the remaining machines.

The procedure is identical for both, matrix and vector. Invoking VecSetValues and MatSetValues will cache values for respectively, vector and matrix. Finally, all processors must invoke vector and matrix assembly routines, so the values are sent to the respective machine.

To better assembly a matrix, for memory allocation purposes, PETSc requires the number of non-zero entries per matrix row. This is expected since it is faster to allocate the complete, big structure, than many small ones. If the user does not provide the needed meta-data, each matrix entry will trigger memory allocation during assembly, resulting in longer matrix assembly times.

For sparse local matrices, this requirement is easily met, creating two data structures holding counters for each matrix row. The structure is then indexed using the column number while loading input file and incremented, reducing posterior overhead of iterating through all nonzero entries.

However in this work, distributed sparse matrices are the most used. For these matrices, it is necessary to provide, for each processor row, the number of non zero entries in the diagonal block, and outside the diagonal block. This is expected, as PETSc stores distributed parallel matrices in sub-matrices, as explained in Section 4.2.1.

The preallocation information should be set by invoking MatMPIAIJSetPreallocation. Additionally, no meta-data is necessary for vector preallocation, as in our problem these are stored in dense format. These steps are performed each time a new matrix object is created.

4.3 Loading Input files

The program accepts two input files, one for configuration and another for the data itself. This option gives the user more flexibility as a single problem methodology can be solved with many different configuration parameters.

4.3.1 Configuration File

The configuration file is human readable, and has the purpose of providing the configuration values, such as thresholds and input file name. The user can customize the following parameters:
Data input file name Indicates the data input file that describes the problem methodology. This parameter takes a directory and name of the file, written as a string of characters.

Data output file name Dictates the output file. This file contains the solution vector for the problem on the data input file. It should contain a string of characters with the directory and name of this file.

Extra output file name The extra output file is a human readable file containing information regarding input loading time, computational time, output write time and algorithm errors. This parameter should be set with a string containing the directory and name of the extra information file.

Object names It is possible to store multiple vectors and matrices in a single input file. As such, the user should provide the name of each object to load as aggregation matrix, best guesses, hierarchy information and relative uncertainty. Each name should be provided as a string of characters.

Matrix type It is possible to set if the input matrix is given in sparse form - triplet list - or in dense form. Since only sparse matrices were used, this option was not changed during this project. This parameter can be set with the string SPARSE or DENSE.

Threshold values Also very important are the values for the different thresholds, such as maximum expected error, and maximum number of iterations. It is relevant to set and record these values since those are responsible for controlling the exit conditions for the algorithm.

Solving method This parameter will set the method used to calculate alpha. It should be set with the integer value of 1, 2 or 3, as described in Equation 2.9, 2.10 and 2.11, respectively.

Linear algorithm Finally, one can also choose the linear method to use. Note that this option will be ignored if the solving method zero is in use the It is possible to select either GMRES or CG, according to the convenience.

4.3.2 Data File

The program loads octave files since all problem topologies were saved from Octave programs. GNU Octave is a high-level programming language, developed for numerical computations. Among other features, Octave provides methods for solving linear and non-linear system of equations, as well as matrix and vector basic operations [21].

It starts by looking for the problem sizes, i.e. the matrix dimensions and number of non-zero entries. After, the program loads each object information into memory, as each vector data is loaded into an array of values. Regarding matrix information, three arrays of data are used: one for column indexes, one for row indexes and a last for the values. These arrays store each matrix entry as a triplet.

With the object data in simple structures, it is possible to perform modifications and permutations to the input data. The objective of pre-processing data is to make the matrix structure more appealing for parallelization and optimization. The concrete modifications and their purpose will be explained ahead in this chapter.

As explained in Section 4.2.3 meta-data is needed with the purpose of matrix allocation. The meta-data consists in two arrays of positive integers, which sizes are equal to the number of matrix rows, and matrix columns. Hence, while loading the input file, the application creates these vectors.

4.4 Output Files Creation

Finally, after all computations, the program writes the output files. It is possible to define two output files: (1) the solution file; and (2) the extra information file.
The procedure to write the solution file is straightforward. Analogous to Section 4.1.3, the solution is distributed across all machines and should be gathered in a single processor in order to be saved in a file.

Despite PETSc supporting binary files, the output is written in octave compatible format, so other applications can easily load its contents.

The second output files - extra information file - contains data regarding execution times, load times and solution errors. Three different times are recorded: file loading and object distribution time, computational time except Lagrange parameter calculation and Lagrange parameter calculation time. It is a human readable file, mainly used when logging performance test results.

4.5 Summary

In this chapter described how PETSc was used in this project's implementation, and how the application flow was organized. The resulting architecture, consists in reading the input file, distribute data across the machines, performing the algorithm and collecting the result on a single machine.

Additionally, it also provides information regarding chosen PETSc functionalities, and the reason for these choices.
Chapter 5

Implementation

The purpose of this chapter is to address the most relevant issues found during implementation process, and their possible solutions. It focuses on performance scalability, and their influential factors, for increasing numbers of processors. Scalability was the problem of greater relevance when developing this project. During testing phases, application performance did not scale as number of processors rise. Analyzing all possible causes, hypothesis were considered. Moreover, this Chapter also explains how PETSc object representation, such as sparse matrices, can directly lead to these issues.

Finally, the chapter ends by offering possible optimizations for operations. A special method is proposed, to avoid matrix multiplication when using algorithm method 1, and a permutation for matrices used in linear solvers.

5.1 Cluster Configuration

The first hypothesis for the low scalability was that PETSc was improperly configured, leading to bad performance.

PETSc provides benchmarks for cluster testing. When executed, these programs run for a varying number of machines, estimating the possible speedup each execution.

The cluster is composed by eleven machines, connected using ethernet. As expected, PETSc predicted a linear speedup, until the number of machines reaches eleven. After, the speedup decreases or, in some cases, maintains. In one way, this dismissed cluster configuration hypothesis, and thus, this was not the cause for bad scalability. Additionally, this also estimates the maximum number of machines and maximum speedup achievable using this cluster.

5.2 Load Balancing

Load balancing was the next hypothesis considered. Using the number of entries per row, it was trivial to compute the number of entries each processor had, allowing to see how matrix entries were divided across processors.

Since matrices are divided as chunks of rows, an unbalanced number of entries may lead to some processors need to process more information than others. For a better performance, all machines should have an identical amount of work.

Moreover, as explained in Section 4.1.3, it is necessary to remove columns and entries from matrices and vectors. This operation can also unbalanced work load, as some machines may have more columns than others, and thus, more matrix entries. That said, this problem can be divided in two sub problems.
5.2.1 Input Data Processing

One of the solutions is to process the input data. Since all data is loaded from disk into memory structures, it becomes more flexible and easy to change.

The approach is to use existent meta-data - number of entries per row - to reorder rows, in a way that all machines have the same amount of entries, in the same amount of rows. To accomplish this, a simple distribution algorithm it is used to assign rows to certain processors, depending on the total entries it processes.

To illustrate this further, consider an analogy of three bags, used to carry objects of different weights. Assigning each object to a bag, the goal is to have as much weight balance as possible across all bags. A simple approach would consist in assigning objects to each bag, considering object and bag weights. More specifically, placing the heaviest available object in the lightest available bag, prioritizing these with fewer objects. Repeating this process iteratively until no object is left, would result in a similar weight and number of objects across all bags.

In this particular case, rows are assigned to each processor, as objects to each bags. Additionally, denser rows are assigned to processors with fewer matrix entries and fewer rows.

In the end, this algorithm will balance work load across all processor as expected.

5.2.2 Object Distribution

As explained in Section 4.1.3, the processors work may become unbalanced after removing entries from vectors. To better understand this issue, consider two vectors, \( t \) and \( h \), divided across two machines, \( A \) and \( B \). A correct division would be:

\[
\begin{align*}
  t &= i_0 \ i_1 \ i_2 \ i_3 \ i_4 \ i_5 \\
    &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \\
  h &= \begin{bmatrix} 1 & 1 & 1 & 2 & 2 \end{bmatrix}
\end{align*}
\]

where the left chunk of each vector belongs to machine \( A \), and the remaining chunk to machine \( B \).

Applying the same logic as in Section 4.1.3 the result would be:

\[
\begin{align*}
  t &= i_0 \ i_1 \ i_2 \\
    &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\
  h &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}
\end{align*}
\]

Notice that, machine \( A \) has three entries whereas machine \( B \) has none, since these indexes were be removed. Note that, after created and distributed, an object cannot change, and PETSc does not provide any automated way of preventing this unbalance.

The solution consists in removing indexes from the original data, distributing information without these indexes. At the end of each hierarchical iteration, the result is gathered and reintegrated in a single machine. In posterior iterations, this data is processed, to remove undesired indexes, and redistributed across machines.

This approach will solve the present problem, at the cost of additional communication at the beginning and end, of each hierarchical iteration.
This hypothesis was dismissed as the cause for low scalability, since the desired performance was not achieved after assuring proper load balancing between machines.

5.3 Matrix Structure Analysis

Lastly, matrix structure and object distribution are considered.

Firstly, it is important to consider that this thesis refers to two different matrices: \( G \) is a horizontal rectangular, matrix loaded from the input file. This matrix is used in matrix-vector products, and matrix-matrix multiplications; and \( A \), a square matrix, computed using Equation 4.1. This matrix specifies the equations of a linear system, which is later solved, using GMRES or CG iterative methods.

Matrices and vectors are naturally distributed across all machines, meaning that, operations such as matrix-vector products, which requires the use of complete vector in all processors, leads to machine communication.

The strategy to reduce communication overhead consists in performing local computations while communicating. To further explain this point, one must first understand how matrices and vectors are stored in PETSc. As explained in Section 4.2, matrices are stored in a two sub-matrix structure: one sub-matrix will store all values belonging to the matrix diagonal blocks, and another sub-matrix for the remaining entries. Furthermore, vectors are stored in dense format, distributed across machines as chunks of contiguous values.

5.3.1 Matrix-Vector Product Scalability

Distributed matrix-vector products are computed each IWLS iteration, using matrix \( G \). Once performing this operation, each processor starts by broadcasting its vector chunk. In other words, all nodes exchange its vector chunk, until the entire cluster has a complete copy of the same. However, while communicating, each processor starts computation using its local vector values, by multiplying its local vector chunk per its local sub-matrix holding diagonal values.

Analyzing this operation, it is possible to conclude that, matrices with denser diagonal blocks and sparser non-diagonal blocks, will require longer for each processor to compute its local part. As a consequence, local computational time will dominate the operation overall time, providing longer times for communication. In summary, the more diagonal the matrix, the less communication overhead.

This key idea will dictate the difference between scalable and non-scalable matrix-vector products. It also leads to one core conclusion: each problem should be analyzed so \( G \) is changed using the optimal permutation. Different problems have different matrices, and thus, each matrix structure requires a different permutation to reach an optimal non-zero pattern.

During the course of this project, it was possible to find a possible permutation for one type of matrix. Consider the matrix structure presented in Figure 5.1, in which each dot represents a non-zero matrix entry.

This matrix shows two sparse diagonals, in red, and a denser one, in blue. Distributing this matrix across two processors would result in the division presented in Figure 5.2. Both processors, \( A \) and \( B \), contain the same amount of rows and same diagonal block sizes, presented in yellow.

However, one can see that some values in non-diagonal matrix, presented in gray. Indeed, for this particular matrix, 52% of entries are on the diagonal block for processor \( A \), and 47% for processor \( B \).

To transform this matrix into a more diagonally heavy one, it would be necessary to permute rows. The permutation would simply interlace the bottom part rows, with the top part rows. In other words,
for a matrix with \( N \) rows, counting from the top, the permutation would sort rows similar to 0, \( N/2 \), 1, \( (N/2) + 1 \), 2, \( (N/2) + 2 \), and so on. The result is a diagonal denser structure, presented in Figure 5.3.

Note that, with this permutation, the matrix is composed of one dense diagonal, presented in red, and two sparse diagonals presented in blue.

This trick allows for denser diagonal blocks, containing 95% and 90% of entries for processors A and B, respectively, in this case. As a result, computational times decreased for this specific test case.

### 5.3.2 Linear Solver Optimization

Yet another concern is to optimize the linear system solver. As expected, iterative linear solvers, such as GMRES and CG, will convergence faster if input matrix shows a highly diagonal structure.
Two possible solutions were used to diagonalize $A$.

- permute $A$ using an algorithm such as Reverse Cuthill-McKee (RCM).
- permute $G$ rows so $A$ is computed as permuted.

PETSc provides implementations for RCM and other permutation algorithms, that could be applied to permute $A$ when created.

However, applying these permutations on distributed matrices leads to a local permutation on each processor. In other words, to avoid large communication overhead, PETSc will not permute lines between machines. The result is a permuted matrix which structure varies with the number of machines, losing the algorithm effect for large clusters. Another problem with this solution is that the matrix will be permuted each time it is created. A in a nutshell, a permutation is performed each IRLS iteration, requiring additional time.

The goal is to obtain $A$, distributed across all machines, and permuted as if in a single machine. To accomplish this, one should create $A$ locally to a single processor, compute the permutation matrices and apply those to $G$. The result is a distributed $A$, computed as if permuted in a single machine. Using this technique it is also avoided to permute $A$ once created, optimizing the overall process.

### 5.3.3 Overall Scalability

Since $G$ is permuted in a way that $A$ is diagonal, $G$ becomes less desirable regarding non-zero structure, leading to a bad scalability in matrix-vector products. In other words, optimizing linear solver operation deteriorates matrix-vector product performance. Optimizing structure of $G$ for matrix-vector products does not assure diagonal structure of $A$.

In order to have a diagonal $A$ and a good $G$, it is necessary to use two $G$ matrices: one permuted into a more diagonal heavy matrix, for matrix-vector products; and a second to $A$ computation.

The only shared point between these two matrices is a single best guesses vector. However, this object would not require any extra communication, in a sense that it could be used by both matrices without any extra permutation.

### 5.4 Meta-data Distribution Optimization

As described in Section 4.2 for a better matrix assembly, PETSc requires meta-data, so the proper structures can be created.

The problem is that, when loading the input file, objects are yet to be created, and since diagonal block sizes change with matrix dimensions and processor number, it is not possible to know which values will be part of the diagonal or non-diagonal sub-matrix. In other words, it is necessary to know the sizes of each processor chunk do separate values into these categories.

This lead to the creation of two extra meta-data structures to count the number of non zeros per row that belong to the diagonal, and non-diagonal blocks. In a nutshell, after file loading and matrix creation, the application processes each matrix entry to determine of it belongs to any processor’s diagonal block. The algorithm shows complexity $O(npz \times np)$, where $nz$ is the number of matrix entries, and $np$ number of processors. Naturally, this solution becomes less viable as number of matrix entries or machines increases.

Yet another problem is that, each processor needs to preallocate the structures for its part of the matrix values, meaning meta-data must be distributed across the processors. To accomplish this, the
matrix global and local dimensions are used so the root machine can calculate which chunk of values belongs to each processor, and distribute the meta-data along the cluster.

### 5.5 Matrix Product Optimization

As described in Section 4.1.1 a three matrix product is required to compute matrix $A$, according to Equation 4.1.

However, since $\hat{s}_{(n)}$ is a diagonal matrix constructed from a vector $s_{(n)}$, scaling a matrix diagonal by $s_{(n)}$ will produce the same result as $G\hat{s}_{(n)}$. Since performing a diagonal scale is faster than a matrix multiplication, matrix $\hat{s}_{(n)}$ is not created, and $s_{(n)}$ is user to directly scale the matrix.

In summary, instead of performing a three matrix multiplication, a matrix scale and a two matrix multiplication are computed.

Yet another optimization, regarding $A$ computation. Also from Equation 4.1 we can see that the same matrix $G$ is used twice. An interesting aspect, is that $G$ remains constant in successive computations of $A$. Consequently, the structure of $A$ will also remain constant, changing only numerical values, as a consequence of $s_{(n)}$ varying.

To take advantage of this, it is possible to compute a transitional matrix $A'$ such that, multiplying each row of the same by $s_{(n)}$, produces an entry in $A$. Each row of $A'$ is the product of two $G$ rows.

Equation 5.1 shows how to compute $A$.

$$A_{ij} = A'(:,:,i\times c+j) \times s_{(n)} \quad (5.1)$$

where $A_{ij}$ denotes the entry at row $i$ and column $j$ of matrix $A$. $c$ is the number of columns of $A$ and $A'(:,:,i)$ denotes the row $i$ of $A'$.

For a better understanding of this operation, consider the example matrix $G$:

$$G = \begin{bmatrix}
1 & 1 & 0 & -1 & 0 & 0 \\
1 & 0 & 1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 & 0 & -1 \\
\end{bmatrix}$$

Computing $A'$ would result in:

$$A' = \begin{bmatrix}
1 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix}$$

The result is $A$, without matrix multiplication. However, since the number of $A'$ rows is equal to the square of $G$ rows, its computation will require large amounts of memory.

This trick is specially interesting when using the algorithm method 1, as explained in Section 4.1. Using only the diagonal of $A$ to perform a Hadamard division, this operation makes $\alpha$ calculation perfectly parallelizable.
In our example, the computation would only require the three diagonal values of $A$, corresponding to $A'$ rows 1, 4 and 8. In this case, computing $A'$ becomes more reliable since the number of rows is equal to the number of rows of $G$. Hence, the computation is done following:

$$A'(:,i) = G(:,i) \times G(:,i)$$ (5.2)

Using this optimization it was possible to considerably reduce the execution time, for algorithm method 1.

5.6 Summary

In this chapter, implementation issues were addressed, discussing their different solutions. Much work is still to be done to fully optimize the algorithm, as the present thesis only focus on the largest computational heavy operations.

The key idea to be taken from this chapter is that, the more diagonal input matrix is, the better the overall performance. Matrices may show an abnormal non-zero structure, and each case should be studied so the optimal permutation can be applied.

The example given in this chapter was a bi-diagonal matrix. It was explained that, by interlacing rows from the second half with the first half, the resulting matrix has a denser diagonal, comparing with the original structure.

Finally, in this project the matrix product optimization was implemented. However, due to matrix sizes, the transitional matrix was only used when using algorithm method 1. The result was a reduction on computational time, comparing with original times.
Chapter 6

Evaluation

Evaluation is a very important phase in a project. In this part, the software is tested to investigate either fulfills the requirements, or not.

For this particular project, it is desirable that the algorithms run faster than sequential ones. On top of that, it should provide the same, if not better, results as the ones from an uniprocessor architecture. Further more, it should be capable of handling growing amount of work and do not use unnecessary memory.

6.1 Overview

Summing up, two qualities are desirable - performance and scalability.

The time a program takes to deliver a solution is one of the most important qualities, when developing a parallel scientific application. Indeed, one of the main goals of using multi-processor architectures, shared or distributed memory, is to obtain faster results, compared with uniprocessor environments.

Since that is one of the main goals of this project, the final product should be evaluated in its performance.

To do so, the computation should be timed and compared with existing sequential solutions.

When developing parallel programs, scalability is another main concern in engineering. Scalability is very close related to performance, and means that, the program should be able to handle growing amount of work, and should be able to adapt in a capable manner to an enlargement to accommodate the growth of work.

This is a very important quality, since a bad scalability often reflect in a bad performance in large environments, usually due to the overhead in communication between identities or memory requirements.

As such, the memory requirements and the number of communications should be monitored.

6.2 Methodology

To test the parallel application, it was available an eleven machine cluster, configured with MPI. These machines have an Intel(R) Core(TM)2 Quad CPU Q6600 processor, and 8GB of memory.

It was also possible to use various types of input problems, from real cases to artificial generated. These problems differ in topology, and most importantly, matrix structure.

Real case problems were available as octave data files. These files vary in problem size, allowing for debug usage or performance measurements.
Regarding artificial files, these were created using an octave program. These problems could be generated with various sizes, providing the expected solution for comparative purposes.

The application ran using `mpirun` command, providing the appropriated arguments.

6.3 Problem Topologies

Four problem topologies were tested. In practice, these problems vary in matrix structure. The tested topologies are listed in Figure 6.1. In each picture, a red dot represents a positive number, whereas a blue dot represents a negative entry.

Additionally, it is important to note that, topologies A and B are from Bureau of Economic Analysis [BEA](https://www.bea.gov). Topology D is a QCEW problem and C is an artificial generated problem. Note that, since Type C topology is artificial generated, matrix non-zero values are equally distributed across rows.

6.4 Results

This section presents the results from various experiments. Each problem was executed using an increasing number of processors. Following the discussion from Chapter 5, in order to achieve compu-
Table 6.1: Comparative view between problem topologies.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Columns</th>
<th>Rows</th>
<th>Matrix Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6 840</td>
<td>2 286</td>
<td>15 792</td>
</tr>
<tr>
<td>B</td>
<td>364 572</td>
<td>73 998</td>
<td>799 830</td>
</tr>
<tr>
<td>C</td>
<td>8 800 000</td>
<td>800 000</td>
<td>16 800 000</td>
</tr>
<tr>
<td>D</td>
<td>2 039 681</td>
<td>1 098 939</td>
<td>5 171 901</td>
</tr>
</tbody>
</table>

tation scalability various hypothesis were considered. Hence, in this section we show the results from each optimization.

Computing in normal execution, i.e. without any changed in matrix structure or optimization, the results were as shown in the charts, as blue column. Note that, some values are not set. In these cases, execution took too much time and was aborted. Furthermore, note that Type D test case contains two different problems for the same topology, differing only on the IO data values. As a result, two columns are used for Type D execution times, each referring to a different problem.

It is also important to note that, our results only display execution times for method 1 of our algorithm.

From the normal execution times, it is understood that our application does not scale well for increasing numbers of processors. In these cases the input data is not equally distributed across machines, limiting the maximum speedup achievable. Moreover, processor diagonal sub-matrices are typically not dense, reducing the potential parallelism.
Our next optimization equally distributes matrix rows and non-zero entries across all machines. The goal is to assign the same amount of work for each processor, maximizing the potential parallelism. The results are shown in the charts as the green column. Type C topology got better results, comparing with normal execution. However, Type B problem became slower for 8 processors. The remaining topologies
had no change in execution times.

Considering the discussion in Section 5.3.1 and since this optimization does not account for the density of diagonal sub-matrices, the results were expected not to scale well. Additionally, Type C topology showed similar results to normal execution. This is expected because, since matrix non-zero entries are equally distributed across rows, processors will have the same amount of work without and balance needed.

Hence, the finally optimization we propose has the goal of transforming Type C matrix structure into a more diagonally heavy one. This is accomplished by interlacing matrix rows, previously addressed in Section 5.3.1. Using this optimization, Type C topology got better execution times and scalability. The red columns in the charts shows the resulting execution times for several problems.

This is expected because, as addressed in Section 5.3.1, diagonal sub-matrices became more dense, and non-diagonal sub-matrices became more sparse. This means that, by changing matrix structure, parallelism potential for matrix-vector products had increased.

Furthermore, since the number of matrix entries in diagonal matrix decreases with increasing number of processors, performance deteriorates after number of machines increases to 8 and 16.

### 6.5 Summary

In this Chapter we presented the results for our application, using method 1. We explored the effects of various matrix non-zero patters in computation scalability. Hence, the program was tested using 4 different topologies. Furthermore, the tests ran for different number of processors, varying from 1 to 16 machines. The test cases used came from various sources, such as from real and artificial generated data.

In a first phase we tested the application without any optimization, and since no data manipulation was made, processors had different workloads. This lead to a limitation in parallel potential and bad scalability.

On the second phase matrix rows and non-zero entries were equally distributed among machines. However, results still showed bad scalability for and increasing number of processors. This was due to the fact that diagonal sub-matrices did not contain most of matrix non-zero values. As such, we tested our application with a third optimization with the goal of changing matrix structure into a more diagonal heavy one. This optimization was specially targeted at topology Type C.

With this optimization we got better results in Type C topology. However, computational speedup decline after the number of machines reaches 8. This is expected since the number of non-zero elements in diagonal sub-matrix will decrease for increasing number of processors.
Chapter 7

Conclusions

7.1 Project Summary

Data balancing is important for areas that demand large amounts of data. This process requires mathematical operations between high dimension matrix and vectors, such as matrix-vector products, matrix-matrix products and linear system solving. However, due to matrix and vector dimensions, this process requires large amounts of computational resources, such as time and memory.

This thesis attempts to provide a parallel implementation of a WLS algorithm of Rodrigues [4] to balance data, in order to reduce processing times. Accordingly to WLS algorithm of Rodrigues, three district methods to balance input data are supported. All methods share the same algorithm logic, differing on the method used to compute a Lagrange parameter $\alpha$, needed each WLS iteration. Method 1 uses a simple element-wise division between two vectors. Methods 2 and 3 require the solution to a sparse linear system of equations. Essentially, the algorithm requires the computation of multiple matrix-vector products, as well as matrix-matrix products. In conjunction with linear system solving, these are the main computational challenges in this project.

To support our implementation two options were considered: develop a new library to provide the basic blocks for our application; or use already existing scientific software. From this discussion, we decided to use the library PETSc since it provides parallel implementations for all operations needed in this work. This was the best option because we did not intend to develop software that already existed, specially when these libraries provide well optimized functionality. Using PETSc we implemented our algorithm, targeting distributed memory machines.

The main difficulty found when developing our program was the low computational scalability. Indeed, when testing, our application did not perform for an increasing number of machines. For this reason, three optimizations were considered and tested: the first balanced the number of matrix entries and rows, equally across all machines, attempting to assign the same work load in all machines; the second optimization focused on diagonalize the input matrix, changing its structure intro a more appealing format for palatalization; finally, the third optimization focused on permuting the input matrix with the goal of changing the sparsity of its diagonal and non-diagonal sub-matrices. With this we attempted to take advantage of the distributed matrix structure to obtain better results in parallel matrix-vector products.

These tests helped us understand which matrix permutations work best for each case study. In a nutshell, different matrices require different permutations to achieve computation scalability. As such, from this analysis, we manage to obtain better results using the third optimization, for a case study with bi-diagonal matrix.
7.2 Future Work

Despite our attempt to find a scalable, multiprocessor solution for this problem, our efforts showed to be insufficient for all topologies. We think it is possible to improve computational performance using adequate transformations for each matrix type. Since our application does not scale for many problem topologies, it is important to research permutations to transform matrices into a more parallel efficient structures. This aspect is important as it dictates the difference between scalable and non-scalable executions.

Another issue to be investigated is the linear system solving. With our implementation we could not use algorithm methods 2 and 3 to find a solution for test problems because linear system solving was not providing the expected solutions. Future work should focus on finding a reason for this unwanted behavior.

Additionally, other important aspect to complete our solution is to account for ill composed input data. Essentially, our algorithm does not account for input data with singular matrices. At present time, the user is responsible for eliminating the redundant rows from the input matrix. However, it is desirable that input data should be processed in order to make it usable by the application.
Bibliography


