Parallelization of the Kaczmarz Algorithm

Inês Alves Ferreira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. José Carlos Alves Pereira Monteiro
Prof. Juan Antonio Acebrón Torres

Examination Committee

Chairperson: Prof. Pedro Filipe Zeferino Aidos Tomás
Supervisor: Prof. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Luís Miguel Teixeira D'Avila Pinto da Silveira

June 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Firstly, I extend my deepest gratitude to my advisors, Professor José Monteiro and Professor Juan Acebron, for agreeing to guide me in this project. Your support and feedback have been valuable, and I am grateful for your availability in addressing my every question.

I would also like to thank the opportunity to participate in the project ”Advanced Computing/HPC MSc Fellows Programme” and EuroCC for the financial support.

To my friends from MEFT, Zeca, Bernardo, Tomás, Manel and Neves, I want to express my appreciation for the joy and companionship you bring. A special thank you to Rita and Filipa for not only making lab assignments and projects bearable but for being such important and special people in my life. The memories we have created together over the past six years are cherished, and I hope that many more are yet to come.

To the people I met in MEEC, Rodrigo and Tomás, I am grateful for the warm welcome into your group. Additionally, I want to thank Ricardo for patiently enduring my numerous questions and offering words of encouragement. You have been an exceptional work partner and a true friend.

Last but not least, I would like to express my heartfelt appreciation to my parents for your sacrifices, for encouraging me, and for supporting my every decision. And to my brother Miguel, thank you for guiding me throughout my academic path, listening to my concerns, and believing in me.

Abstract

The Kaczmarz algorithm is an iterative method that solves linear systems of equations. It stands out among iterative algorithms when dealing with large systems for two reasons. Firstly, in each iteration, the Kaczmarz algorithm uses a single equation, resulting in minimal computational work per iteration. Secondly, solving the entire system may only require a small subset of equations. These characteristics have attracted significant attention to the Kaczmarz algorithm. Researchers have observed that randomly choosing equations can improve the convergence rate of the algorithm. This insight led to the development of the Randomized Kaczmarz algorithm and, subsequently, several other variations emerged.

In this thesis, we analyze the behavior of the Kaczmarz method and its sequential variations. We found that a randomized version of the algorithm that samples equations without replacement can outperform both the original and Randomized Kaczmarz methods.

Additionally, we explore approaches to parallelizing the Kaczmarz method. In particular, we implement the Randomized Kaczmarz with Averaging method that, for noisy systems, unlike the standard Kaczmarz algorithm, reduces the final error of the solution. While efficient parallelization of this algorithm is not achievable, we introduce a block version of the averaging method that exhibits significantly improved speedups compared to its sequential counterpart.

Lastly, we apply the Kaczmarz method to solve real-world problems involving noisy linear systems derived from computed tomography. The objective is to obtain reconstructed images that minimize the reconstruction error. We show that, by incorporating known image constraints, the error of the solution is significantly reduced.

Keywords

Linear systems; Iterative algorithms; Parallel and distributed computing; Rate of convergence; Least-Squares problem; Image reconstruction

Resumo

O algoritmo de Kaczmarz é um método iterativo que resolve sistemas lineares de equações. Destaca-se de entre outros algoritmos iterativos ao resolver sistemas com grandes dimensões por duas razões. Em primeiro lugar, em cada iteração, o algoritmo de Kaczmarz utiliza uma única equação, resultando em trabalho computacional reduzido. Em segundo lugar, pode ser apenas necessário um pequeno subconjunto de equações do sistema para o resolver. Estas características têm atraído grande atenção para o algoritmo de Kaczmarz. Investigadores observaram que escolher as equações aleatoriamente pode melhorar a velocidade de convergência do algoritmo, o que levou ao desenvolvimento do algoritmo aleatório de Kaczmarz que, consequentemente, causou o aparecimento de variações do mesmo.

Nesta tese, analisamos o comportamento do método de Kaczmarz e das suas variações sequenciais. Descobrimos que a versão aleatória do algoritmo que escolhe equações sem reposição pode superar tanto o método original como a versão aleatória.

Além disso, exploramos abordagens para paralelizar o método de Kaczmarz. Em particular, implementamos o método aleatório de Kaczmarz com média que, para sistemas com ruído, ao contrário do algoritmo original de Kaczmarz, reduz o erro final da solução. Embora não seja possível obter uma parallelização eficiente deste algoritmo, apresentamos uma versão por blocos do método anterioremente referido que apresenta aumentos significativos nos tempos de execução em relação à sua contraparte sequencial.

Por fim, aplicamos o método de Kaczmarz para resolver problemas reais que envolvem sistemas lineares com ruído derivados de problemas de tomografia computadorizada. O objetivo é obter imagens reconstruídas que minimizem o erro de reconstrução. Mostramos que, incorporando restrições que sabemos que são verificadas pela imagem, o erro da solução é significativamente reduzido.

Palavras Chave

Sistemas lineares; Algoritmos iterativos; Computação paralela e distribuída; Velocidade de convergência; Problema de mínimos quadrados; Reconstrução de imagens

Contents

List of Figures

List of Tables

Acronyms

AsyRK	Asynchronous Parallel Randomized Kaczmarz
CARP	Component-Averaged Row Projections
CG	Conjugate Gradient
CT	Computed Tomography
CGLS	Conjugate Gradient for Least-Squares
CK	Cyclic Kaczmarz
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
GD	Gradient Descent
GSSRK	Gramian Selectable Set Randomized Kaczmarz
GRK	Greedy Randomized Kaczmarz
NSSRK	Non-Repetitive Selectable Set Randomized Kaczmarz
RBK	Randomized Block Kaczmarz
RDBK	Randomized Double Block Kaczmarz
REK	Randomized Extended Kaczmarz
RGS	Randomized Gauss-Seidel
RK	Randomized Kaczmarz
RKA	Randomized Kaczmarz with Averaging
RKAB	Randomized Kaczmarz with Averaging with Blocks
SGD	Stochastic Gradient Descent
SCD	Stochastic Coordinate Descent
SRK	Simple Randomized Kaczmarz

SRKAWOR	Simple Randomized Kaczmarz with Averaging Without Replacement
SRKABWOR	Simple Randomized Kaczmarz with Averaging with Blocks Without Replacement
SRKWOR	Simple Randomized Kaczmarz Without Replacement
SSRK	Selectable Set Randomized Kaczmarz

Chapter 1

Introduction

Solving linear systems of equations is a fundamental part of linear algebra and, consequently, of mathematics. One example of the application of solving linear systems in the real-world are problems derived from computed tomography. During a CT scan, one has to reconstruct an image of the scanned body using radiation data measured by detectors. Nonetheless, physical quantities are always measured with some error and, since CT data is not an exception, the problem of reconstructing images is hampered by noise. Therefore, it is important to create efficient computer algorithms for linear systems that can accurately find the solution that minimizes the error, especially for large-scale problems that generally take longer to solve.

There are two types of numerical methods used to solve linear systems of equations: direct methods and iterative methods. A direct method is characterized by a closed-form solution that can be computed in a finite number of steps. Direct methods compute solutions with high levels of precision but they can take a long time to compute, especially if matrices are large. Iterative methods calculate an approximate solution that hopefully improves with the number of iterations. Depending on the required precision of the solution, iterative methods can be faster than direct methods, particularly for large systems.

There are two special classes of iterative methods: row-action methods and column-action methods. These make use of a single row or column of the system's matrix per iteration. Each iteration of these methods has diminutive computational work compared with other iterative methods that use the entire matrix. An example of a row-action algorithm is the Kaczmarz algorithm, the focus of this dissertation.

Many real-world problems consist of large linear systems. Methods like Kaczmarz that use one equation at a time can be used in real-time while data is being collected. Furthermore, if the amount of data is so large that storing it in a machine is not possible, row/column action methods are still able to solve the system.

To improve the performance of algorithms we can use parallel computing. Parallelization, in the context of parallel and distributed computing, refers to the technique of breaking down a computational task or workload into smaller sub-tasks that can be executed simultaneously or in parallel. It involves dividing the problem into multiple independent parts and assigning each part to a separate processing unit or computing resource.

There are two types of parallelization: using shared memory or using distributed memory. In shared memory, the processing units correspond to cores inside a single machine; in distributed memory, several machines can be connected physically or within a network. Since there are advantages and disadvantages to both approaches, it is also an option to combine shared and distributed memory.

The main goal of parallelization is to improve performance by decreasing the execution time of a given task. However, parallelizations that use distributed memory have the ability to process large amounts of data that could not be processed if we were to use one machine only. Parallelization is not a straightforward technique since we must consider data dependencies between tasks, the communication overhead between processing units, and synchronization points, among others.

1.1 Problem Statement

Many variations of the original Kaczmarz method have been proposed in the literature over the years. However, there is no investigation that compares all the variations between themselves. Our first task is to analyze the behavior of Kaczmarz-based methods that sample matrix rows according to different criteria. More specifically, we evaluate the relationship between the number of iterations needed to find a solution and the corresponding execution time for systems with different dimensions.

After a thorough analysis of the sequential variations of the Kaczmarz method, we parallelize the method using shared and distributed memory separately and the combination of both.

Lastly, we apply our investigation of the Kaczmarz method to a real-world problem consisting in reconstructing images provided by computed tomography data.

1.2 Organization of the Document

The organization of this document is as follows. In Chapter 2 we describe the several types of linear systems and respective solutions and how they can be obtained. Furthermore, we introduce the original algorithm (Cyclic Kaczmarz algorithm) and the Randomized Kaczmarz algorithm as iterative methods that solve linear systems, together with other randomized variations. In Chapter 3 we introduce some of the methods inspired by the Randomized Kaczmarz algorithm and present some attempts to parallelize the Kaczmarz method. In Chapter 4 we present the implementation details of the sequential version of the algorithm and some of its variations, together with the experimental results. In Chapters 5 and 6 we discuss several approaches to parallelizing the Kaczmarz method using shared and distributed memory, respectively. In Chapter 7 we apply the Kaczmarz method to inconsistent systems for computed tomography problems. Finally, in Chapter 8, we conclude this dissertation and discuss future work.

Chapter 2

Systems of Linear Equations: Properties and Algorithms

In the first section of this chapter, we outline the categorization of linear systems based on factors such as matrix dimensions and the existence of a solution. The following section elaborates on the methods used to solve linear systems and the corresponding computational algorithms employed. The chapter culminates by presenting the Kaczmarz algorithm, along with its key attributes, and subsequently introduces some randomized variants.

2.1 Types of Linear Systems

A linear system of equations can be written as

$$Ax = b\,, \tag{2.1}$$

where A is an $m \times n$ matrix, $x \in \mathbb{R}^n$ is called the solution and b is a vector in $\mathbb{R}^m$.

Linear systems can be classified based on distinct criteria. When we consider the existence of a solution, systems can be categorized as either **consistent** or **inconsistent**. For a **consistent** system there is at least one value of x that satisfies all the equations in the system. However, it is possible that there are equations that constradict each other and, in that case, the system is said **inconsistent**.

When considering the relationship between the number of equations and variables, systems can be classified as **overdetermined** or **underdetermined**. When $m \geq n$, there are more equations than variables, and the system is said to be **overdetermined**. If an **overdetermined** system is **consistent** and there is a single solution that satisfies (2.1), we denote it by x^*. Otherwise, if the system is **inconsistent**, we are usually interested in finding the least-squares solution, that is

$$x_{LS} = arg\min_{x} \|Ax - b\|_2^2\,. \tag{2.2}$$

In real-world overdetermined systems, it is more frequent to have inconsistent systems than consistent systems. The least-squares solution can be computed using the normal equations $x_{LS} = (A^TA)^{-1}A^Tb = A^\dagger b$, where $A^\dagger$ is the Moore–Penrose inverse or pseudoinverse (demonstrated in Section A.1 of Appendix A). If the system has a matrix with $m = n$, we are in the presence of an exactly determined system.

Underdetermined systems verify $m < n$, meaning that we have fewer equations than variables. If the system is **inconsistent** there is no solution. On the other hand, if there is no contradiction between equations, we will still have $m-n$ degrees of freedom and the system is **consistent** with infinite solutions. In this case, we are often interested in the least Euclidean norm solution,

$$x_{LN} = arg\min_{x} \|x\|_2 \quad \text{subject to} \quad Ax = b\,, \tag{2.3}$$

which can be computed using $x_{LN} = A^T(AA^T)^{-1}b$ (demonstrated in A.2 of Appendix A).

2.2 How to Solve Linear Systems

There are two classes of numerical methods that solve linear systems of equations: direct and indirect or iterative. Direct methods compute the solution of the system in a finite number of steps. Although solutions given by direct methods have a high level of precision[1], the computational cost and memory usage can be high for large matrices when using these methods. Some widely used direct methods are Gaussian Elimination and LU Factorization. Iterative methods generate a sequence of approximate solutions that get increasingly closer to the exact solution with each iteration. Iterative methods are generally less computationally demanding than direct methods and their convergence rate[2] is highly dependent on properties of the matrix of the system. The precision of the solutions given by iterative methods can be controlled by external parameters, meaning that there is a trade-off between the running time and the desired accuracy of the solution. Therefore, if high precision is not a requirement, iterative methods can outperform direct methods, especially for large and sparse matrices. Some popular iterative methods are the Jacobi method and the Conjugate Gradient (CG) method.

A particular class of iterative methods is row-action methods. These use only one row of matrix A in each iteration, meaning that the computational work per iteration is small compared to methods that make use of the entire matrix. There are also column-action methods that, instead of using a single row per iteration, use a single column. An example of a row action algorithm is the Kaczmarz method which is introduced in the following section.

2.3 The Kaczmarz Method

2.3.1 Definition

The Kaczmarz method [2] is an iterative algorithm that solves consistent linear systems of equations $Ax = b$, where A is a $m \times n$ matrix with $m \geq n$ and $b \in \mathbb{R}^m$. Let $A^{(i)}$ be the i-th row of A and b_i be the

[1]Precision can be defined as the Euclidean distance of the difference between the obtained solution and the real solution of the system.

[2]The rate of convergence [1] of an algorithm quantities how quickly a sequence approaches its limit. In the case of iterative methods, a higher rate of convergence means that the method will take less iterations to approach the solution.

i-th coordinate of b. The original version of the algorithm can then be written as

$$x^{(k+1)} = x^{(k)} + \alpha_i \frac{b_i - \langle A^{(i)}, x^{(k)} \rangle}{\|A^{(i)}\|_2^2} A^{(i)T} , \quad \text{with} \quad i = k \bmod m , \tag{2.4}$$

with k starting at 0 and where α_i is a relaxation parameter that is set to 1. This means that, in each iteration, the estimate of the solution will satisfy a different constraint, until a point is reached where all the constraints are satisfied. Since the rows of matrix A are used in a cyclic manner, this algorithm is also known as the Cyclic Kaczmarz (CK) method. The Kaczmarz method also converges to x_{LN} in the case of underdetermined systems. For more details see Section 3.3 of [3].

2.3.2 Geometric Interpretation

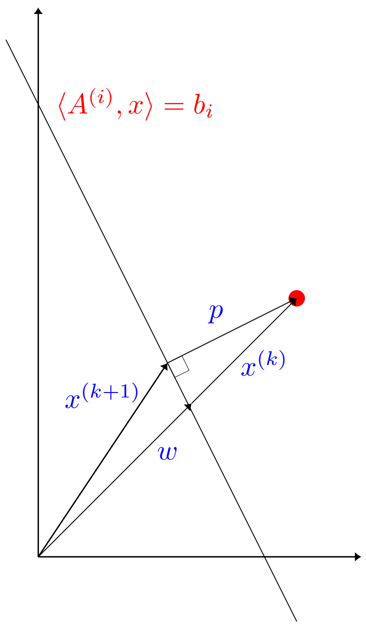


Figure 2.1: Geometric interpretation of a single iteration of the Kaczmarz method.

This section presents a geometric interpretation of the Kaczmarz method. We define the hyperplane $H_i = x : \langle A^{(i)}, x\rangle = b_i$. Each iteration $x^{(k+1)}$ can be thought of as the projection of $x^{(k)}$ onto H_i, which we will now prove. Let us assume that the number of columns of A is 2 so that we can visualize the system in a plane, as demonstrated in Figure 2.1. We will now show that $x^{(k+1)}$ is given by (2.4) (if α_i is set to one). We start by observing that

$$x^{(k+1)} = x^{(k)} - p . \tag{2.5}$$

Note that p is the projection of $x^{(k)} - w$ into $A^{(i)}$, since $A^{(i)}$ is a normal vector to the hyperplane $\langle A^{(i)}, x\rangle = b_i$. To find an expression for p, we first need to find w, which is just $x^{(k)}$ multiplied by a constant, such that $w = k\, x^{(k)}$. The constant k can be found using the constraint that w has to belong to the hyperplane, that is, $\langle A^{(i)}, kx^{(k)}\rangle = b_i \Rightarrow k = b_i/\langle A^{(i)}, x^{(k)}\rangle$. Since the projection of the column vector u onto the column vector v is given by

$$\left(\frac{\langle u, v\rangle}{\|v\|^2}\right)v\,, \tag{2.6}$$

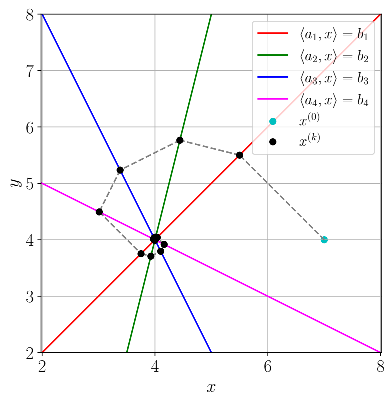
we can use it to calculate p, the projection of $x^{(k)} - w$ into $A^{(i)}$. Note that $A^{(i)}$ is a line of the matrix A, so we must use ${A^{(i)}}^T$ to represent it as a column vector. The result is

$$p = \frac{\langle A^{(i)}, (1-k)x^{(k)}\rangle}{\|A^{(i)}\|^2}{A^{(i)}}^T = \left(1 - \frac{b_i}{\langle A^{(i)}, x^{(k)}\rangle}\right)\frac{\langle A^{(i)}, x^{(k)}\rangle}{\|A^{(i)}\|^2}{A^{(i)}}^T = \frac{\langle A^{(i)}, x^{(k)}\rangle - b_i}{\|A^{(i)}\|^2}{A^{(i)}}^T\,. \tag{2.7}$$

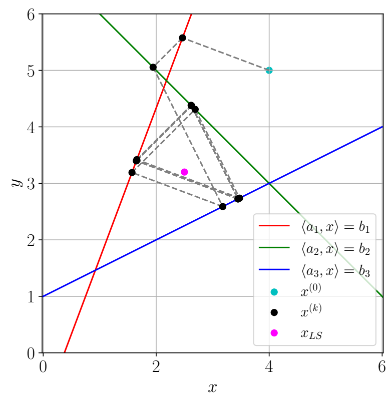
Finally we can obtain $x^{(k+1)}$ using (2.5) such that

$$x^{(k+1)} = x^{(k)} - p = x^{(k)} - \frac{\langle A^{(i)}, x^{(k)}\rangle - b_i}{\|A^{(i)}\|^2}{A^{(i)}}^T = x^{(k)} + \frac{b_i - \langle A^{(i)}, x^{(k)}\rangle}{\|A^{(i)}\|^2}{A^{(i)}}^T\,. \tag{2.8}$$

Now that we have shown the process of computing a single iteration, we will now show how the estimate of the solution evolves throughout several iterations.



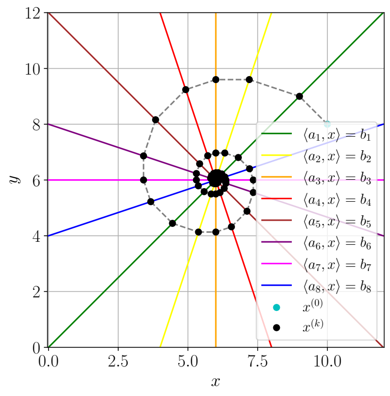
(a) Example of a consistent system with an unique solution.



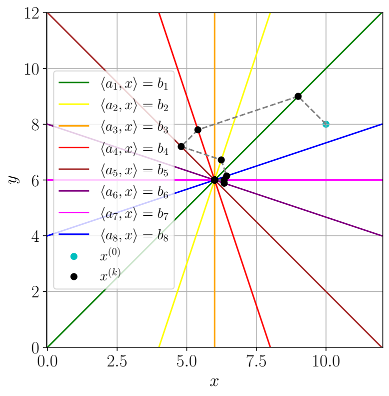
(b) Example of an inconsistent system and its least-squares solution.

Figure 2.2: Convergence of the Kaczmarz method in 2 dimensions for solvable and nonsolvable systems.

Figure 2.2 provides the geometrical interpretation of a linear system with 4 equations in 2 dimensions: each equation of the system is represented by a line in a plane; the evolution of the estimate of the solution, $x^{(k)}$, is also shown throughout several iterations. Since we are using the rows of the matrix in a cyclical fashion, we start by projecting the initial estimate of the solution onto the first equation and then the second, and so on. The difference between Figures 2.2a and 2.2b is the following: Figure 2.2a shows a consistent system with a unique solution, x^*, the point where all equations meet. It is clear that the estimate of the solution given by the Kaczmarz method encounters x^* after some time; in Figure 2.2b we have an inconsistent system with its least-squares solution marked in magenta. Note that it is impossible for the Kaczmarz to reach it since the estimate of the solution will always remain a certain distance from x_{LS}. In the next section, we will show how this distance can be quantified.



(a) Example of the convergence of the Kaczmarz method for a consistent solvable system using cyclical selection of rows.



(b) Example of the convergence of the Kaczmarz method for a consistent solvable system using random selection of rows.

Figure 2.3: Convergence of the Kaczmarz method in 2 dimensions using two different row selection criteria.

Figure 2.3 shows the special case of a consistent system with a highly coherent matrix. In these systems the angle between consecutive rows of the matrix is small. If we go through the matrix cyclically, shown in Figure 2.3a, the convergence is quite slow since the change in the solution estimate is minimal. On the other hand, in Figure 2.3b, where the rows of the matrix are used in a random fashion, the estimate of the solution approaches the system solution much faster. This was experimentally observed and led to the development of randomized versions of the Kaczmarz method.

2.4 Randomized Kaczmarz Method

Kaczmarz [2] showed that, if the linear system is solvable, the method converges to the solution x^*, but the rate at which the method converges is very difficult to quantify. Known estimates for the rate of convergence of the cyclic Kaczmarz method rely on quantities of matrix A that are hard to compute and that make it difficult to compare with other algorithms (for some examples see references [4–6]). Ideally, we would like to have a rate of convergence that depends on the condition number of the matrix A, which is difficult for this algorithm since it relies on the order of the rows of the matrix, and the condition number is only related with geometric properties of the matrix.

It has been observed [7–9] that, instead of using the rows of A in a cyclic manner, choosing rows randomly can improve the rate of convergence of the algorithm. To tackle the problem of finding an adequate rate of convergence for the Kaczmarz method and to try to explain the empirical evidence that randomization accelerates convergence, Strohmer and Vershynin [10] introduced a randomized version of the Kaczmarz method that converges to x^*. Instead of selecting rows in a cyclical fashion, in the randomized version, in each iteration, we use the row with index i, chosen at random from the probability distribution

$$P\{i = l\} = \frac{\|A^{(l)}\|^2}{\|A\|_F^2} \quad (l = 1, 2, ..., m) \,. \tag{2.9}$$

This version of the algorithm is called the Randomized Kaczmarz (RK) method. Strohmer and Vershynin proved that RK has exponential error decay, also known as linear convergence [3]. The rate of convergence depends only on the scaled condition number of A, $\kappa(A)$, and not on the number of equations in the system. Let x_0 be the initial guess, x^* be the solution of the system, and $\sigma_{min}(A)$ be the smallest singular value of matrix A. The expected convergence of the algorithm for a consistent system can then be written as

$$\mathbb{E}\,\|x^* - x^{(k)}\|^2 \leq (1 - \kappa(A)^{-2})^k \|x^* - x^{(0)}\|^2 \leq \Big(1 - \frac{\sigma^2_{min}(A)}{\|A\|_F^2}\Big)^k \|x^* - x^{(0)}\|^2 \,. \tag{2.10}$$

Later, Needell [11] extended these results for inconsistent systems, showing that RK reaches an estimate that is within a fixed distance from the solution. This error threshold is dependent on the matrix A and can be reached with the same rate as in the error-free case. The expected convergence proved by Strohmer and Vershynin can then be extended such that

$$\mathbb{E}\,\|x^* - x^{(k)}\|^2 \leq \Big(1 - \frac{\sigma^2_{min}(A)}{\|A\|_F^2}\Big)^k \|x^* - x^{(0)}\|^2 + \frac{\|r_{LS}\|^2}{\sigma^2_{min}(A)}. \tag{2.11}$$

where $r_{LS} = b - Ax_{LS}$ is the least-squares residual. This extra term is called the convergence horizon and is zero when the linear system is consistent.

This analysis shows that it is not necessary to know the whole system to solve it, but only a small part of it. Strohmer and Vershynin show that in extremely overdetermined systems the Randomized Kaczmarz method outperforms all other known algorithms and, for moderately overdetermined systems, it outperforms the celebrated conjugate gradient method. They reignited not only the research on the Kaczmarz method but also triggered the investigation into developing randomized linear solvers.

Chen [12] showed the Kaczmarz method can be seen as a special case of other row action methods. A case worth mentioning is that the Randomized Kaczmarz method can be derived from the Stochastic Gradient Descent (SGD) method using certain parameters. The SGD method is widely used in machine learning, meaning that parallelizations of this algorithm can be adapted to the Randomized Kaczmarz method. An analysis of the relationship between the two methods is present in Appendix B.

2.5 Simple Randomized Kaczmarz Method

Apart from the comparison between the original Kaczmarz method with the Randomized Kaczmarz method, Strohmer and Vershynin [10] also compared both these methods with the Simple Randomized Kaczmarz (SRK). In this method, instead of sampling rows using their norms, rows are sampled using a uniform probability distribution. Although they did not prove the convergence rate for this method,

[3]The concept of linear convergence in numerical analysis is often referred to, by mathematicians, as exponential convergence.

Schmidt [13] did:

$$\mathbb{E}\,\|x^* - x^{(k)}\|^2 \leq \left(1 - \frac{\sigma_{min}^2(A)}{m\|A\|_\infty^2}\right)^k \|x^* - x^{(0)}\|^2. \tag{2.12}$$

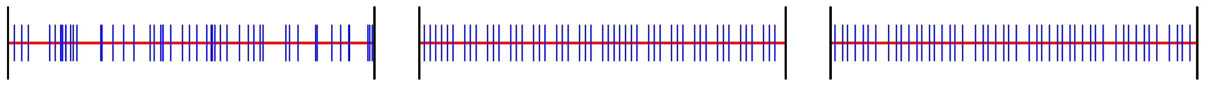
Schmidt also mentions that RK should be at least as fast as SRK, and faster if any two rows don't have the same norm.

2.6 Sampling Rows Using Quasirandom Numbers

In Section 2.3.2 we discussed how sampling rows in a random fashion instead of cyclically can improve convergence for highly coherent matrices. Furthermore, convergence should be faster if indices corresponding to consecutively sampled rows are not close so that the angle between these rows is not small. However, when sampling random numbers, these can form clumps and they can be poorly distributed, meaning that even if we sample rows randomly, these can still be very close regarding their position in the matrix and, consecutively, in the angle between them. To illustrate the importance of how the generation of random numbers affects the results of the Kaczmarz algorithm, Figure 2.4a shows 50 random numbers sampled from the interval $[1, 1000]$ using a uniform probability distribution, illustrating the process of sampling row indices for a matrix with 1000 rows. Note that there are, simultaneously, clumps of numbers and areas with no sampled numbers.

This is where quasirandom numbers, also known as low-discrepancy sequences, come in: they are sequences of numbers that are evenly distributed, meaning that there are no areas with a very low or very high density of sampled numbers. Quasirandom numbers have been shown to improve the convergence rate of Monte-Carlo-based methods, namely methods related to numerical integration [14].

Several low-discrepancy sequences can be used to generate quasirandom. Here we will work with two sequences that are widely used: the Halton sequence [15] and the Sobol sequence [16]. To show that indeed these sequences can generate numbers that are more evenly distributed than the ones generated using pseudo-random numbers, Figures 2.4b and 2.4c show 50 sampled numbers from the interval $[1, 1000]$ using the Sobol and Halton sequences.



(a) Random numbers generated using an uniform probability distribution. **(b)** Quasirandom numbers generated using the Sobol sequence. **(c)** Quasirandom numbers generated using the Halton sequence.

Figure 2.4: Distribution of 50 sampled numbers in interval $[1, 1000]$.

Chapter 3

The Revival of Row and Column Action Methods

The work by Strohmer and Vershynin in [10] motivated other developments in row/column action methods by randomizing classical algorithms. In this chapter, we present several iterative methods, some of which are modifications to the RK method. We conclude with some attempts at parallelization of both the Cyclic and Randomized Kaczmarz methods.

3.1 Randomized Coordinate Descent Method

Leventhal and Lewis [17] developed the Randomized Coordinate Descent Method, also known as Randomized Gauss-Seidel (RGS) algorithm. Like RK, for overdetermined consistent systems, this algorithm converges to the unique solution x^*. Unlike RK, for overdetermined inconsistent systems, RGS converges to the least squares solution x_{LS}; for underdetermined consistent systems RGS does not converge to the least euclidean norm solution x_{LN} [3]. RGS is an iterative method that uses only one column of matrix A in each iteration which is chosen at random from the probability distribution

$$P\{j=l\} = \frac{\|A_{(l)}\|^2}{\|A\|_F^2} \quad (l = 1, 2, ..., n)\,. \tag{3.1}$$

It also uses an intermediate variable $r \in \mathbb{R}^m$. The algorithm has three steps to be completed in a single iteration:

$$\alpha^{(k)} = \frac{\langle A_{(j)}, r^{(k)}\rangle}{\|A_{(j)}\|^2}\,, \tag{3.2}$$

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} e_{(j)}\,, \quad \text{and} \tag{3.3}$$

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} A_{(j)}\,, \tag{3.4}$$

where $e_{(j)}$ is the jth coordinate basis column vector (all zeros with a 1 in the jth position) and r is initialized to $b - Ax^{(0)}$. Leventhal and Lewis also showed that, like RK, RGS converges linearly in expectation. Later, Needell, Zhao, and Zouzias [18] developed a block version of the RGS method that uses several columns in each iteration, with the goal to improve the convergence rate. For further discussion see Section C.1 in Appendix C.

3.2 Randomized Extended Kaczmarz Method

The Randomized Kaczmarz method can only be applied to solvable linear systems but most systems in real-world applications are affected by noise, and are consequently inconsistent, meaning that it is important to develop a version of RK that solves least-squares problems. To extend the work developed by Strohmer and Vershynin [10] to inconsistent systems, Zouzias and Freris [19] introduced the Randomized Extended Kaczmarz (REK) method. This method is a combination of the Randomized Orthogonal Projection algorithm [20] together with the Randomized Kaczmarz method and it converges linearly in expectation to the least-squares solution, x_{LS}. The algorithm is a mixture of a row and column action method since, in each iteration, we use one row and one column of matrix A. Rows are chosen with probability proportional to the row norms and columns are chosen with probability proportional to column norms. In each iteration, we use a row with index i, chosen from the probability distribution in (2.9) and a column with index j, chosen at random from the probability distribution (3.1). Each iteration of the algorithm can then be computed in two steps. In the first one, the projection step, we calculate the auxiliary variable $z \in \mathbb{R}^m$. In the second one, the Kaczmarz step, we use z to estimate the solution x. Variables z and x are initialized, respectively, to b and 0. The two steps of each iteration are then

$$z^{(k+1)} = z^{(k)} - \frac{\langle A_{(j)}, z^{(k)} \rangle}{\|A_{(j)}\|^2} A_{(j)}\,, \quad \text{and} \tag{3.5}$$

$$x^{(k+1)} = x^{(k)} + \frac{b_i - z_i^{(k)} - \langle x^{(k)}, A^{(i)} \rangle}{\|A^{(i)}\|^2} A^{(i)T}\,. \tag{3.6}$$

The rate of convergence can be described by

$$\mathbb{E}\,\|x_{LS} - x^{(k)}\|^2 \leq \Big(1 - \frac{\sigma_{min}^2(A)}{\|A\|_F^2}\Big)^{\lfloor k/2 \rfloor} \Big(1 + 2\,\frac{\sigma_{max}^2(A)}{\sigma_{min}^2(A)}\Big)\,\|x_{LS}\|^2\,. \tag{3.7}$$

Later on, Needell, Zhao, and Zouzias [18] developed a variation of this algorithm that uses blocks to accelerate convergence. For further discussion see Section C.2 in Appendix C.

3.3 Greedy Randomized Kaczmarz Method

The Greedy Randomized Kaczmarz (GRK) method introduced by Bai and Wu [21] is a variation of the Randomized Kaczmarz method with a different row selection criterion. Note that the selection criterion for rows in the RK method can be simplified to uniform sampling if we scale matrix A with a diagonal matrix that normalizes the Euclidean norms of all its rows. But, in iteration k, if the residual vector $r^{(k)} = b - Ax^{(k)}$ has $|r^{(k)}(i)| > |r^{(k)}(j)|$, we would like for row i to be selected with a higher probability than row j. In summary, GRK differs from RK by selecting rows with larger entries of the residual vector with higher probability. Each iteration is still calculated using (2.4), the row index i chosen in iteration k

is computed using the following steps:

$$\text{Compute} \quad \epsilon_k = \frac{1}{2}\Big(\frac{1}{\|b - Ax^{(k)}\|_2^2} \max_{1\le i\le m} \Big\{\frac{|b_i - A^{(i)}x^{(k)}|^2}{\|A^{(i)}\|_2^2}\Big\} + \frac{1}{\|A\|_F^2}\Big) \tag{3.8}$$

$$\text{Determine the index set of positive integers} \quad \mathcal{U}_k = \Big\{i : |b_i - A^{(i)}x^{(k)}|^2 \ge \epsilon_k \|b - Ax^{(k)}\|_2^2 \, \|A^{(i)}\|_2^2\Big\} \tag{3.9}$$

$$\text{Compute vector} \quad \tilde{r}^{(k)}{}^{(i)} = \begin{cases} b_i - A^{(i)}x^{(k)}, & \text{if } i \in \mathcal{U}_k \\ 0, & \text{otherwise} \end{cases} \tag{3.10}$$

$$\text{Select } i_k \in \mathcal{U}_k \text{ with probability} \quad P\{i_k = i\} = \frac{|\tilde{r}^{(k)}{}^{(i)}|^2}{\|\tilde{r}^{(k)}\|_2^2} \tag{3.11}$$

The Greedy Randomized Kaczmarz method presents a faster convergence rate when compared to the Randomized Kaczmarz method, meaning that it is expected for GRK to outperform RK.

3.4 Selectable Set Randomized Kaczmarz Method

Just like the Greedy Randomized Kaczmarz method, the Selectable Set Randomized Kaczmarz (SSRK) method [22] is a variation of the Randomized Kaczmarz method with a different probability criterion for row selection that avoids sampling equations that are already solved by the current iterate. The set of equations that are not yet solved is referred to as the selectable set. In each iteration of the algorithm, the row to be used in the Kaczmarz step is chosen from the selectable set. The algorithm is as follows

- Sample row i_k according to probabilities $p_1, ..., p_m$ with rejection until $i_k \in \mathcal{S}_k$;
- Compute the estimate of the solution $x^{(k+1)}$;
- Update the selectable set $\mathcal{S}_{k+1}$ such that $i_k \notin \mathcal{S}_{k+1}$.

The first step of the algorithm can be achieved by repeatedly sampling i_k according to the same probabilities as the ones used in RK until we get a row that is present in the selectable set; or, one can change the probability distribution such that the probability of a row is 0 if that row's index is not in the selectable set or, otherwise, it is given by $p_i / \sum_{j\in\mathcal{S}_k} p_j$. If the selectable set is small, it is computationally advantageous to explicitly change the sampling distribution in each iteration. On the other hand, if the selectable set contains many rows, a rejection sampling approach is better since it avoids constantly recomputing the distribution. The last step of the algorithm can be done in two ways, meaning that there are two variations of the SSRK method: the Non-Repetitive Selectable Set Randomized Kaczmarz (NSSRK) method and the Gramian Selectable Set Randomized Kaczmarz (GSSRK) method.

For the NSSRK method, the selectable set is updated in each iteration by simply using all rows except the row that was sampled in the previous iteration. In that case, the last step of the algorithm can be written as $\mathcal{S}_{k+1} = [m] \backslash i_k$.

The GSSRK method makes use of the Gramian of matrix A, that is $G = AA^T$, where each entry can be written as $G_{ij} = \langle A^{(i)}A^{(j)}\rangle$. It is known that if an equation $A^{(j)}x = b^{(j)}$ is solved by iterate $x^{(k)}$ and if $A^{(i_k)}$ is orthogonal to $A^{(j)}$, that is, if $G_{i_k j} = 0$, then the equation is also solved by the next iterate $x^{(k+1)}$. This means that if $j \notin \mathcal{S}_k$ and if $G_{i_k j} = 0$, then equation $A^{(j)}x = b^{(j)}$ is still solvable by the next iterate $x^{(k+1)}$ and the index j should remain unselectable for one more iteration. However, indices that satisfy $G_{i_k j} \neq 0$ should be reintroduced in the selectable set in each iteration. In summary, the last step of the SSRK algorithm can be written as $S_{k+1} = (S_k \cup \{j : G_{i_k j} \neq 0\})\backslash\{i_k\}$.

In terms of performance, the authors observed that: NSSRK and RK are almost identical; GRK outperforms GSSRK, NSSRK and RK; depending on the data sets, GSSRK can either outperform NSSRK and RK or it can be similar to RK. It is important to note that these conclusions were only made in terms of how fast the error decreases as a function of the number of iterations. There is no analysis of the performance of the algorithms in terms of time.

3.5 Randomized Kaczmarz with Averaging Method

The Randomized Kaczmarz method is difficult to parallelize since it uses sequential updates. Furthermore, just as was mentioned before, RK does not converge to the least-squares solutions when dealing with inconsistent systems. To overcome these obstacles, the Randomized Kaczmarz with Averaging (RKA) method [23] was introduced by Moorman, Tu, Molitor, and Needell. It is a block-parallel method that, in each iteration, computes multiple updates that are then gathered and averaged. Let q be the number of threads and τ_k be the set of q rows randomly sampled in each iteration. In that case, the Kaczmarz step can be written as

$$x^{(k+1)} = x^{(k)} + \frac{1}{q}\sum_{i\in\tau_k} w_i \frac{b_i - \langle A^{(i)}, x^{(k)}\rangle}{\|A^{(i)}\|_2^2} A^{(i)T} \tag{3.12}$$

where w_i are the row weights. The projections corresponding to each row in set τ_k should be computed in parallel. The authors of this method have shown that not only has RKA linear convergence like RK but that it is also possible to decrease the convergence horizon for inconsistent systems if more than one thread is used. More specifically, they showed that using the same probability distribution as in RK for row sampling, that is, proportional to the row's squared norm, and using uniform weights $w_i = \alpha$, the convergence can be accelerated such that the error in each iteration satisfies

$$\mathbb{E}\,\|x^* - x^{(k)}\|^2 \leq \sigma_{max}\Big(\Big(I - \alpha\frac{A^TA}{\|A\|_F^2}\Big)^2 + \frac{\alpha^2}{q}\Big(I - \frac{A^TA}{\|A\|_F^2}\Big)\frac{A^TA}{\|A\|_F^2}\Big)\|x^* - x^{(0)}\|^2 + \frac{\alpha^2}{q}\frac{\|r_{LS}\|^2}{\|A\|_F^2}. \tag{3.13}$$

Notice that larger values of q lead to faster convergence and a smaller convergence horizon. The authors also give some insight into how α can be chosen to optimize convergence. In the special case of uniform weights and consistent systems, the optimal value for α is given by:

$$\alpha^* = \begin{cases} \dfrac{q}{1+(q-1)s_{min}}, & s_{max} - s_{min} \leq \dfrac{1}{q-1} \\ \dfrac{2q}{1+(q-1)(s_{min}+s_{max})}, & s_{max} - s_{min} > \dfrac{1}{q-1} \end{cases} \tag{3.14}$$

where $s_{min} = \sigma^2_{min}(A)/\|A\|^2_F$ and $s_{max} = \sigma^2_{max}(A)/\|A\|^2_F$. Although the authors proved that, if the computation of the q projections can be parallelized, RKA can have a faster convergence rate than RK, they did not implement the algorithm using shared or distributed memory, meaning that no results regarding speedups are presented.

3.6 Parallel Implementations of the Kaczmarz Method

There are two main approaches to parallelizing iterative algorithms, both of which divide the equations into blocks. The first mode, called block-sequential or block-iterative, processes the blocks in a sequential fashion, while computations on each block are done in parallel. In the second mode of operation, the block-parallel, the blocks are distributed among the processors and the computations in each block are simultaneous; the results obtained from the blocks are then combined to be used in the following iteration.

3.6.1 Component-Averaged Row Projections

The Component-Averaged Row Projections (CARP) [24] was introduced to solve partial differential equations (PDEs). Previous methods used a block-sequential approach and are not adequate to solve PDEs: some methods require computation and storage of data related to the inverses of submatrices; others are very robust but do not have a consistent way to partition the matrix into submatrices. CARP is a block-parallel method of the Kaczmarz method where the blocks of equations are assigned to processors and the results are then merged to create the next iteration. CARP is more memory efficient than previous block-based projection methods since the only memory requirement for each processor is the submatrix of the local equations coefficients and respective entries of the b vector and solution vector. In terms of parallel behavior, CARP was developed in both shared and distributed memory and it exhibits an almost linear speedup ratio. Although CARP is shown to converge for consistent systems, no linear convergence rate is given. It is also important to note that the optimal performance of CARP depends on the choice of the relaxation parameter and the number of inner iterations in each block, which are not known.

3.6.2 Asynchronous Parallel Randomized Kaczmarz Algorithm

Liu, Wright, and Sridhar [25] developed the Asynchronous Parallel Randomized Kaczmarz (AsyRK) algorithm for shared memory systems. The asynchronous parallel technique used in AsyRK was developed by Niu et al [26] for HOGWILD!, a parallelization scheme for SGD (see Appendix B) developed without locking. HOGWILD! was built specifically for sparse problems to minimize the time spent in synchronization. Since SGD is an iterative algorithm, similar to CARP or RKA, we can calculate several iterations in parallel and then update the solution vector x. Normally we would think that x should be locked because, otherwise, the processors would override each other's updates. But, if the data is sparse, memory overwrites are minimal since each processor only modifies a small part of x. This method reduces significantly the overhead of synchronization; even if memory overwrites do occur, the error introduced by them is very small and has minimal impact. There is a small restriction regarding the update of x: there is a maximum number of updates, ρ, that can occur between the time at which any processor reads the current x and the time at which it makes its update. The asynchronous parallel variant of the Stochastic Gradient Descent method exhibits near-linear speedup with the number of processors when using sparse data. The application of this no-lock technique to the asynchronous parallelization of the Stochastic Coordinate Descent (SCD) method [27], called AsySCD [28], has also shown improvements in the rate of convergence when compared to sequential versions. Note that, just like RK is a special case of the SGD method (see Appendix B), RGS (see Appendix B) is also a special case of the SCD method.

The AsyRK algorithm was built by applying the HOGWILD! technique to RK. Liu, Wright, and Sridhar show that the AsyRK method can outperform AsySCD in running time by an order of magnitude. The algorithm is now described. Let us consider that there is an iteration counter k that is incremented each time x is updated by a thread; $j(k)$ is the iterate at which x was read by the thread that updated $x^{(k)}$ to $x^{(k+1)}$ (these are only the same if x is read and updated and there are no other threads making changes to x in that time frame). AsyRK includes the following steps:

- The matrix A is partitioned into blocks of equal size, such that a subset of equations is assigned to each thread;
- Each thread randomly orders its rows such that the rows in each iteration are sampled without replacement [1]. After a full scan of a thread's submatrix, the thread reshuffles the rows.
- Making the assumption that the rows of matrix A are normalized, for each sampled row in each thread, the only computation needed to update x is that of $\Delta x = {A^{(i)}}^T(b_i - \langle A^{(i)}, x^{j(k)}\rangle)$. Note that since A is sparse, Δx is a vector of mainly zeros. Finally, the non-zero entries of Δx are summed to the corresponding entries of $x^{(k)}$ to update the solution to iteration $x^{(k+1)}$.

[1] When developing Randomized Block Kaczmarz (RBK) C.1, Needell and Tropp also found that sampling without replacement is more effective.

Although AsyRK has good results when compared to AsySCD, it is important to remember that this method was developed for sparse matrices only and that linear speedup is only observed if the number of processors, ρ, fulfills the requirement $(2e\lambda_{max}(\rho+1))/m \leq 1$, where λ_{max} is the maximum eigenvalue of $A^T A$.

3.7 Summary

In this section, we summarize all the variations of the original Kaczmarz method analyzed so far in Table 3.1.

Table 3.1: Summary of the variations of the Kaczmarz method. In the case of underdetermined consistent systems, x_{LN} refers to the least Euclidean norm solution. For overdetermined systems, x^* refers to the unique solution of consistent systems and x_{LS} refers to the least-squares solution of inconsistent systems.

Name	Abbreviation	Year	Convergence		
			x_{LN}	x^*	x_{LS}
Cyclic Kaczmarz 2.3	CK	1937	$\surd$	$\surd$	$\times$
Randomized Kaczmarz 2.4	RK	2007	$\surd$	$\surd$	$\times$
Randomized Gauss-Seidel 3.1	RGS	2009	$\times$	$\surd$	$\surd$
Randomized Extended Kaczmarz 3.2	REK	2013	$\surd$	$\surd$	$\surd$
Greedy Randomized Kaczmarz 3.3	GRK	2018	$\surd$	$\surd$	$\times$
Non-Repetitive Selectable Set Randomized Kaczmarz 3.4	NSSRK	2021	$\surd$	$\surd$	$\times$
Gramian Selectable Set Randomized Kaczmarz 3.4	GSSRK	2021	$\surd$	$\surd$	$\times$
Randomized Kaczmarz with averaging 3.5	RKA	2021	$\surd$	$\surd$	$\times$
Component-Averaged Row Projections 3.6.1	CARP	2005	$\surd$	$\surd$	$\times$
Asynchronous Parallel Randomized Kaczmarz 3.6.2	AsyRK	2014	$\surd$	$\surd$	$\times$

The first seven rows of Table 3.1 correspond to sequential methods. In our simulation, we will use overdetermined systems, which are more common in real-world problems than underdetermined systems. We will compare the RGS and REK methods for inconsistent systems, since they are the only ones able to solve them, and the remaining sequential methods will be tested for consistent systems.

Chapter 4

Sequential Version

In this chapter, we evaluate the performance of sequential implementations of the Kaczmarz method and some of its variants. The chapter is organized as follows. Section 4.1 presents the implementation details for the considered variants of the Kaczmarz method, as well as a description of the experimental setup. Section 4.2 discusses the obtained experimental results.

Simulations were implemented in the C++ programming language; its source code and corresponding documentation are publicly available [1]. All experiments were carried out on the Accelerates Cluster [2]. This cluster has 1600 cores distributed over 80 nodes, each of them with two 2.8 GHz central processing units (Intel Xeon E5-2680 v2 CPU) with 32 GB memory.

4.1 Implementation

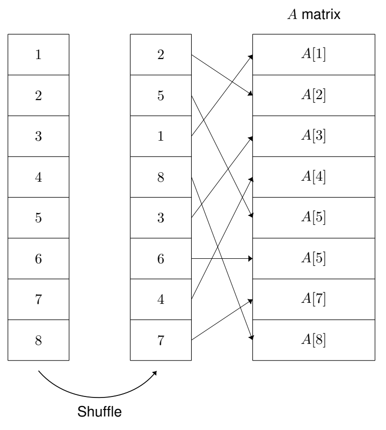
The following sequential methods were implemented: Cyclic Kaczmarz method (CK - Section 2.3); Randomized Kaczmarz method (RK - Section 2.4); Simple Randomized Kaczmarz method (SRK - Section 2.5); Randomized Gauss-Seidel method (RGS - Section 3.1); Randomized Extended Kaczmarz method (REK - Section 3.2); Greedy Randomized Kaczmarz method (GRK - Section 3.3); Gramian Selectable Set Randomized Kaczmarz method (GSSRK - Section 3.4); and Non-Repetitive Selectable Set Randomized Kaczmarz method (NSSRK - Section 3.4).

Furthermore, extending on the findings of Needell and Tropp presented in Section C.1 that sampling without replacement is an effective row selection criterion, we also developed the Simple Randomized Kaczmarz Without Replacement (SRKWOR) method. In sampling without replacement in the context of row selection, if a row is selected, it is necessary that all other rows of matrix A are also selected before this row can be selected again. A straightforward technique for sampling rows without replacement is to randomly shuffle the rows of matrix A and then iterate through them in a cyclical manner. We consider two variants of this method. In the first, which we called SRKWOR without shuffling, the matrix is shuffled only once during preprocessing. In the second method, which we called SRKWOR with shuffling, the matrix is shuffled after each cyclical pass. The first variant trades off less randomization for a smaller shuffling overhead than the second variant.

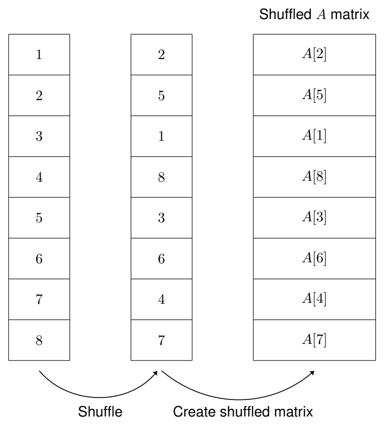
Regarding the process of shuffling the rows of the matrix, we consider two possible approaches: the first option is to work with a shuffled array of row indices, a technique we call two-fold indexing

[1]Code available here: https://github.com/inesalfe/Thesis-Kaczmarz.git
[2]http://epp.tecnico.ulisboa.pt/accelerates/

(Figure 4.1a); and the second option is to create a new matrix with the desired order (Figure 4.1b). In the first option, the memory usage is diminutive but the access time to a given row can be more significant since we first have to access the array of indices and then the respective matrix row. In the second option, we access the matrix directly, while also taking advantage of spatial cache locality: the fact that rows are stored sequentially in memory promotes more cache hits than in the first case. Both options were implemented and compared.



(a) Two-fold indexing.



(b) Shuffled matrix.

Figure 4.1: Example of how to reorder rows of a matrix with 8 rows.

To motivate the parallelization of the Kaczmarz method, it is worth showing that it can outperform the celebrated Conjugate Gradient method. Therefore, other than the Kaczmarz method and its variants, we will also use the CG [3] and Conjugate Gradient for Least-Squares (CGLS) [4] methods from the EIGEN linear algebra library [5]. For CG [6] we used the default preconditioner (diagonal preconditioner) and both the upper and lower triangular matrices (the parameter *UpLo* was set to *Lower—Upper*). For CGLS [7] we used also used the default preconditioner (least-squares diagonal preconditioner). Since the CG method can only be used for squared systems, it was necessary to transform the system in (2.1) into a squared system by multiplying both sides of the equation by A^T, such that the new system is given by $A^TAx = A^Tb$. The computation time of A^TA and A^Tb was added to the total execution time of CG.

4.1.1 Stopping Criterion

Since the Kaczmarz method and its variants are iterative methods, it is required to define a stopping criterion. For the implemented methods that solve consistent systems, the chosen stopping criterion is made up of two conditions:

[3] https://eigen.tuxfamily.org/dox/classEigen_1_1ConjugateGradient.html
[4] https://eigen.tuxfamily.org/dox/classEigen_1_1LeastSquaresConjugateGradient.html
[5] https://eigen.tuxfamily.org/index.php?title=Main_Page
[6] https://eigen.tuxfamily.org/dox/classEigen_1_1DiagonalPreconditioner.html
[7] https://eigen.tuxfamily.org/dox/classEigen_1_1LeastSquareDiagonalPreconditioner.html

- Condition 1 - The squared norm of the difference between the current and previous iterations must not surpass a certain threshold ε_1, that is, $\|x^{(k)} - x^{(k-1)}\|^2 < \varepsilon_1$. This means that we have reached a point where barely any changes are being made to the estimate of the solution. This condition by itself is not enough since, in the randomized versions, there is a chance that the same row is chosen in two consecutive iterations, meaning that the estimate of the solution is not going to change between those iterations and thus $\|x^{(k)} - x^{(k-1)}\|^2 = 0$.
- Condition 2 - If the first condition holds, a second test is made to confirm that the solution has been found. Since the residual $r = b - Ax^{(k)}$ should be zero for the solution, we check if the squared norm of the residual is also inferior to a certain threshold ε_2, that is, $\|r\|^2 < \varepsilon_2$. This parameter describes how accurate the solution is.

Note that both conditions need additional computations apart from those required in each iteration. Furthermore, the computation of the residual in the second condition is a very computationally expensive task, since it requires computing $Ax^{(k)}$. To make the stopping criteria as efficient as possible two modifications were introduced. Firstly, since the number of iterations for the Kaczmarz method is usually very high, especially for large systems, the first condition of the stopping criteria is only verified every $step$ iterations. Second, to reduce the number of times that the residual is calculated, the first condition was defined to be more strict that the second one.

The desired accuracy of the solution is defined to be $\varepsilon_2 = 10^{-10}$. Given this parameter, we need to find an optimal value for ε_1. If ε_1 is very small, then we may be needlessly computing extra iterations while sooner iterations could already satisfy the second condition. If ε_1 is very large, a lot of time will be spent computing the residual. So, given ε_2, we tested the Kaczmarz method for $\varepsilon_1 = \{10^{-15}, 10^{-20}, 10^{-20}, 10^{-25}\}$ and $step = \{5, 10, 20, 100, 1000, 10000\}$. Since the goal is to apply the Kaczmarz method to large systems, we chose the parameters that worked best for them, $\varepsilon_1 = 10^{-25}$ and $step = 1000$.

CG and CGLS have a different stopping criterion than RK, based on a maximum number of iterations and/or an upper bound for the relative residual error ($\|Ax^{(k)} - b\|/\|b\|$). To ensure a fair comparison between the EIGEN methods and RK we must make a few changes. First, we determine the number of iterations that the different methods take to achieve a given error. Then we use those numbers as the maximum number of iterations and measure the execution time of the methods. With this procedure not only do we guarantee that the solutions given by the methods have similar errors but we also only measure the time spent computing iterations, without taking into account the stopping criteria.

Computing the number of iterations needed to achieve a solution that satisfies a given error is simple for RK since the implemented method already computes the iterations necessary to achieve a solution with a given residual upper bound. We only need to create a simpler version of the method that uses a given number of iterations as a stopping criterion to measure the execution time of the method without the computations related to the stopping criteria. For EIGEN methods, the process of computing the maximum number of iterations is slightly more complicated and is the following: we start by setting the maximum number of iterations to 1. We then compute the solution and the norm of the residual - if it falls below the upper bound defined for RK we have found the number of iterations. Otherwise, we keep increasing the maximum number of iterations until we find one that satisfies the previous condition. Although this process might be seen as extremely slow note that methods such as CG and CGLS have typically a much smaller number of iterations when compared with row or column action methods like the Kaczmarz method.

The methods that solve least-squares problems (REK and RGS) also require different stopping criteria: for overdetermined inconsistent systems the residual is never zero and the second condition cannot be used. Since we will also want to compare REK and RGS with CGLS we followed a similar procedure to the one previously described for comparing RK with CG and CGLS. In summary, we compute the number of iterations for RK, CG, and CGLS to reach the error upper bound $\|x^{(k)} - x_{LS}\|^2 < \epsilon_3$ and then measure the execution time of those iterations. ϵ_3 was chosen to be 10^{-10}

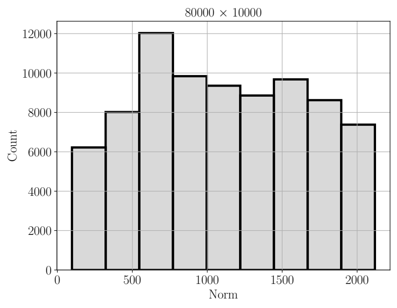
4.1.2 Data Structures for Matrix Storage

The linear systems that arise in many applications are represented by sparse matrices. A sparse matrix is a matrix where the vast majority of elements are zero. By contrast, if most elements in the matrix are non-zero, the matrix is said to be dense. In computer programs, matrices are typically stored as a two-dimensional array. This format is not appropriate for sparse matrices since we would be ”wasting” a lot of memory storing zeros. A better approach is to only store the non-zero elements of a sparse matrix. We call the number of non-zero elements NNZ. Among the efficient formats, there are two formats that are efficient in both access and matrix operations: the Compressed Sparse Row (CSR) and the Compressed Sparse Column (CSC) formats. The CSR format uses three arrays to store non-zero elements: the first contains all the values of the non-zero elements and, by consequence, has a size equal to NNZ; the second has the column indices of those elements and also has size NNZ; the last vector has size $m + 1$ and stores the cumulative number of non-zero elements up to, but not including, the i-th row. In the first and second arrays the values appear as if we were traversing the matrix row-by-row. The CSC format is very similar to the previous one but instead of traversing the matrix row-by-row, we do it column-by-column. The first array is the same, the second array has row indices, and the last array stores the cumulative number of non-zero elements by column. Since the matrices can be either sparse or dense, two versions of each variation of the algorithm were developed: one for dense matrices, using a two-dimensional array to store matrix A; and another for sparse matrices, using the CSR, CSC or both CSR and CSC formats, depending on whether the variation in question is a row action or column action method (or both). The Cyclic Kaczmarz method and the variants where different row selection criteria are used (RK, GRK, GSSRK and NSSRK) are all row action methods, thus the versions for sparse matrices were developed using the CSR format. Since the RGS method is a column action method, the CSC format was used for the sparse matrices version. Finally, the REK method is both a row and column action method, one column is used in the first step of the algorithm and then one row is used in the second step; For this reason, there was the need to use both the CSR and the CSC format, hinting that the memory usage for this algorithm is very inefficient when compared to other methods.

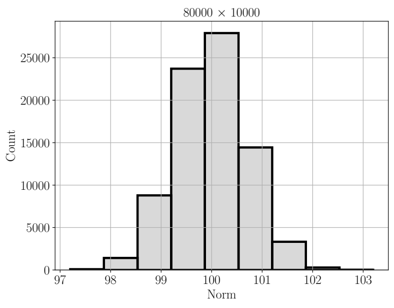
4.1.3 Data Sets

The Kaczmarz method and its variants were tested for dense and sparse matrices using artificial data sets generated using C++. For dense systems, three main data sets were generated: one with contrasting row norms, a second one with similar row norms, and a third one with coherent rows. The goal of the first two was to evaluate how different row selection criteria perform against matrices with different row norms distributions. The goal of the third dense data set was to compare the randomized versions with the original version of the Kaczmarz method for a system similar to the one represented in Figure 2.3. The purpose of also generating sparse systems is to determine the impact of using a two-dimensional array versus the CSR / CSC format to store sparse matrices and they were generated also with contrasting row norms.

In the first dense data set matrix entries were sampled from normal distributions where the average μ and standard deviation σ were obtained randomly. For every row, μ is a random number between -5 and 5 and σ is a random number between 1 and 20. For dense matrices, the matrix with the largest dimension was generated and smaller-dimension matrices were obtained by ”cropping” the largest matrix. This keeps some similarities between matrices of different dimensions for comparison purposes. Since the focus of these methods is the resolution of overdetermined systems, row sizes of matrices can be, 2000, 4000, 20000, 40000, or 80000, and column sizes vary between 50, 100, 200, 500, 750, 1000, 2000, 4000 or 10000. To guarantee a unique solution for consistent systems the solution vector x is sampled from a normal distribution with μ and σ using the same procedure as before, and vector b is calculated as the product of A and x. The second data set was generated with the same procedure except that the parameters of the normal distribution are the same for all rows $\mu = 0$ and $\sigma = 1$. To verify that row norms are indeed contrasting for the first data set and similar for the second one, Figure 4.2 shows a histogram of row norms for the largest matrix of the first and second data sets. The goal of the third dense data set is to have coherent rows. This can be achieved by having consecutive rows with few changes between them. Just like in the previous two data sets, we compute the largest matrix and solution and the smaller systems can be constructed by cropping from the biggest system. The generation of the largest matrix was completed using the following steps: we start by generating the entries of the first row by sampling



(a) First data set generated using $\mu \in [-5, 5]$ and $\sigma \in [1, 20]$.



(b) First data set generated using $\mu = 0$ and $\sigma = 1$.

Figure 4.2: Histogram of row norms for the largest matrix of the first two data sets.

from a normal distribution $N(2, 20)$ - these parameters ensure that the entries are not too similar; we then define the next row as a copy of the previous one, randomly select five different columns, and change the corresponding entries onto new samples of $N(2, 20)$; this procedure is repeated until the entire matrix is computed. In summary, we have a random matrix where every two consecutive rows only differ in 5 elements.

Regarding the sparse data set, we generated systems with a single dimension, 80000×1000, but different densities. We used a similar procedure to generate the matrix of each system as the one used for the first dense data set. The difference is that only a limited number of entries for each row is sampled using normal distributions since most entries should be set to 0. The number of non-zero entries in each row was set to be 1, 2, 5, 8, 10, 20, 30, and 50, meaning that 8 systems were created with matrices with densities ranging from 0.1% to 5%.

Finally, a data set with inconsistent systems was generated to test the methods that solve least-squares problems (REK and RGS). This was accomplished by adding an error term to the consistent systems from the first dense data set. Let b and b_{LS} be the vectors of constants of the consistent and inconsistent systems. The lesser was defined such that $b_{LS} = b + N(0, 1)$. The least-squares solution, x_{LS}, was obtained using the CGLS method.

4.1.4 Effect of Randomization

Since the row selection criterion has a random component in most variants of the algorithm, the solution vector x, the maximum number of iterations, and the running time varies with the chosen seed for the random number generator. To get a robust estimate of the number of iterations and execution times, for each input, the algorithm is run several times with different seeds, and the solution x is calculated as the average of the outputs from those runs. In every run, the initial estimate of the solution, $x^{(0)}$, was set to 0. Execution times for the dense data sets correspond to the total of 10 runs of the algorithm with $step = 1000$. For the sparse data, since the execution times are small, 100 runs were used instead of 10.

In the RK, SRK, GRK, GSSRK, NSSRK, REK, and RGS methods, in each iteration, a row and/or column is used, and these are sampled from a discrete probability distribution with probabilities that, for example, for the RK method, depend on the norms of the rows of the matrix A. In the implementation of these methods, sampling numbers that follow a discrete probability distribution was accomplished using the class DISCRETE_DISTRIBUTION from the STD library.

4.2 Results

4.2.1 Randomized Kaczmarz Algorithm

The simulations for the RK method in this section used the first dense data set. We start with the analysis of the method's convergence (described by (2.10)), shown in Figure 4.3. Note that the y-axis scale is logarithmic, meaning that the error decreases exponentially with the number of iterations, proving that the method exhibits linear convergence.

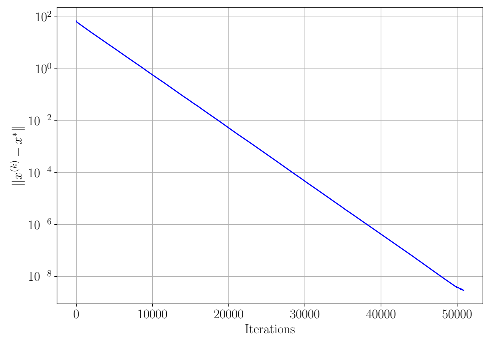
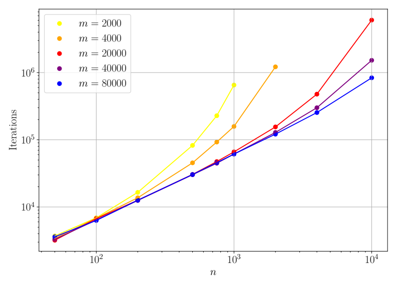
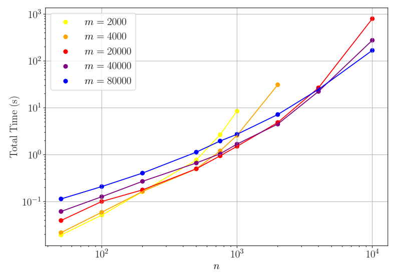


Figure 4.3: Error as a function of the number of iterations for an 80000×1000 dense overdetermined system for the Randomized Kaczmarz method. The error is given by the difference between the estimate of the solution at the current iteration and the solution of the system.

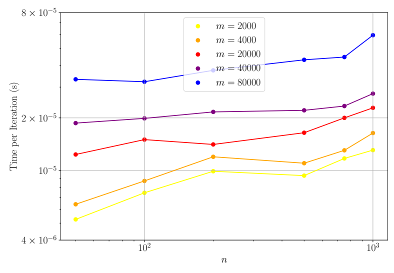
The following analysis focuses on the number of iterations and execution time as functions of the number of rows, m, and columns, n. From Figure 4.4a it is clear that the number of iterations increases with n. Increasing n while maintaining m makes systems harder to solve since there are more variables for the same number of restrictions. However, for a given value of n, there is not a clear correlation between execution time and the number of rows. This is due to the connection between rows and information: for overdetermined systems, more rows translate into more information to solve the system. Figure 4.4b shows that the total computation time also increases with n due to the correlation with the number of iterations. However, for a given value of n, systems with a larger number of rows may take more or less time than systems with a smaller number of rows. To better understand the dependency of the total time with n, the time per iteration was computed and is shown in Figure 4.4c. We can see that the time per iteration increases with n and that iterations for systems with larger m take longer to compute. This is expected since the work per iteration depends on n and since the stopping criteria require computing the norm of the residual, which depends on m. For smaller n, since the number of



(a) Number of iterations.



(b) Total execution time.

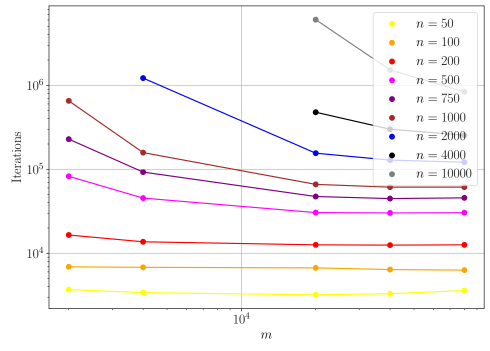


(c) Execution time per iteration.

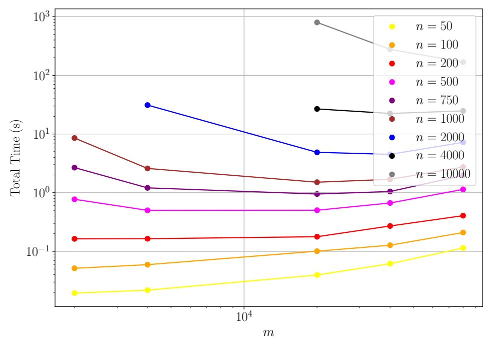
Figure 4.4: Results for the Randomized Kaczmarz algorithm using a fixed number of rows and a varying number of columns.

iterations is similar for all values of m, and iterations for larger m take longer to solve, the total time is larger for larger m. When n increases, the number of iterations for smaller m increases in a larger proportion than the time per iteration for larger m. The same analysis that was previously made for systems with fixed row numbers is now repeated for systems with fixed column numbers. In terms of iterations, as can be seen in Figure 4.5a, these decrease until they reach a stable value. This is again due to the connection between a number of rows and information. Increasing the number of rows adds restrictions to the system, making it easier to solve and lowering the number of iterations. However, there is a point where the number of iterations needed to solve the system is less than the number of rows of the system, after which an increase in m is no longer useful. The initial decrease in iterations is accompanied by a decrease in time, as seen in Figure 4.5b. Then, time starts to increase since the individual iterations for larger m require a higher computational effort.

Figure 4.6 presents the impact of the stopping criteria for a few systems with larger values of m and n. Figure 4.6a shows the percentage of time spent computing the stopping criteria. As expected, for a fixed n, the time computing the stopping criteria is greater for the system with a larger number of rows, since the time spent computing the norm of the residual is proportional to m. To explain why the percentage decreases with n, we fitted function $y = \beta_1 x^{\beta_2}$ to the time per iteration with and without the

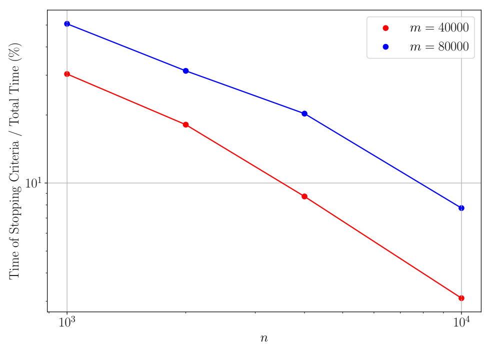


(a) Number of iterations.

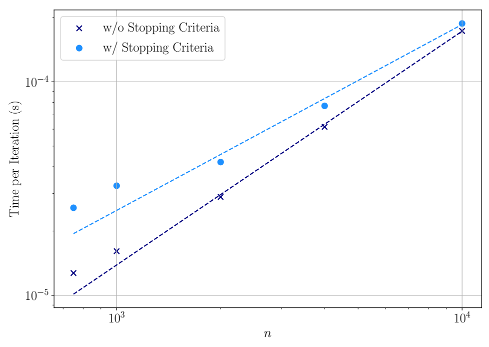


(b) Execution time.

Figure 4.5: Results for the Randomized Kaczmarz algorithm for dense systems using a fixed number of columns and a varying number of rows.



(a) Percentage of time spent computing the stopping criteria.



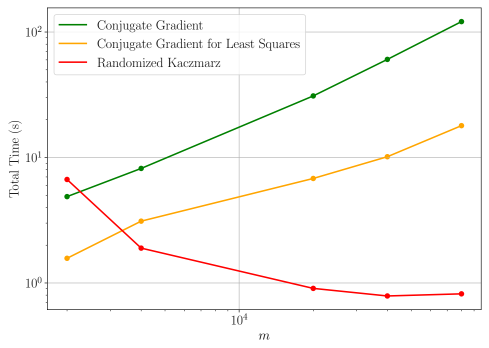
(b) Time per iteration for systems with $m = 80000$. The dashed lines were obtained with the fit function $y = \beta_1 x^{\beta_2}$.

Figure 4.6: Efficacy of the stopping criteria for systems with a fixed number of rows and a varying number of columns.

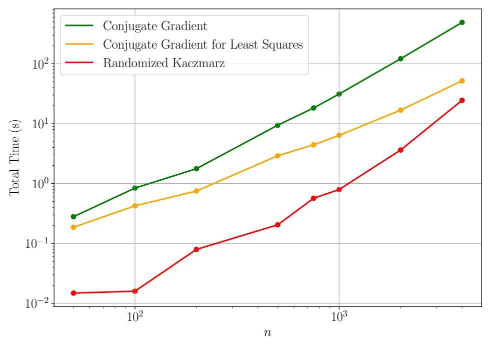
stopping criteria for $m = 80000$, shown in Figure 4.6b. The parameters of the fit, without the stopping criteria, were $\beta_1 = 7.20 \times 10^{-9}$ and $\beta_2 = 1.09$, and, with the stopping criteria, $\beta_1 = 6.10 \times 10^{-8}$ and $\beta_2 = 8.71 \times 10^{-1}$. [8] The values of the parameters β_2 show that time without the stopping criteria is slightly superlinear with n and the time with the stopping criteria is slightly sublinear with n. Since the number of iterations increases with n, and so does the time per iteration, the total execution time will also increase with n. Therefore, the quotient of time spent computing the stopping criteria and the total execution time decreases with n, as seen in Figure 4.6a.

To finish the analysis of RK for dense systems, we show how this method can outperform CG and CGLS. From both Figures 4.7a and 4.7b we can conclude that, regardless of matrix dimension, CGLS is a faster method than CG for solving overdetermined problems. This is only normal since CGLS is an extension of CG to non-square systems and the execution time of CG takes into account computing $A^T A$. From Figure 4.7a we can see that, except for the smallest system, RK is faster than CGLS. Figure 4.7b shows that, although RK is faster for the matrix dimensions shown in the plot, the difference

[8] These results are presented with two significant figures.



(a) Computational time until convergence for systems with $n = 1000$.



(b) Computational time until convergence for with $m = 20000$.

Figure 4.7: Comparison between RK, CG and CGLS for dense overdetermined systems.

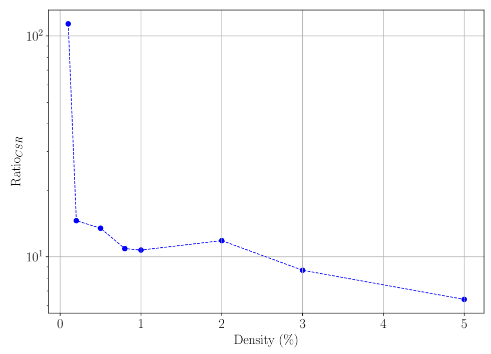


Figure 4.8: Speedup as a function of matrix density for sparse matrices with dimensions 80000×1000.

between this method and CGLS decreases when n increases, that is, when m and n get closer. In summary, RK should be used to the detriment of CG and/or CGLS for very overdetermined systems. However, it is important to note that, for the CG method, a very large portion of the execution time is used to transform the system in (2.1) into a squared system. In fact, if we were to analyse the time spent only in iterations of the CG method, this would be faster than RK. Nonetheless, we are working with overdetermined systems and the computation time of $A^T A$ and $A^T b$ cannot be ignored.

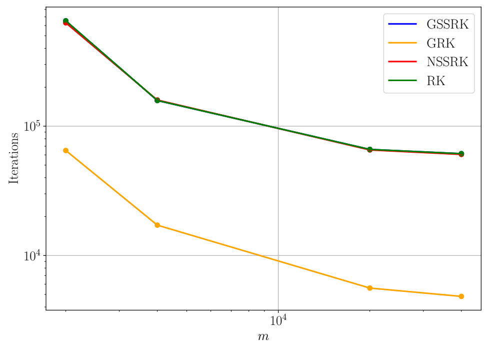
So far we have shown the results of the simulations using the first data set. The previous plots in this section were reproduced using the second dense data set instead of the first one, but, since the conclusions are similar, we decided not to show them here. Finally, we intend to show how the use of the CSR format for sparse matrices decreases the runtime of the algorithm. We applied the algorithm for sparse matrices with dimensions 80000×1000 with different densities and used both a two-dimensional array and the CSR format to store matrix A. We then computed the speedup defined by

$$\text{Ratio}_{CSR} = \frac{\text{CPU time (two-dimensional array)}}{\text{CPU time (CSR format)}}. \tag{4.1}$$

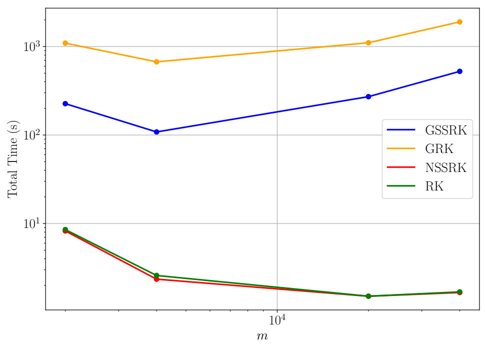
From Figure 4.8 it is clear that the difference between the two formats is larger for smaller densities. In fact, for a density of 0.1% (only one nonzero entry per row), using the CSR format results in computational times 100 times smaller than using a two-dimensional array, while for a density of 5% (fifty nonzero entries per row), using the CSR format is 6 times faster. These results are according to intuition, as the number of multiplications necessary to compute a dot product between two vectors is proportional to matrix density. The main result here is that using the CSR format for sparse matrices is significantly faster than using a two-dimensional array.

4.2.2 Variants of the Kaczmarz Algorithm for Consistent Systems

We now compare several variants of the Kaczmarz method between themselves. In this section, we will use analyze the results for the first dense data set using a fixed column number. We start with



(a) Number of iterations. Note that the iterations for GSSRK, NSSRK, and RK and overlapped.

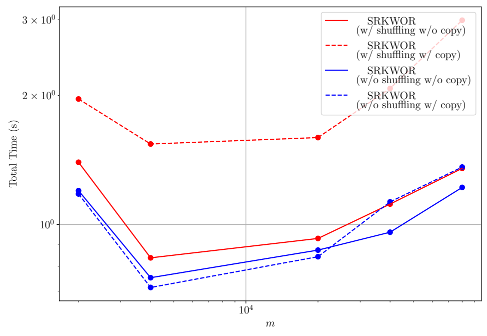


(b) Execution time.

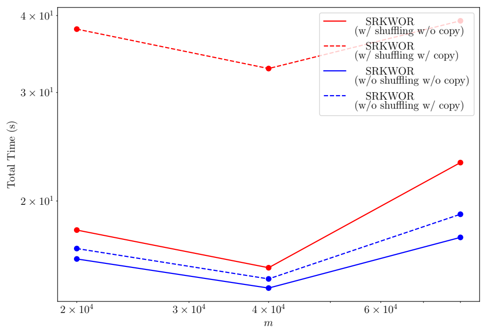
Figure 4.9: Results for some variants of the Kaczmarz algorithm for dense systems using a fixed number of columns $n = 1000$ and a varying number of rows.

the methods introduced in Chapter 3 that, like RK, select rows based on their norms. Figure 4.9a contains the evolution of the number of iterations while the right plot contains the total execution time until convergence. From the number of iterations we can conclude that the GRK method has the most efficient row selection criterion. However, this does not translate into a lower execution time, as observed in Figure 4.9a, meaning that each individual iteration is more computationally expensive than individual iterations of the other methods. This is due to two factors: first, there is the need to calculate the residual in each iteration; second, since rows are chosen using a probability distribution that relies on the residual, and since the residual changes in each iteration, there is the need to update, in each iteration, the discrete probability distribution that is used to sample a single row. For other methods the residual is only computed to check the second condition of the stopping criterion and the discrete probability distribution used for row selection depends only on matrix A, and therefore, is the same for all iterations of the algorithm. The RK, GSSRK, and NSSRK methods have an indistinguishable number of iterations. In terms of time (Figure 4.9b), NSSRK and RK have similar performance while GSSRK is a slower method. This happens since, for dense matrices, it is very rare that rows of matrix A are orthogonal and the selectable set usually corresponds to all the rows of the matrix - this means that a lot of time is spent checking for orthogonal rows and very few updates are made to the selectable set.

Before we move on to the analysis of the other Kaczmarz-based methods with different row selection criteria, we would like to determine, for the SRKWOR, whether we should use the option with shuffling or without shuffling. Furthermore, it is also useful to compare the performance between two-fold indexing and creating a reordered copy of matrix A. These four variants of SRKWOR are presented in Figure 4.10. Figures 4.10a and 4.10b show that, for most dimensions, the option without shuffling is faster than the option with shuffling. For SRKWOR with shuffling, using two-fold indexing is better than creating a new reordered matrix. The behavior of SRKWOR without shuffling is different since there appears to be little difference between the execution times of the two methods. In summary, we conclude that SRKWOR



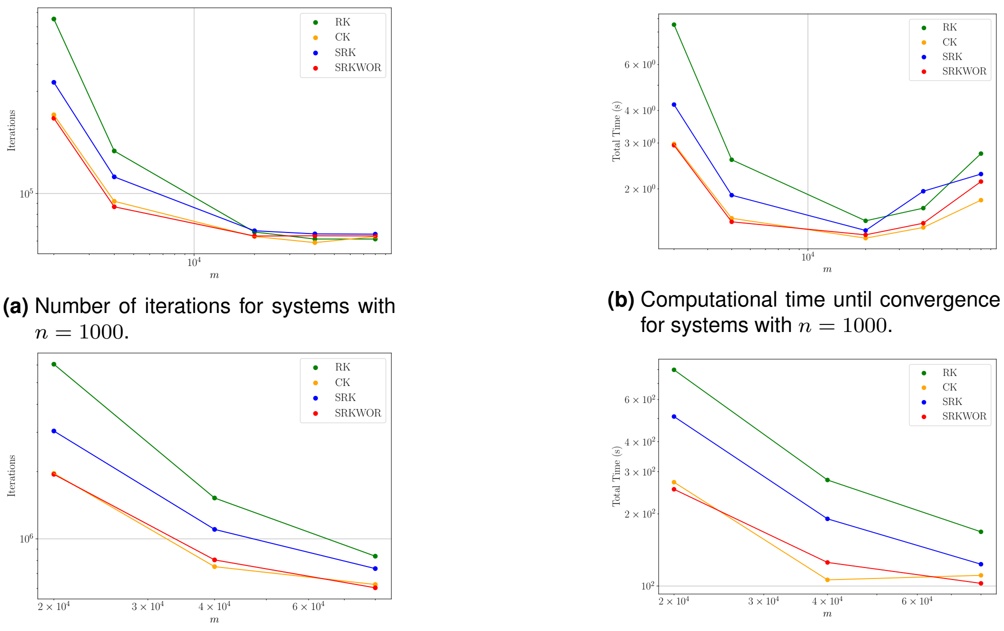
(a) Computational time until convergence for several overdetermined systems with $n = 750$.



(b) Computational time until convergence for several overdetermined systems with $n = 4000$.

Figure 4.10: Results for SRKWOR for dense systems using a fixed number of columns and a varying number of rows. The dashed lines, referred to as "w/ copy" refer to the option that uses a reordered matrix while "w/o copy" refers to two-fold indexing.

is more efficient without shuffling and using two-fold indexing. From this point on, references to the SRKWOR method correspond to the version without shuffling and with two-fold indexing.



(a) Number of iterations for systems with $n = 1000$.

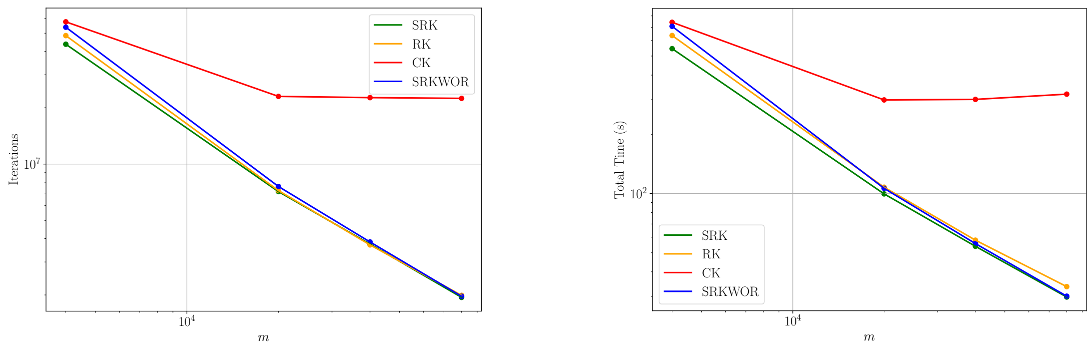
(b) Computational time until convergence for systems with $n = 1000$.

(c) Number of iterations for systems with $n = 10000$.

(d) Computational time until convergence for systems with $n = 10000$.

Figure 4.11: Results for some variants of the Kaczmarz algorithm for dense systems using a fixed number of columns and a varying number of rows.

When Strohmer and Vershynin in [10] introduced the RK method, they compared its performance to the CK and the SRK methods. Here, we make the same analysis with the addition of SRKWOR. The results for the number of iterations and computational time are presented in Figure 4.11. The first observation to be made is that CK yields very similar results to SRKWOR, which is expected when working with random matrices. The SRK method, for smaller values of m, exhibits a lower number



(a) Number of iterations. **(b)** Execution time.

Figure 4.12: Results for some variants of the Kaczmarz algorithm for the coherent dense data set using a fixed number of columns $n = 1000$ and a varying number of rows.

of iterations and time than the RK method, which shows that, for this data set, sampling rows with probabilities proportional to their norms is not a better row sampling criterion than using a uniform probability distribution. Furthermore, CK and SRKWOR outperform RK and SRK for most dimensions in iterations and time. These results show that sampling without replacement is indeed an efficient way to choose rows - this may happen since, when using a random approach with replacement, the same row can be chosen many times which makes progress to the solution slower.

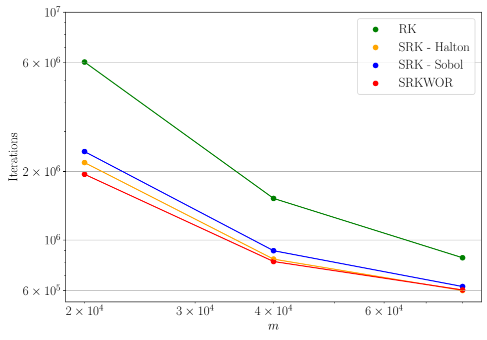
Contrary to the intuition provided by Strohmer and Vershynin [10], the RK inspired method from Figure 4.11 does not outperform CK. As Wallace and Sekmen [29] refer, choosing rows in a random way should only outperform the CK method for matrices where the angle between consecutive rows is very small, that is, highly coherent matrices. To confirm this we also compare the results of the methods used in Figure 4.11 for the third dense data set. Figure 4.12 shows that, for highly coherent matrices, CK does have a slower convergence compared to RK and its random variants.

We finish this section with some simulations of the Kaczmarz method using quasirandom numbers, described in Section 2.6. So far, it seems that, for the first dense data set, the fastest method is SRKWOR. For this reason, we compare sampling rows using the quasirandom numbers generated with the Sobol and Halton sequence with the Randomized Kaczmarz method and SRKWOR.

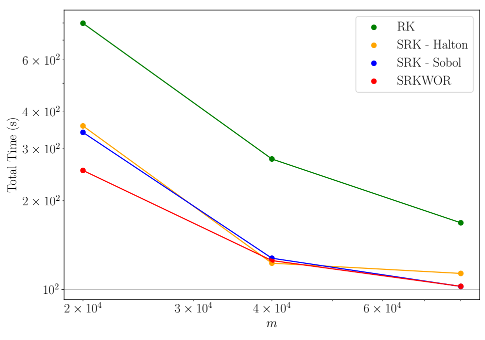
From the results presented in Figure 4.13, we can draw several main conclusions: firstly, the results are similar for the Halton and Sobol sequence in terms of iterations and time; second, quasirandom numbers can outperform RK in both iterations and time; finally, quasirandom numbers can have similar execution times to SRKWOR, depending on the dimension of the problem.

From the analysis of the variants of the Kaczamrz algorithm for consistent systems, we can conclude that, for the data sets that we used, the only methods capable of outperforming the Randomized Kaczmarz method are SRK (using random and quasirandom numbers) and SRKWOR. From all these methods the SRKWOR appears to be the fastest.

The results using the second data set with similar row norms are in general very similar to the ones



(a) Number of iterations.



(b) Execution time.

Figure 4.13: Results for some variants of the Kaczmarz algorithm for dense systems using a fixed number of columns $n = 10000$ and a varying number of rows.

presented here for the first data set. The only major difference is that RK and SRK have similar results since, by having rows with similar norms, the probability distribution used by RK will be similar to a uniform probability distribution, used by SRK. A more in-depth analysis is presented in Appendix D.

4.2.3 Variants of the Kaczmarz Algorithm for Least-Squares Systems

In this section, we discuss the results for the REK and RGS methods. We will also show that, for the data set that we generated, other benchmark methods are faster. In Section 4.2.1, we compared the RK method with CG and CGLS and concluded that the CGLS is faster than the CG method for overdetermined systems. For this reason, we will compare REK and RGS with CGLS. The execution time of the three methods, presented in Figure 4.14, shows that, regardless of the dimension of the problem, CGLS is faster than REK and RGS. Between REK and RGS, RGS is slightly faster than REK. In conclusion, REK and RGS are not the most suitable methods for solving least-squares problems.

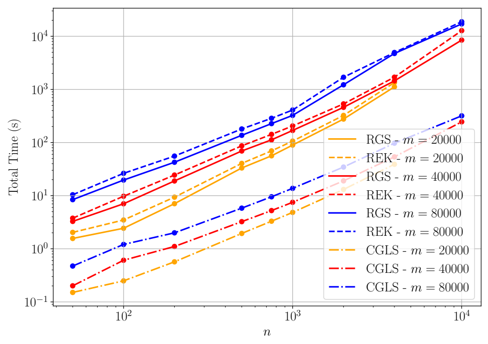


Figure 4.14: Execution time for REK, RGS and CGLS for dense inconsistent systems using a fixed number of rows and a varying number of columns.

Chapter 5

Parallel Implementations Using Shared Memory

It is not a trivial task to parallelize the Kaczmarz algorithm, as it is an iterative algorithm where each iteration depends on the previous one. There are, nonetheless, two main strategies for the parallelization of iterative algorithms (Section 3.6): block-sequential, where we parallelize the work inside each iteration; and block-parallel, where several iterations are distributed and computed in parallel, after which the results are combined. A block-sequential approach is only efficient if there is a substantial amount of work in each iteration to make up for the overhead of parallelization. This is not the case for the Kaczmarz algorithm, where the work in each iteration consists in computing the internal product $\langle A^{(i)}, x^{(k)} \rangle$ and updating each entry of the solution, a relatively small task compared to methods that require, for example, matrix-vector multiplication. Thus, as the amount of work in each iteration of the Kaczmarz algorithm is proportional to the number of columns in matrix A, n, a block-sequential approach will be inadequate for systems with small n. In this chapter, we present several shared memory approaches to the parallelization of RK and SRKWOR using both block-sequential and block-parallel strategies.

The organization of the chapter is as follows. In Section 5.1, we discuss the implementation and results of parallelization using a block-sequential approach. The remaining Sections in this chapter are dedicated to block-parallel approaches. In Section 5.2 we parallelize the RKA method presented in Section 3.5. In Sections 5.3 and 5.4, we discuss two new approaches, based on RKA. In Section 5.5, we discuss an almost asynchronous parallel implementation of RK. Finally, in Section 5.6 we discuss the application of some of these parallelization approaches to solving inconsistent systems.

The implementations of the several methods were accomplished using the OpenMP[1] C++ API. All the experiments and results reported throughout this chapter were conducted on the Accelerates Cluster. Each node of the cluster has two central processing units (Intel Xeon E5-2680 v2 CPU) with 32 GB memory and each CPU has 10 cores. Our simulations use 1, 2, 4, and 8 cores from a given CPU. Regarding the data sets that were used, the simulations in Sections 5.1 to 5.5 used consistent systems taken from the first dense dataset discussed in Section 4.1.3. Only for the last section of the chapter, Section 5.6, did we use inconsistent systems from the least-squares problems dataset.

[1]https://www.openmp.org/

5.1 Parallelization of Each Iteration

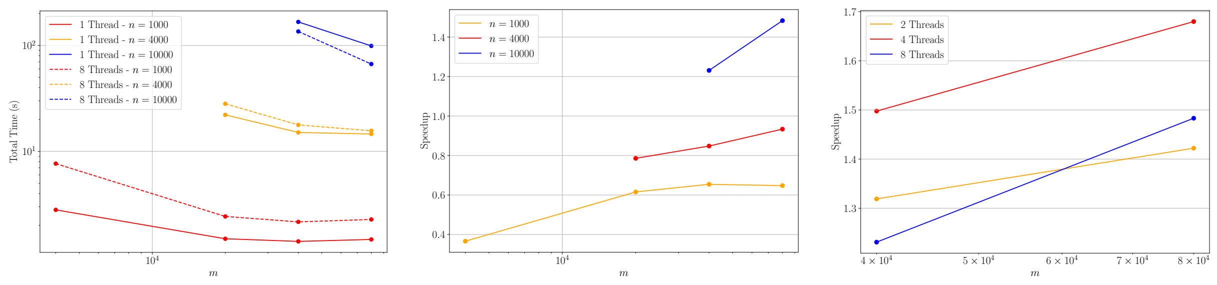
Algorithm 1 Pseudocode for an iteration of the parallel implementation of RK. x corresponds to the estimate of the solution in the current iteration, it. $A_i^{(row)}$ corresponds to the i-th column of a given row of matrix A. $\mathcal{D}$ is a probability distribution that samples row indices with probability proportional to their norms.

```
1: $it \leftarrow it + 1$
2: OMP single
3:     $row \leftarrow$ sampled from $\mathcal{D}$
4:     $dot\_product \leftarrow 0$
5: OMP for reduction (+:dot_product) $i = 0, ..., N$ do
6:     $dot\_product \leftarrow dot\_product + A_i^{(row)} \times x_i$
7: OMP single
8:     $scale \leftarrow \frac{b_{row} - dot\_product}{\|A^{(row)}\|_2^2}$
9: OMP for $i = 0, ..., N$ do
10:        $x_i \leftarrow x_i + scale \times A_i^{(row)}$
```

In a first attempt at parallelization, we show that using a block-sequential approach that parallelizes the work inside each iteration does not always exhibit speedup and, when it does, it is far from ideal, which was somewhat expected due to the small workload per iteration. Algorithm 1 presents the pseudocode for an iteration of the parallel implementation of RK using a block-sequential approach. For the purpose of comparison, the pseudocode for the sequential implementation of RK is in Algorithm 8 in Appendix E. As previously mentioned, the main computations in each iteration are the calculation of the internal product and the update of the solution. The former can be effortlessly parallelized using the OpenMP *reduce* command with the sum operation (lines 4 to 6 of Algorithm 1). The latter is easily handled by distributing the entries of the solution by the available threads using the OpenMP *for* command since the update of the entries of x can be done independently (lines 9 and 10 of Algorithm 1).

Our goal is to evaluate the performance of the parallelization of iterations only and not take into account the stopping criteria. To accomplish this, we run the sequential version of the algorithm and compute the error, $\|x^{(k)} - x^*\|$, in each iteration. When the error drops below a given tolerance, $\varepsilon = 10^{-5}$, the algorithm is terminated and the number of completed iterations is saved. Then, both the sequential and parallel versions of the algorithm are run for the number of iterations previously computed. This allows us to have execution times for the sequential and parallel implementations that do not take into account stooping criteria so that we can compute the corresponding speedup.

The block-sequential implementation of SRKWOR can also be described by Algorithm 1 with the exception of the row selection (line 3). As previously described in Section 4.1, the row selection for SRKWOR consists in shuffling an index array and going through it in a cyclical fashion. We have also



(a) Execution time for several overdetermined systems as a function of the number of rows for 8 threads.

(b) Speedup for several overdetermined systems as a function of the number of rows for 8 threads.

(c) Speedup for systems with $n = 10000$ as a function of the number of rows for 2, 4, and 8 threads.

Figure 5.1: Results for the parallel implementation of RK using a block-sequential approach.

implemented a block-sequential parallelization of SRKWOR. However, since the results and analysis are identical to that of RK, we will not discuss it here.

This implementation of the parallelization of RK was run for 1, 2, 4, and 8 threads. Figures 5.1a and 5.1b show the execution time and speedup for 8 threads, where speedup is defined as the quotient of the sequential execution time and the parallel time using 8 threads. It is clear that the parallel implementation is only faster than the sequential implementation for $n = 10000$. We can also note that speedups are far from linear when compared to the number of threads and that they increase with the number of columns, which was anticipated since increasing the workload per iteration attenuates the overhead of parallelization. Figure 5.1c shows the speedup using other numbers of threads for larger systems ($n = 10000$). It is clear that, although speedups are observed in all experiments, these are far from the ideal value, equal to the number of threads.

5.2 Randomized Kaczmarz with Averaging

Given that it was shown in the previous section that it is not possible to implement an efficient block-sequential parallelization of the Randomized Kaczmarz algorithm, we now move on to block-parallel implementations. In this section, we implement the RKA algorithm (Section 3.5) and a variation of the RKA algorithm using sampling without replacement.

In each iteration of RKA, each thread samples a row of the matrix, computes an updated version of the estimate of the solution, and then the results for all threads are averaged. In this implementation, we decided to use uniform row weights ($w_i = \alpha$), meaning that we can rewrite (3.12) such that

$$x^{(k+1)} = x^{(k)} + \frac{\alpha}{q} \sum_{i \in \tau_k} \frac{b_i - \langle A^{(i)}, x^{(k)} \rangle}{\|A^{(i)}\|_2^2} A^{(i)T} . \tag{5.1}$$

where q is the number of threads. Note that, if $q = 1$, we recover the RK method. Using the previous equation, we implemented a sequential version of RKA so that we could validate the results obtained by

Algorithm 2 Pseudocode for an iteration of the parallel implementation of RKA.

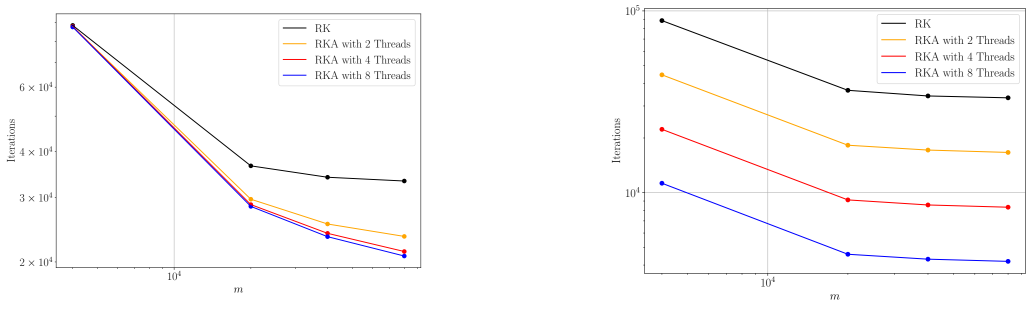
```
1: $it \leftarrow it + 1$
2: OMP barrier
3: OMP for $i = 0, ..., N$ do
4:     $x_i^{(prev)} \leftarrow x_i$
5: $row \leftarrow$ sampled from $\mathcal{D}$
6: $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(prev)} \rangle}{q \, \|A^{(row)}\|_2^2}$
7: OMP critical
8:     for $i = 0, ..., N$ do
9:         $x_i \leftarrow x_i + scale \times A_i^{(row)}$
```

the authors of RKA and evaluate the effectiveness of the parallelization (Algorithm 9 in Appendix E). We will now go through the details of the computations involved in each iteration of the parallel implementation of RKA, presented in Algorithm 2.

In lines 3 and 4 of Algorithm 2 we store the estimate of the solution from the previous iteration, $x^{(prev)}$. This is necessary since, if the scale factor in line 6 was computed using the current estimate of the solution, x, one thread could be computing the scale factor while another thread was updating x in the critical section ahead and the scale factor would not have the correct value.

In line 5 of Algorithm 2 a row is sampled according to a probability distribution proportional to the norms of the rows of matrix A. However, it does not make sense to have threads sampling the same sequence of rows since we would be averaging identical results. This can be easily avoided by giving each thread a different seed for the random number generator that samples rows. Nonetheless, it is not only possible that different threads sample the same row, but also likely, if rows have very different norms.

In lines 7 to 9 of Algorithm 2 the results are combined. To ensure that all threads update the estimate of the solution and that no two threads are updating x simultaneously, a critical section must be created, meaning that the gathering of results is done sequentially. To try to introduce some parallelism in this step two other approaches were tested: in the first, different threads start updating x in a different entry, and the *atomic* command is used; in the second we use the *reduce* command on the solution vector with the sum operation, but this requires previously setting x to zero. Both approaches proved to be slower than using the critical section. It is also important to mention that the time respective to combining the results is directly proportional to the number of threads. Note that, since there is no implicit barrier at the end of the critical section, a synchronization point was introduced in line 2, so that we avoid having one thread updating $x^{(prev)}$ in line 4 while another thread is still in the previous iteration updating x in line 9. We can already identify two problems in the parallel implementation of this algorithm: not only do we have a synchronization point in each iteration, but also the averaging of results is done sequentially.



(a) $\alpha = 1$.

(b) $\alpha = \alpha^*$.

Figure 5.2: Iterations for RK and RKA using 2, 4, and 8 threads for several overdetermined systems with $n = 1000$ with a varying number of rows.

As in Section 5.1, we do not consider the stopping criteria when measuring execution times. For that, we measure the necessary number of iterations to reach convergence by running the sequential and parallel algorithms until $\|x^{(k)} - x^*\| < 10^{-5}$ is verified. We then measure the execution time by rerunning the algorithm for the previously computed number of iterations. Note that the number of iterations will differ between RK and RKA. The discussion of the results for RKA is present in Section 5.2.1.

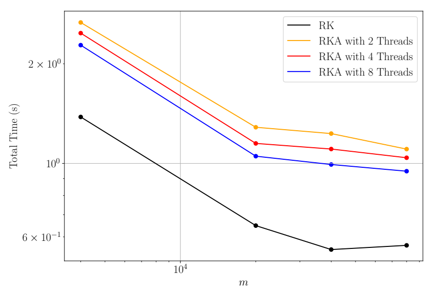
Since the RKA algorithm can improve the convergence of the RK method, we developed a variation of RKA called Simple Randomized Kaczmarz with Averaging Without Replacement (SRKAWOR), with the goal of increasing the rate of convergence of SRKWOR. An iteration of SRKAWOR can also be described by Algorithm 2 with the exception of the row selection in line 5, since SRKAWOR samples rows without replacement and not based on their norms. In Section 5.2.2 we test two approaches to sampling without replacement and discuss the results for SRKAWOR.

5.2.1 Results for the Randomized Kaczmarz with Averaging Method

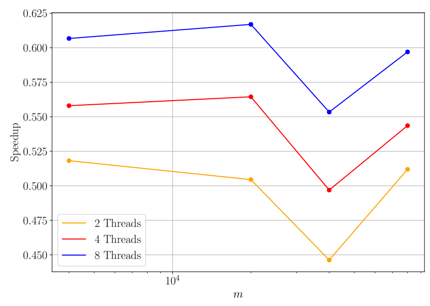
In this section, we discuss the results for the RKA method that samples rows based on their norm. Two choices for the parameter α (5.1) were made: $\alpha = 1$ and $\alpha = \alpha^*$, where α^* is the optimal parameter for RKA for consistent systems, given by (3.14). The sequential and parallel versions of RKA were tested for 2, 4, and 8 threads (1 thread is the RK method). We start by showing how RKA can be used to accelerate the convergence of RK.

Figure 5.2 shows the number of iterations using the two choices of α previously discussed. It is clear that, regardless of α, the number of iterations of RKA is inferior to that of RK. Furthermore, when the number of threads is increased, fewer iterations are needed for RKA to converge. However, the reduction in the number of iterations is much more significant when using the optimal values for the weights. For this reason, the results presented in the remainder of this section were obtained using the optimal row weights α^*.

Figure 5.3 shows the execution time and speedup using optimal row weights α^*. Although Fig-



(a) Execution time.

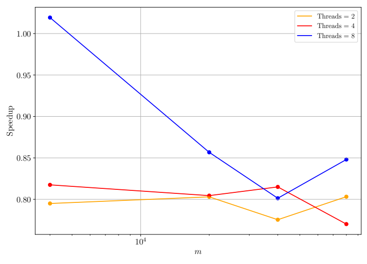


(b) Speedup.

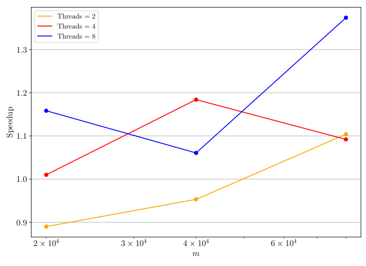
Figure 5.3: Results for RK and the parallel implementation of RKA using 2, 4, and 8 threads for several overdetermined systems with $n = 1000$ with a varying number of rows.

ure 5.2b shows that RKA requires fewer iterations to converge than RK, Figure 5.3a shows that, regardless of the number of threads, RKA is a slower method than RK. Note that, for RKA, regardless of the number of threads, the work done by each thread during the computation of the results corresponding to one row of matrix A is the same as RK (lines 5 and 6 of Algorithm 2). This means that the increase in execution time for RKA can only be due to the averaging of results. Although the number of iterations is smaller for a larger number of threads, this decrease is not enough to make up for the time spent in updating $x^{(k)}$, which has to be done sequentially. Figure 5.3b represents the speedup computed as the quotient between the execution time of RK and the execution time of the parallel implementation of RKA for the several numbers of threads and shows that speedups are far from linear.

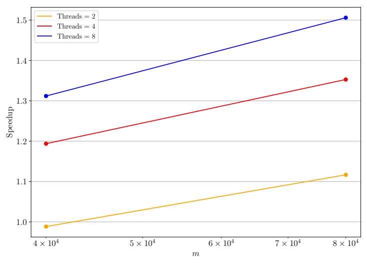
We now evaluate the parallelization of the RKA algorithm by comparing it with its sequential version. Figure 5.4, contains the speedup calculated as the quotient between the total execution time of the sequential and parallel implementations of RKA for several threads. Note that regardless of the number of threads, speedups improve for data sets with larger n. This is expected since a larger number of columns means that the work per iteration also increases and, consequently does the execution time. However, speedups are still far from ideal and there are cases where the sequential version is faster than the parallel version, for example, for almost all threads and all values of m using $n = 1000$ (Figure 5.4a).



(a) Systems with $n = 1000$.

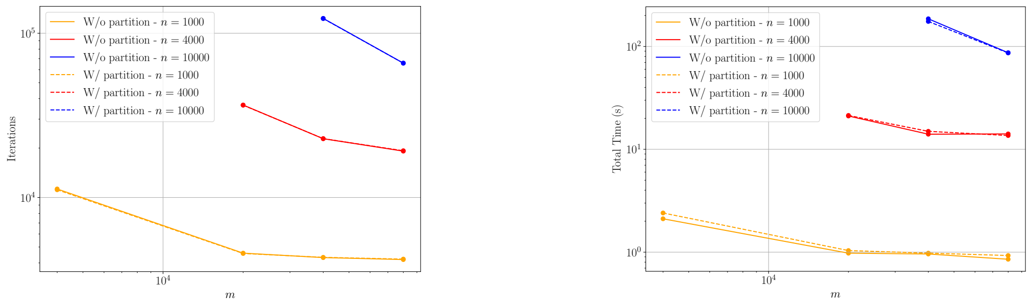


(b) Systems with $n = 4000$.



(c) Systems with $n = 10000$.

Figure 5.4: Speedup for RKA using 2, 4, and 8 threads for several overdetermined systems using a fixed number of columns and a varying number of rows.



(a) Iterations using 8 threads.

(b) Execution time using 8 threads.

Figure 5.5: Results for the parallel implementation of RKA using a partition scheme for the rows of matrix A.

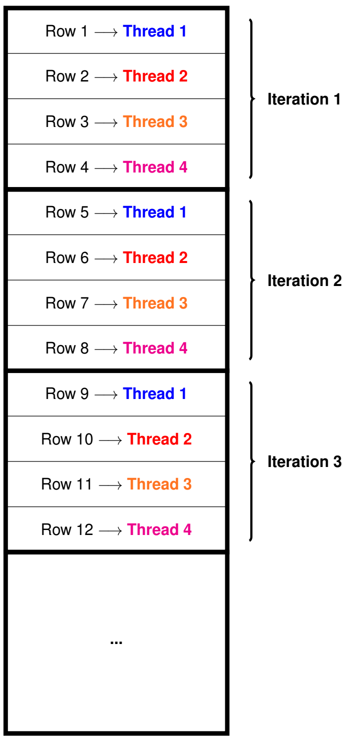
As previously mentioned several threads may sample the same row in a given iteration. To force threads to sample different rows, we developed a variation of the RKA algorithm where matrix A is distributed among threads. Therefore, the sequential and parallel implementation of this algorithm can still be described by Algorithms 2 and 9 with the exception of line 5 where rows are sampled. To partition the system between threads we used a distributed approach. For a given thread with identifier t_{id}, its block of rows spans from index $low = \lfloor t_{ID} \frac{m}{q} \rfloor$ to index $high = \lfloor (t_{ID} + 1) \frac{m}{q} \rfloor - 1$, inclusively. Figure 5.5 shows the difference between partitioning the matrix between 8 threads (or not). It is clear that there is no advantage in using this partition scheme since the number of iterations and execution time suffer minor changes when compared to each thread having access to the entire matrix.

In conclusion, it is not possible to efficiently parallelize RKA due to the large overhead in communication between threads, especially when averaging the results. For efficient parallelization, we need to find a way to average the results in parallel or to decrease the number of times that the results are averaged and, therefore, decrease the impact of communication. This last approach will be explored in the next block-parallel approach, presented in Section 5.3.

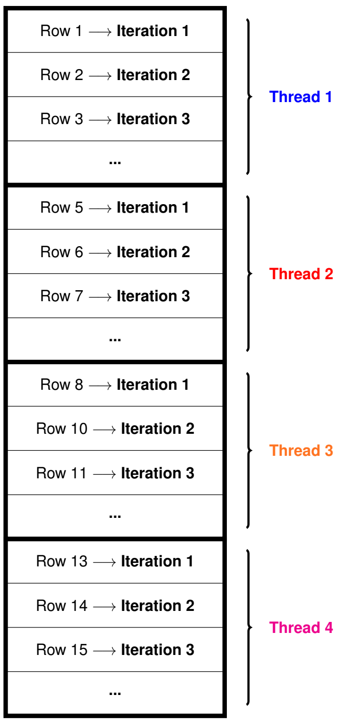
5.2.2 Results for the Simple Randomized Kaczmarz with Averaging Without Replacement Method

In this section, we discuss the results for the SRKAWOR method that uses sampling without replacement. In the previous section, we used two options for parameter α and concluded that using the optimal values α^* significantly decreases the number of iterations. However, the optimal values are only valid for row sampling using a probability distribution that depends on row norms, meaning that it is not anticipated that α^* are the optimal values for SRKAWOR.

As previously mentioned, the implementation of the SRKAWOR method is identical to the RKA, except for the technique used to sample rows. We will now explain in detail how to sample rows without replacement while guaranteeing that, in a given iteration, no two threads are computing results based



(a) In the first sampling option rows are sampled cyclically.



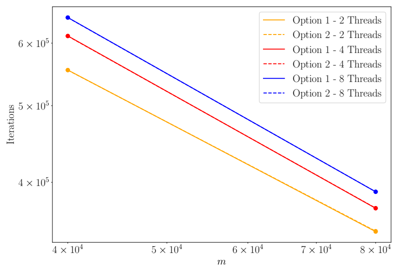
(b) In the second sampling option the matrix is divided between threads.

Figure 5.6: Two options for sampling without replacement using an example with 4 threads.

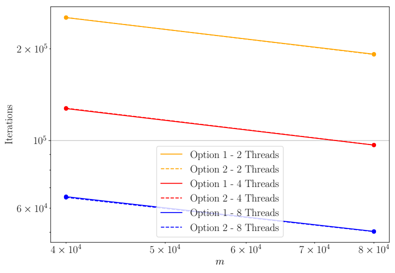
on the same row. Similarly to SRKWOR, we use a shuffled array of indices to represent a reordered version of matrix A. There are two possible sampling schemes to select row indices for the shuffled array. In the first option, represented in Figure 5.6a, entries of the array are used cyclically - in each iteration threads selected entries according to their ID such that the index selected in iteration it is $idx = mod(q \times it + t_{ID}, m)$.

In the second option, represented in Figure 5.6b, the array is split between threads and then each thread cyclically uses its block of entries. The index of the first entry of the block correspondent to a thread with identifier t_{ID} is given by: $B_{low} = \lfloor t_{ID} \times \frac{m}{q} \rfloor$ and the size of that block is given by $B_{size} = \lfloor (t_{ID}+1) \times \frac{m}{q} \rfloor - \lfloor t_{ID} \times \frac{m}{q} \rfloor$. In summary, a thread with identifier t_{ID} uses, in iteration it, the row with index $idx = B_{low} + mod(it, B_{size})$.

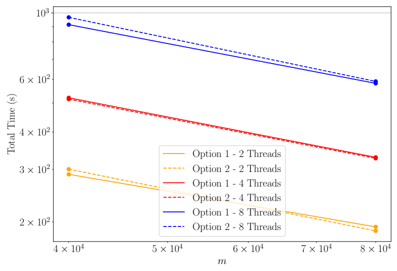
To determine which sampling scheme is better, we ran the parallel implementation of SRKAWOR for several numbers of threads, whose results are shown in Figure 5.7. Note that there is no distinguishable difference between the two options for both choices of parameter α in terms of the number of iterations. Regarding execution time, using $\alpha = 1$, the difference between the two options is also minimal, as seen in figure 5.7c. However, for $\alpha = \alpha^*$, using 8 threads, execution time using the second option is inferior to the first option. In conclusion, there is only a thread for one of the options of α where there is a noticeable



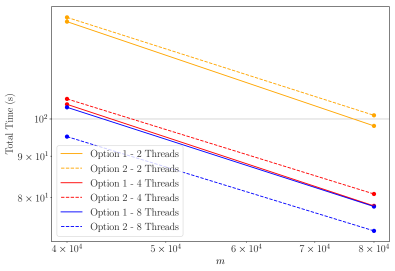
(a) Iiterations for the two sampling schemes of SRKAWOR using $\alpha = 1$.



(b) Iterations for the two sampling schemes of SRKAWOR using $\alpha = \alpha^*$.



(c) Execution time for the two sampling schemes of SRKAWOR using $\alpha = 1$.

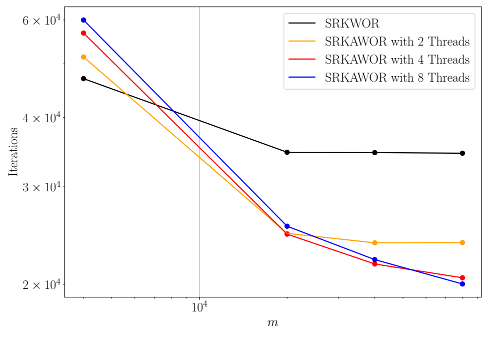


(d) Execution time for the two sampling schemes of SRKAWOR using $\alpha = \alpha^*$.

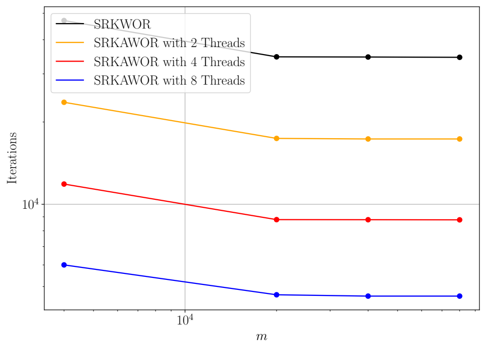
Figure 5.7: Results for SRKAWOR using 2, 4, and 8 threads for several overdetermined systems using a fixed number of columns of $n = 10000$ and a varying number of rows.

difference in execution time. Since this is not enough to determine if there is a sampling option that has better performance, we will use option one as the default sampling option from now on.

Now we compare the performance of the parallel implementation of SRKAWOR with the SRKWOR method for both options of α. The results in Figure 5.8 show that contrary to RKA (Figure 5.2), for parameters $\alpha = 1$, the number of iterations is not necessarily smaller for a larger number of threads, as seen in Figure 5.8a. For example, for a system, 40000×1000, the number of iterations using 8 threads is larger than that for 4 threads. On the other hand, using $\alpha = \alpha^*$, regardless of the dimension of the

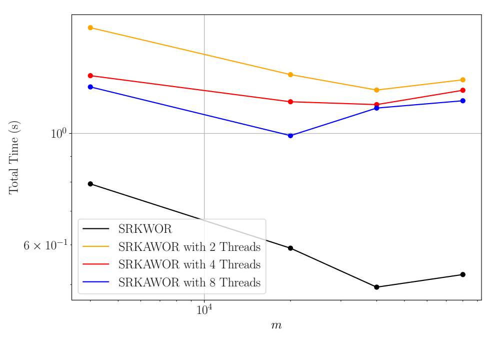


(a) $\alpha = 1$.

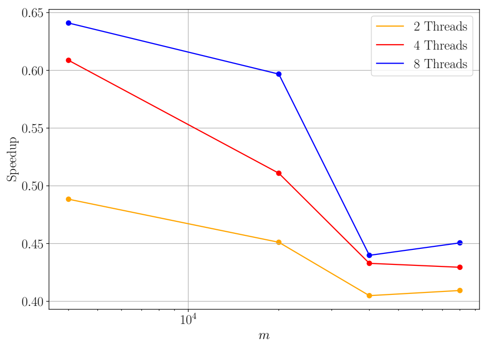


(b) $\alpha = \alpha^*$.

Figure 5.8: Iterations for SRKWOR and SRKAWOR using 2, 4, and 8 threads for several overdetermined systems with $n = 1000$ with a varying number of rows.



(a) Execution time.



(b) Speedup.

Figure 5.9: Results for SRKWOR and the parallel implementation of SRKAWOR using 2, 4, and 8 threads for several overdetermined systems with $n = 1000$ with a varying number of rows.

system, it is always beneficial to use more threads. This indicates that the optimal parameter α for SRKAWOR is closer to the optimal parameter α^* calculated for RKA than to $\alpha = 1$. For this reason, the results presented in the remainder of this section were obtained using $\alpha = \alpha^*$.

The execution time and speedup results in Figure 5.9 are very similar to the results in Figure 5.3. Once more, although SRKAWOR (with $\alpha = \alpha^*$) requires fewer iterations to converge than SRKWOR, that decrease in iterations is not enough to compensate for the time spent in averaging the results, making SRKAWOR a slower method than SRKWOR.

The speedups calculated as quotient between the total execution time of the sequential and parallel implementations of SRKAWOR for several threads exhibit very similar results to those from Figure 5.4, so we will not show them here. The main conclusion for SRKAWOR is the same as for RKA: the communication has a dominant impact on execution time and, therefore, efficient parallelization is not possible.

5.3 Randomized Kaczmarz with Averaging with Blocks

Efficient parallelization of RKA is not possible due to the large cost of communication that happens in every iteration. To decrease the impact of communication we developed a variation of the RKA method called Randomized Kaczmarz with Averaging with Blocks (RKAB). In a single iteration of RKA, each thread only processes one row of the matrix before the results are averaged. In RKAB, instead of each thread only processing one row per iteration, the results corresponding to a block of several rows are computed, meaning that the results from several threads are only gathered once in a while. We can already identify one advantage and one disadvantage of RKAB in regards to RKA: on one hand, communication will happen much more sporadically; on the other hand, since results are only shared between threads after processing a block of rows, there is less shared information between threads.

Just like for RKA, we developed a sequential version of RKAB (Algorithm 10 in Appendix E). Here,

Algorithm 3 Pseudocode for an iteration of the parallel implementation of RKAB.

```
1: $it \leftarrow it + 1$
2: OMP barrier
3: $row \leftarrow$ sampled from $\mathcal{D}$
4: $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x \rangle}{\|A^{(row)}\|_2^2}$
5: for $i = 0, ..., N$ do
6:     $x_i^{(thread)} \leftarrow x_i + scale \times A_i^{(row)}$
7: for $b = 0, ..., block\ size - 1$ do
8:     $row \leftarrow$ sampled from $\mathcal{D}$
9:     $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(thread)} \rangle}{\|A^{(row)}\|_2^2}$
10:    for $i = 0, ..., N$ do
11:        $x_i^{(thread)} \leftarrow x_i^{(thread)} + scale \times A_i^{(row)}$
12: for $i = 0, ..., N$ do
13:     $x_i^{(thread)} \leftarrow x_i^{(thread)} - x_i$
14: OMP barrier
15: OMP critical
16:     for $i = 0, ..., N$ do
17:         $x_i \leftarrow x_i + \frac{x_i^{(thread)}}{q}$
```

we discuss a detailed explanation of the work inside a single iteration of the parallel implementation of RKAB presented in Algorithm 3.

In Algorithm 3, instead of saving the previous iteration (Algorithm 2), every thread has a private variable $x^{(thread)}$ that stores the current results for that thread. In the first row of the block, threads use the estimate of the solution from the previous iteration (lines 3 to 6 of Algorithm 3). For the remaining rows of the block, threads use their local estimative of the solution, $x^{(thread)}$, to compute the scale factor (lines 7 to 11 of Algorithm 3). Note that the size of the block, $block\ size$, has to be determined by the user and that using the RKAB method with $block\ size = 1$ is equivalent to the RKA method. Later on, we will discuss how to choose $block\ size$.

The process of averaging the results is slightly different from RKA. In each thread, after the results of the entire block are computed, we subtract x to $x^{(thread)}$ so that we can update x by summing the changes. Another option would be setting x to 0 and summing the entirety of the results but the computational effort is the same. Note that we need the barrier in line 14 of Algorithm 3 so that we do not have one thread updating x in the critical region while another thread is behind computing $x^{(thread)}$ in line 13.

The stopping criterion will be the same as the one used for RKA. We compute the number of iterations necessary to have $\|x^{(k)} - x^*\| < 10^{-5}$ and then use it as a maximum number of iterations to measure the execution time of the algorithm.

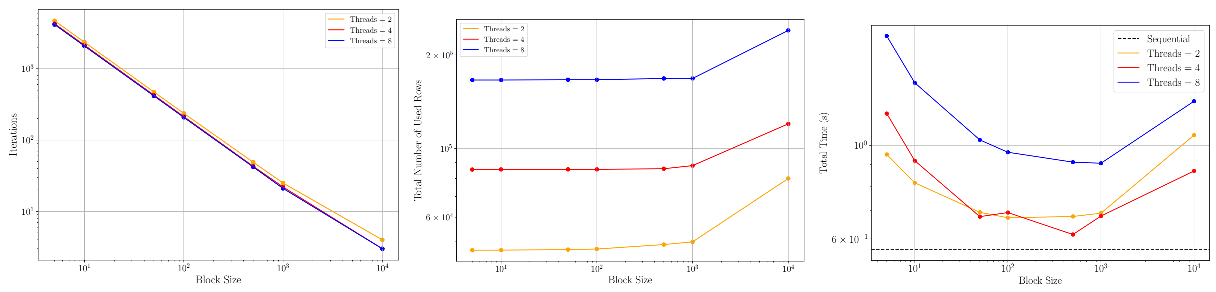
Just like we developed a variation of RKA that uses sampling without replacement (SRKAWOR), here we will also do the same with RKAB, with the Simple Randomized Kaczmarz with Averaging with Blocks Without Replacement (SRKABWOR) method. Once more, SRKABWOR can be described with the same pseudocode as RKAB (Algorithm 3), with the exception of lines 3 and 8 where the rows are sampled.

In Section 5.3.1 we walk through the results for RKAB and compare it with the RK method. In Section 5.3.2 we discuss the details of sampling without replacement and discuss the results for SRKABWOR.

5.3.1 Results for the Randomized Kaczmarz with Averaring with Blocks Method

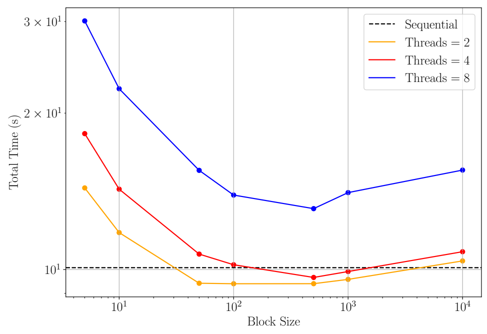
During the implementation of RKA we used uniform row weights, α, and analyzed the results using $\alpha = 1$ and $\alpha = \alpha^*$, where α^* are the optimal parameters for consistent systems. However, RKAB is a different algorithm than RKA and there is no calculated optimal value for α. For that reason, we will first analyze the dependency of RKAB on other parameters like the $block\ size$ using $\alpha = 1$. At the end of this section, we will evaluate how several choices of row weights, α, can impact the performance of RKAB.

We will start by analyzing how the execution time of RKAB depends on the size of the blocks by setting $block\ size = \{5, 10, 50, 100, 500, 1000, 10000\}$. Similarly to the previous parallelization attempts, the sequential and parallel versions of RKAB were tested for 1, 2, 4, and 8 threads. Figure 5.10 shows the results for RKAB for a system with dimensions 80000×1000. Figure 5.10a shows that, for all threads, when we increase $block\ size$, the number of iterations decreases. This was expected since, by processing a larger number of rows in each iteration, the estimate of the solution will converge faster than if only the use row was used. Furthermore, for fixed $block\ size$, larger numbers of threads require fewer iterations. However, that decrease is small, since the total number of used rows, shown in Figure 5.10b, increases when more threads are used. Figure 5.10b also shows that, for a given number of threads, regardless of

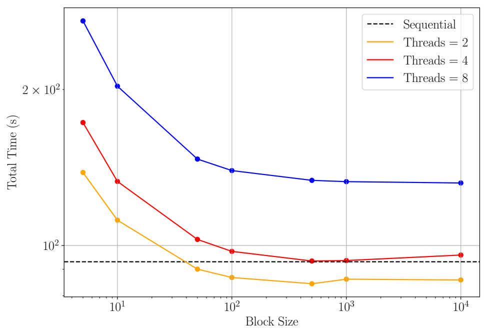


(a) Iterations for a system 80000×1000.
(b) Number of lines used for a system 80000×1000.
(c) Total computational time for a system 80000×1000.

Figure 5.10: Results for the parallel implementation of RKAB using 2, 4, and 8 threads for an overdetermined system 80000×1000 for several values of $block\ size$.



(a) Total computational time for a system 80000×4000.



(b) Total computational time for a system 80000×10000.

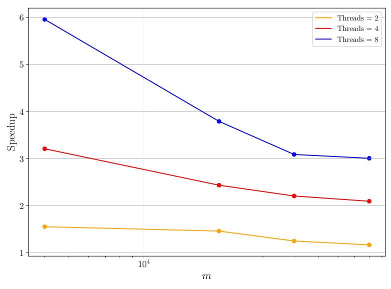
Figure 5.11: Results for the sequential version of RK and the parallel implementation of RKAB using 2, 4, and 8 threads for several overdetermined systems.

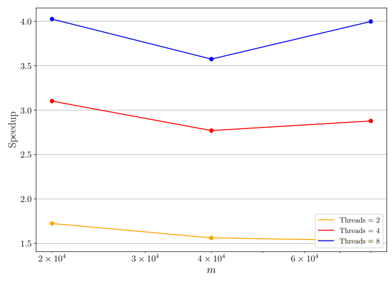
$block\ size$, the total number of used rows stays the same (with the exception of the $block\ size = 10000$). The results in Figure 5.10c can be easily explained by using the number of iterations and the total number of rows. First, for blocks of a fixed size, time generally increases with the number of threads since, although iterations decrease, that decrease is not large enough to make up for the overhead in synchronization. Second, increasing the block size, in general, decreases time since, although the total amount of work is similar between block sizes (see Figure 5.10b), the number of times that the threads have to communicate to average the results is smaller. The larger block size is a special case since it is the only one for which the total number of used rows and time increase. Note that this is the only value for which $block\ size > n$. For a full rank matrix, using the same number of rows as columns is enough information to solve the system. This means that using a block size much larger than n makes it so that the individual solution estimates in each thread are already close to the real solution and, therefore, that there is little to no benefit in averaging similar results.

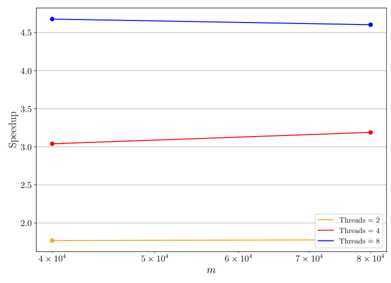
To further validate the conclusions regarding the block size, Figure 5.11 shows the results for a system with other numbers of columns and it also shows the sequential time of RK. Note that, once more, for a system 80000×4000, time increases when we go from $block\ size = 1000$ to $block\ size = 10000$. However, for the system with 80000×10000, time does not increase for the larger block size. In summary, we can use the number of columns as a rule of thumb to select the block size. Still, just like RKA, for most numbers of threads and for most block sizes, the parallel implementation is not faster than the sequential implementation.

We will now evaluate the parallelization of the RKAB algorithm by comparing it with its sequential version. Figure 5.12, contains the speedup calculated as the quotient between the total execution time of the sequential and parallel implementations of RKAB for several threads using $block\ size = n$. The results show much higher speedups than the ones obtained with RKA (see Figure 5.12).

In conclusion, although the parallelization of the RKAB can be done more effectively than the paral-







(a) Systems with $n = 1000$.

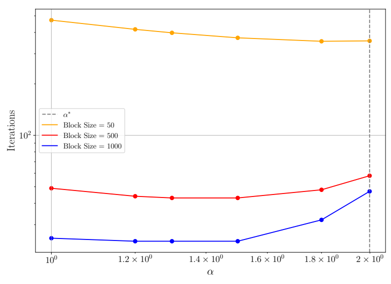
(b) Systems with $n = 4000$.

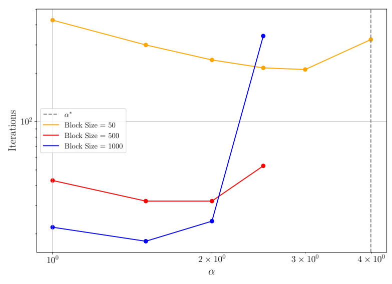
(c) Systems with $n = 10000$.

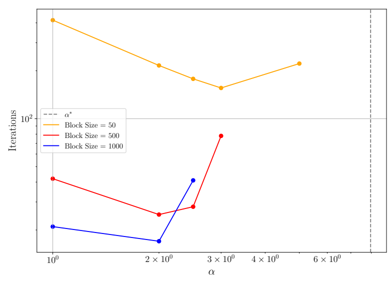
Figure 5.12: Speedup for RKAB using 2, 4, and 8 threads for several overdetermined systems using a fixed number of columns and a varying number of rows. *block size* was set to the number of columns.

lelization of RKA, the RKAB method, just like RKA, is not faster than the sequential RK for most systems in our data set.

We now discuss how different row weights, given by parameter α, can affect the performance of RKAB. For each number of threads, several values of α were chosen between 1 and the optimal value of α, α^*, for RKA (5.1). Since the values of α^* for a system 80000×1000 are 1.999, 3.992 and 7.962 for 2, 4, and 8 threads, we choose $\alpha = \{1.0, 1.2, 1.3, 1.5, 1.8, 1.999\}$ for 2 threads; $\alpha = \{1.0, 1.5, 2.0, 2.5, 3.0, 3.991\}$ for 4 threads; and $\alpha = \{1.0, 2.0, 2.5, 3.0, 5.0, 7.962\}$ for 8 threads. Figure 5.13 shows the results for the system with dimensions 80000×1000. Note that, for 4 and 8 threads, there are a few values of α for which the number of iterations is not shown (for example, 4 threads using *block size* $= 500$ and $\alpha = 3.0$). These cases are not shown because the RKAB does not converge for these values of α. This conclusion was obtained by confirming that the error $\|x^{(k)} - x^*\|$ is increasing with each iteration, meaning that the estimation of the solution is not approaching the solution of the system. Figure 5.13a shows that, for 2 threads, α^* is not the optimal value of α for RKAB. Although, regardless of *block size*, RKAB converges for α^*, there are other values of α for which the number of iterations is smaller ($\alpha = 1.8$ for *block size* $= 50$,







(a) Number of iterations using 2 threads using different block sizes.

(b) Number of iterations using 4 threads using different block sizes.

(c) Number of iterations using 8 threads using different block sizes.

Figure 5.13: Number of iterations for the RKAB method as a function of parameter α for a system with dimensions 80000×1000.

$\alpha = 1.3$ and $\alpha = 1.5$ for $block\ size = 500$ and $\alpha = 1.2$, $\alpha = 1.3$ and $\alpha = 1.5$ for $block\ size = 1000$). It also appears that the optimal value of α for RKAB decreases with the increase of $block\ size$. For 4 and 8 threads (Figures 5.13b and 5.13c), not only is α^* not the optimal value of α, but also, depending on the block size, RKAB might not even converge. Just like for 2 threads, for a given number of threads, the optimal α seems to decrease when the block size is increased.

In summary, RKA can converge for a given α while RKAB does not; the optimal values of α for RKA are not the same as the ones for RKAB; the optimal value of α for RKAB depends on $block\ size$.

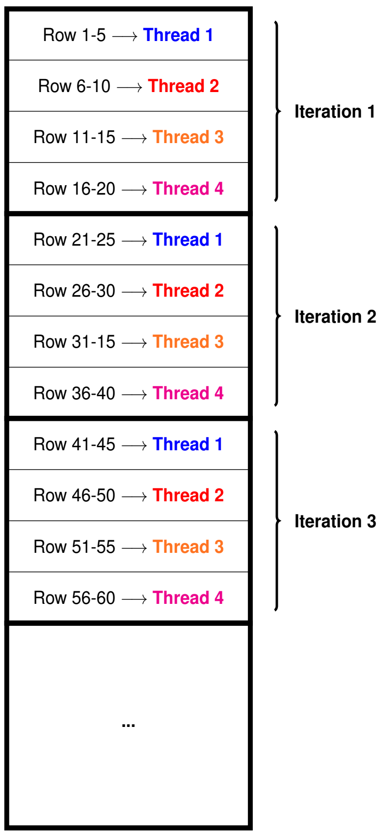
5.3.2 Results for the Simple Randomized Kaczmarz with Averaring with Blocks Without Replacement Method

In this section, we discuss the SRKABWOR method, a variation of RKAB that uses sampling without replacement. Similarly to the SRKAWOR method (Section 5.2.2), there are two possible approaches for sampling blocks of rows without replacement, shown in Figure 5.14.

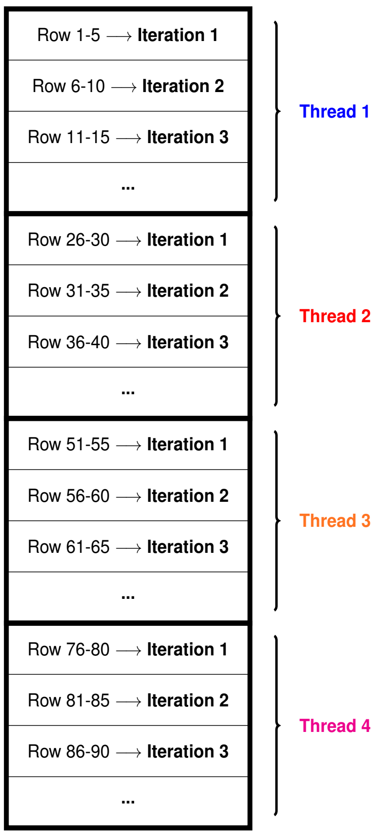
In the first option, represented in Figure 5.14a, entries of the array are used cyclically. Let $idx(r, it, t_{ID})$ be the array index corresponding to r-th row of the block in iteration it for threads with ID. The array entry of the first row that a thread with identifier t_{ID} uses in the first iteration is given by $idx(0, 0, t_{ID}) = block\ size \times t_{ID}$. During that iteration, the array index corresponding to the r-th row in the block is $idx(r, 0, t_{ID}) = mod(idx(0, 0, t_{ID}) + r, m)$. After each iteration, the first row of the block is incremented such that $idx(r, it + 1, t_{ID}) = idx(r, it, t_{ID}) + block\ size \times q$.

In the second option, represented in Figure 5.14b, the array is split between threads and each thread cyclically uses its chunk of entries. As discussed in Section 5.2.2, the chunk of entries own by thread with identifier t_{ID} starts at index $B_{low} = \lfloor t_{ID} \frac{m}{q} \rfloor$ and has size $B_{size} = \lfloor (t_{ID} + 1) \frac{m}{q} \rfloor - \lfloor t_{ID} \frac{m}{q} \rfloor$. Therefore, in iteration it, the array index corresponding to the r-th row in the block is given by $idx(r, it, t_{ID}) = B_{low} + mod(it \times block\ size + r, B_{size})$. Figure 5.15 shows the execution time for the two sampling options for several values of $block\ size$ for two different systems. Note that just like for SRKAWOR, there is no sampling option that is systematically better than the other. For this reason, we will use the first sampling option as the default for this method. Moreover, the evolution of execution time for the several block sizes in Figures 5.15a and 5.15b is very similar to the results for RKAB, in Figures 5.10c and 5.11a. This means that we can adopt the same rule for choosing the block size as the one used for RKAB, that is, using $block\ size = n$. Finally, Figure 5.15 shows that the parallel implementations of SRKABWOR are, in general, slower than the sequential SRKWOR method.

We will now evaluate how the effectiveness of the parallelization of SRKABWOR, by comparing the parallel implementation with the sequential implementation of SRKABWOR. The speedups calculated as quotient between the total execution time of the sequential and parallel implementations of SRKABWOR for several threads exhibit very similar results to those from Figure 5.12, so we will not show them

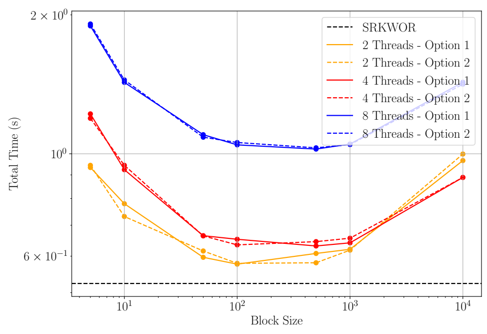


(a) In the first sampling option rows are sampled cyclically.

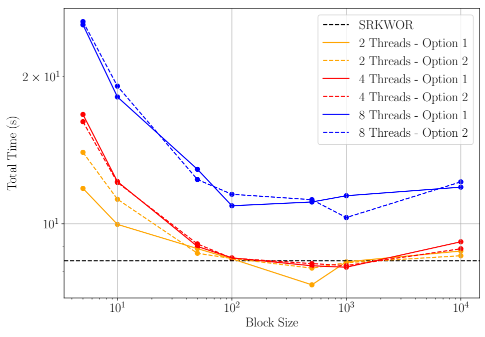


(b) In the second sampling option the matrix is divided between threads. Each thread owns a block of 25 rows.

Figure 5.14: Two options for sampling without replacement using an example with 4 threads and a matrix with 100 rows.



(a) Total computational time for a system 80000×1000.



(b) Total computational time for a system 80000×4000.

Figure 5.15: Results for the parallel implementation of SRKABWOR using 2, 4, and 8 threads for several overdetermined systems.

here. We conclude that, just like RKAB, the parallelization of SRKABWOR is more efficient than that of SRKAWOR, but the SRKABWOR method is still generally slower than the sequential SRKWOR.

5.4 Randomized Kaczmarz with Averaging with Blocks with Repetition

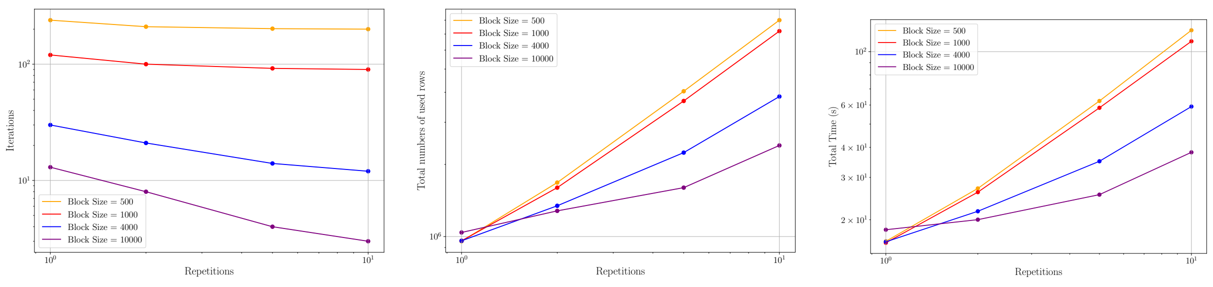
In an attempt to try and improve the RKAB method further, we designed a variation of the method where each block of rows in each thread is processed several times before the results are combined.

There are two advantages to processing the same block more than one time: it increases the amount of work in each iteration, which makes communication more sporadic; and by reusing the rows in the block, we save time in memory access. The disadvantage is that, by reusing rows, we are reprocessing information that has already been used. To implement this variation, since each row is sampled from a probability distribution, we must store the indices of the rows used in each block to that they can be reused. Furthermore, an additional parameter that controls the number of repetitions of each block is needed.

This block repetition approach can also be used for the improvement of the SRKABWOR method (Section 5.3.2). Since there is a direct correspondence between a given iteration and the row indices of the block that will be used, there is no need to store the indices of used rows. However, since the results are very similar to using block repetition for RKAB we will not discuss them here.

We now discuss some results, present in Figure 5.16, for the RKAB method with block repetition for dimension 80000×4000, using 8 threads and $\textit{block size} = \{50, 500, 1000, 4000, 10000\}$. The number of repetitions was chosen as $\textit{repeat} = \{1, 2, 5, 10\}$. Note that using this variation with $\textit{repeat} = 1$ is equivalent to using the RKAB method. Moreover, the row weights given by α were set to 1.

Figure 5.16a shows that, regardless of the size of the block, there is a slight decrease in the number of iterations when the number of repetitions increases. Figure 5.16b shows the total number of used



(a) Iterations. **(b)** Total number of used rows. **(c)** Total execution time.

Figure 5.16: Results for the RKAB method with block repetition for a system with dimensions 80000×4000 using 8 threads. The number of repetitions was set to 1, 2, 5, and 10 and the block size to 500, 1000, 4000, and 10000.

rows, given by the product of the number of iterations, threads, block size, and the number of repetitions. It is clear that, for a given block size, the decrease in iterations is not big enough to compensate for the increase in the total number of used rows. Furthermore, except for $repeat = 1$, larger blocks require fewer rows to converge. This is due to the impact of the number of iterations in the total number of used rows. For the case with no repetitions, since the difference between iterations for the several block sizes is smaller, the block size will have a larger impact in computing the total number of used rows than the number of iterations. In the cases where there is the same block of rows is reused, iterations have a larger impact in computing the total number of used rows. The results in Figure 5.16b directly translate into time, shown in Figure 5.16c.

In summary, there appears to be no advantage in repeating the blocks, regardless of the block size used.

5.5 An Almost Asynchronous Parallel Implementation of the Randomized Kaczmarz Method

The previous attempts at block-parallel implementation of RK such as RKA and RKAB have shown that the synchronization overhead is the main obstacle in building an effective parallelization. To fully eliminate the cost of communication, some methods have each thread update the solution vector at will, like in the AsyRK method. In this section, we will discuss an attempt of creating an asynchronous parallel implementation of RK.

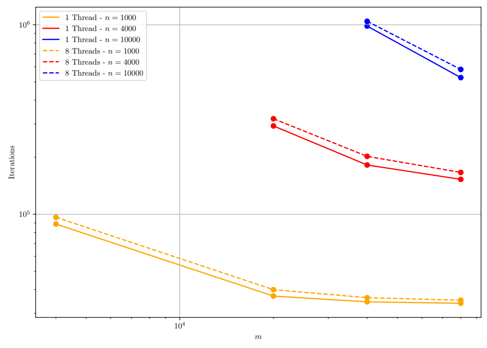
To compute iterations in parallel, different threads must work on different rows of the matrix while still sampling rows according to their norms. This can be easily accomplished by giving each thread a different seed for the random number generator that samples the rows, just like in RKA.

The method for combining the results for this implementation is the following: the entries of the solution vector are distributed among threads such that each thread can only update a different part of $x^{(k)}$. This is to make sure that no two threads are updating the same position of the $x^{(k)}$. The decomposition of the entries of $x^{(k)}$ among threads uses a distributed approach. Considering that $x^{(k)}$ has size n and that the number of threads is t, a thread with identifier t_{ID} updates entries of $x^{(k)}$ starting at index $\lfloor t_{ID}\frac{n}{t}\rfloor$ and ending at index $\lfloor (t_{ID}+1)\frac{n}{t}\rfloor - 1$ inclusively.

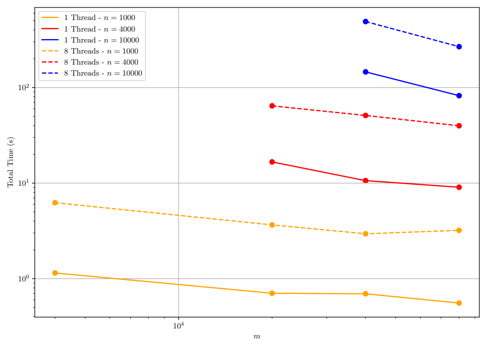
To reduce synchronization, each thread updates $x^{(k)}$ at will, meaning that a thread may read the entries of $x^{(k)}$ to compute $\langle A^{(i)}, x^{(k)}\rangle$ while another thread is updating its segment of $x^{(k)}$. Consequently, the results will be different every time we run the algorithm. Since the results are not reproducible, the stopping criterion used in the previous Sections is unusable due to the fact that a given number of iterations might be enough to achieve a given tolerance in one run of the algorithm but not another. To solve this problem, to measure the execution times of this implementation, we opted for an alternative where we measure the time necessary to achieve $\|x^{(k)} - x^*\| < 10^{-5}$. Just like in the sequential RK, this test is performed every 1000 iterations. Regarding the computation of $\|x^{(k)} - x^*\|$ in the parallel implementation there are two options: first, we could introduce a synchronization point every 1000 iterations and parallelize the computation of the error; second, we could have only one thread computing the error sequentially and have it communicate to the other threads whether they should continue or stop working. We opted for the first option meaning there will still be some level of synchronization. In summary, the execution times for this implementation will take into account the stopping criterion due to the fact that, since this approach is asynchronous, the results for every run of the algorithm are not reproducible.

5.5.1 Results for the Randomized Kaczmarz Method

The results using 1 and 8 threads are presented in Figure 5.17. Simulations using 2 and 4 threads exhibit similar results so we will not discuss them here. Figure 5.17a shows that the parallel version requires more iterations than the sequential version. This increase in iterations is also accompanied by an increase in time. Several factors might be contributing to this ineffective parallelization: for starters, although we let threads update $x^{(k)}$ at will, we still have a synchronization point for the stopping criteria that happens every 1000 iterations, for which threads might be waiting on each other; furthermore, a thread might be using $x^{(k)}$ with segments that still have not been updated by other threads in their previous iteration, in other words, an older and out-of-date version of the estimation of the solution vector.



(a) Iterations.

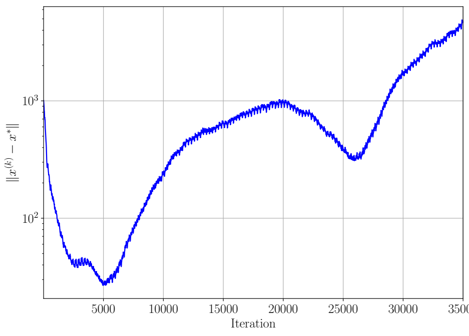


(b) Execution time.

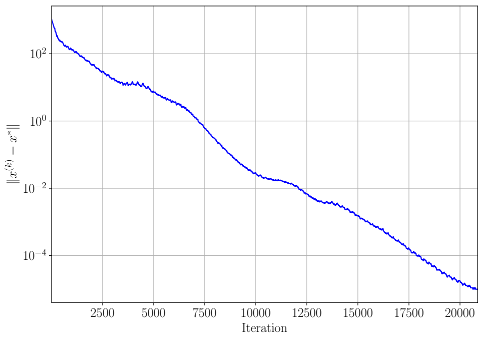
Figure 5.17: Results for a block-parallel implementation of Randomized Kaczmarz method using 8 threads for several overdetermined systems using a fixed number of columns and a varying number of rows.

5.5.2 Results for the Simple Randomized Kaczmarz Without Replacement Method

In this section, we changed the row selection criterion to sample without replacement. As previously discussed in Section 5.2.2 there are two ways to sample without replacement that threads are not working on the same row (Figure 5.6). However, these variations do not always converge. Furthermore, it can converge using the first sampling scheme from Figure 5.6 and not the second (or vice versa). An example for which the method converges for one but not another sampling scheme is shown in Figure 5.18, where we show the error evolution every 10 iterations. Note that, in the first sampling scheme, shown in Figure 5.18a, although the error initially decreases, after it reaches a global minimum, it increases again and the estimate of the solution gets progressively further away from the solution of the system. For the second sampling scheme, shown in Figure 5.18b, the error gets closer to zero when the number of iterations increases. In conclusion, this parallel implementation should not be used. Not only is it slower than the sequential RK method, as shown in the previous section, but also, in the case of sampling without replacement, the parallel implementation might not even converge.



(a) Fist sampling scheme from Figure 5.6.



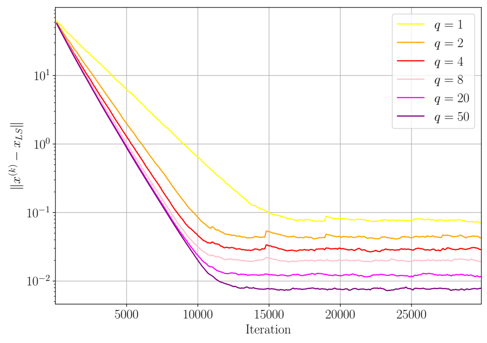
(b) Second sampling scheme from Figure 5.6.

Figure 5.18: Error evolution for a block-parallel implementation of Simple Randomized Kaczmarz Without Replacement method using 4 threads for a system 2000×500. The data represented here was obtained for a single run of the algorithm.

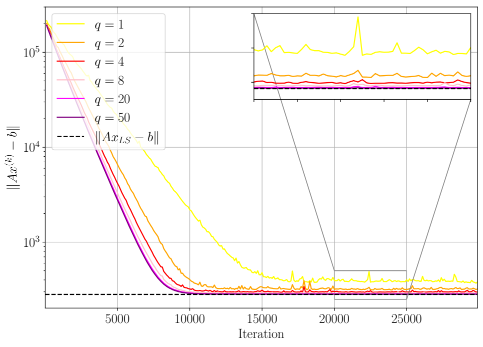
5.6 Application of RKA and RKAB to inconsistent systems

In Sections 5.2 and 5.3 we discussed the parallel implementations of RKA and RKAB using shared memory. We concluded that, in general, neither RKA nor RKAB can consistently beat the sequential RK in terms of execution time. However, RKA is able to decrease the convergence horizon for inconsistent systems when more than one thread is used, something that is not possible for RK (Section 3.5). In this section, we show that RKAB is also able to achieve this, which is relevant since the parallelization of RKAB is more effective than that of RKA.

To show how the convergence horizon can change for several numbers of threads we will show the evolution of the norm of the error, $\|x^{(k)} - x_{LS}\|$, and the norm of the residual $\|Ax^{(k)} - b\|$. To obtain these variables, we ran the RKA and RKAB algorithms with a maximum number of iterations and stored the error and norm every $step$ iterations. Since we will test numbers of threads between 1 and 50, that are not supported by the cluster, and since the goal is not to measure execution time but only to get the error and the residual, we used the sequential implementations of RKA and RKAB for the simulations in



(a) Error.



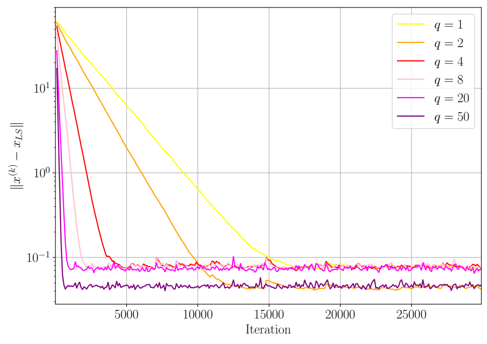
(b) Residual.

Figure 5.19: Results for RKA (with $\alpha = 1$) for a system 80000×1000. Here we show the first 30000 iterations and the error and residual were stored every $step = 100$ iterations.

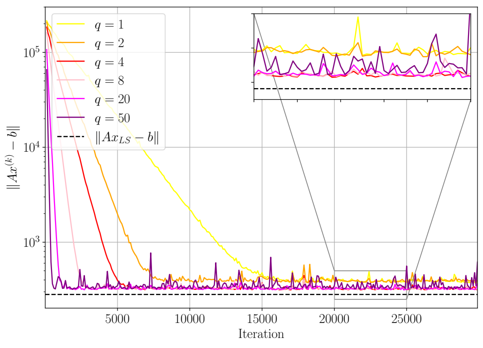
this Section. The inconsistent system used in this section has dimensions 80000×1000 and was taken from the least-squares data set that was generated to test REK and RGS (Section 4.2.3).

Figure 5.19 shows the error and residual evolution for the RKA algorithm using unitary row weights ($\alpha = 1$ in (5.1)). Figure 5.19a shows that using a higher number of threads, q, decreases the error value around which the method stabilizes. Figure 5.19b shows that the residual $q = 20$ and $q = 50$ stabilizes around the residual for the least-squares solution. However, this does not mean that the solution given using these numbers of threads is the least-squares solution since the error is not zero.

Figure 5.19 shows the error and residual evolution for the RKA algorithm using the optimal row weights for consistent systems ($\alpha = \alpha^*$ from (3.14)). Note that, although it is not expected for α^* to be optimal for these systems since they are inconsistent, we would like to analyze the impact of different α values in the convergence horizon. Figure 5.20a shows some big differences compared to Figure 5.19a. Note that increasing q can decrease the final error but this is not true for all values of q. Furthermore, it is not true that the error for q threads is smaller than that for $q - 1$ threads, something that was observed

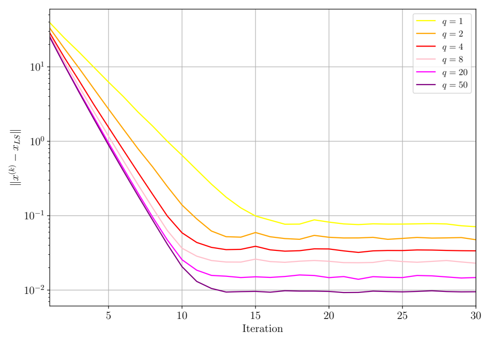


(a) Error.

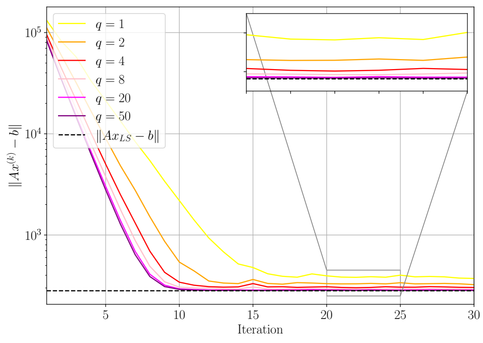


(b) Residual.

Figure 5.20: Results for RKA (with $\alpha = \alpha^*$) for a system 80000×1000. Here we show the first 30000 iterations and the error and residual were stored every $step = 100$ iterations.



(a) Error.



(b) Residual.

Figure 5.21: Results for RKA (with $\alpha = 1$) for a system 80000×1000. Here we show the first 30 iterations and the error and residual were stored every $step = 1$ iterations.

for $\alpha = 1$ in Figure 5.19a. Furthermore, the error values for $\alpha = 1$ can be smaller up to one order of magnitude than those for $\alpha = \alpha^*$. Figure 5.19b shows that the residual can decrease when using more than one thread but that decrease is smaller than the one for $\alpha = 1$. Although using $\alpha = \alpha^*$ does not decrease the convergence horizon as significantly as $\alpha = 1$, Figures 5.20a and 5.20b show that the error and residual stabilize much earlier than for $\alpha = 1$.

Figure 5.21 shows the error and residual evolution for the RKAB algorithm using block sizes equal to the number of columns of the system ($block\ size = 1000$) and unitary row weights ($\alpha = 1$). Figure 5.21 is very similar to Figure 5.19 in terms of the relationship between the error and residual for increasing numbers of threads, which shows that the RKAB method, like RKA, can decrease the convergence horizon. However, RKAB requires fewer iterations to converge since the work per iteration is much more significant than the work in one iteration of RKA.

In conclusion, for inconsistent systems, the RKA and RKAB methods can be used to decrease the convergence horizon. Furthermore, for RKA, unitary row weights ($\alpha = 1$) can decrease more the convergence horizon than using the optimal row weights for consistent systems ($\alpha = \alpha^*$). However, the optimal values for consistent systems $\alpha = \alpha^*$ can increase convergence, that is, the stabilization of the error and residual happen in fewer iterations than using $\alpha = 1$.

Chapter 6

Parallel Implementations Using Distributed Memory

In the previous chapter, we concluded that it is not possible to achieve an efficient block-sequential parallel implementation of RK using shared memory since none of our parallelization attempts can consistently beat the execution time of the sequential RK. However, in Section 5.4, we have shown that RKA and RKAB can be used to decrease the convergence horizon for inconsistent systems and that this decrease is proportional to the number of used threads.

In parallel implementations using shared memory, one is limited by the number of cores in a single machine. Distributed memory allows us to use more than one machine, and therefore, increase the number of cores available. With more cores, we can use RKA and RKAB to obtain solutions for inconsistent systems with smaller errors. Another advantage of using distributed memory is that we can process data sets that cannot be stored in a single machine. However, there are also disadvantages to using distributed memory in favor of shared memory. For example, when using distributed memory, data is transmitted between different machines, leading to high communication costs. For these reasons, it is also an option to combine both shared and distributed memory to optimize parallel implementations.

In this chapter, we implement and discuss the results for the RKA and RKAB methods using distributed memory only and the combination of distributed and shared memory. We decided not to implement and analyze the variations using sampling without replacement (SRKAWOR and SRKABWOR) since the results in the previous chapter proved them to have a similar performance to sampling rows based on their norms.

The organization of this chapter is as follows: in Section 6.1 we walk through the implementation of RKA for distributed memory only and for the combination of distributed and shared memory; Section 6.2 is identical to Section 6.1 but focusing on RKAB.

The implementations using distributed memory were accomplished using the C++ API MPI [1]. Similarly to the sequential versions and the parallel implementations for shared memory, experiments were carried out on the Accelerates Cluster; the consistent systems used during the simulation were taken from the first dense dataset discussed in Section 4.1.3; and execution times correspond to the total time of 10 runs of the algorithm.

[1] https://www.mpi-forum.org/

6.1 Distributed Memory Implementation of the Randomized Kaczmarz with Averaging Method

In this section, we discuss the implementation and results for RKA for distributed memory only and for both distributed and shared memory combined.

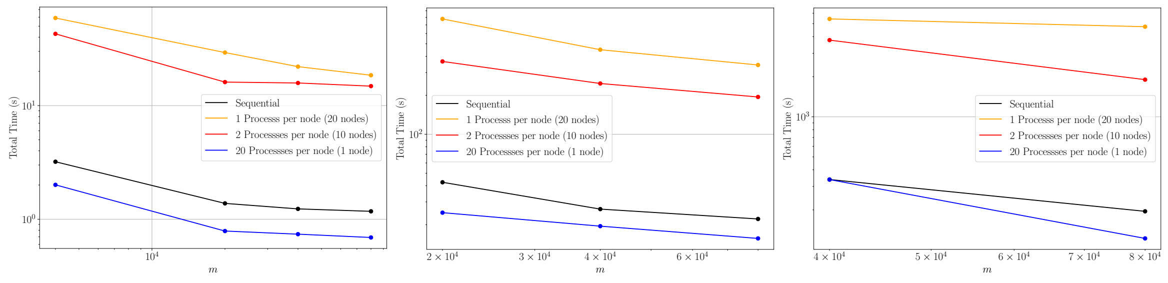
Since one of the advantages of using distributed memory is to be able to process data sets that are not able to be stored in a single machine, in this implementation we divide matrix A and vector b between the several available machines. This is similar to what we did in Section 5.2.1 by giving each thread a chunk of the system, where we concluded that there is not a major difference in iterations and time between partitioning the system or not. Like in Section 5.2.1, we developed a new sequential implementation of RKA that simulates the partition of the system amongst the several processes. [2] The implementation details of a single iteration of the parallel implementation of RKA for distributed memory are shown in Algorithm 4.

Algorithm 4 Pseudocode for an iteration of the parallel implementation of RKA.

```
1: $it \leftarrow it + 1$
2: $row \leftarrow$ sampled from $\mathcal{D}$
3: $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(prev)} \rangle}{\|A^{(row)}\|_2^2}$
4: for $i = 0, ..., N$ do
5:     $x_i \leftarrow \frac{x_i + scale \times A_i^{(row)}}{np}$
6: MPI Allreduce $(x, +)$
```

Note that Algorithm 4 is much simpler than the implementation of RKA for OpenMP (Algorithm 2). This is due to the fact that in distributed memory we do not need to worry about conflicts between processes reading and writing in the same memory position. This eliminates the need for the storage of the estimate of the solution from the previous iteration ($x^{(prev)}$ in Algorithm 2). The averaging of the results was accomplished with the *Allreduce* command with the sum operation. In OpenMP we had to update the solution sequentially using a critical section. The dependency of the communication time on the number of processes in MPI is different than the relationship of the communication time with the number of threads in OpenMP. In OpenMP the averaging of the results takes $O(q)$ time, where q is the number of threads, while here it takes $O(log(np))$ since the MPI *Allreduce* operation is implemented

[2]The sequential implementation of RKA for distributed memory can be described by Algorithm 9 with the exception of line 5. q will represent the number of processes instead of the number of threads.



(a) System with $n = 1000$. **(b)** System with $n = 4000$. **(c)** System with $n = 10000$.

Figure 6.1: Execution time for RKA using 20 MPI tasks for several overdetermined systems using a fixed number of columns and a varying number of rows. Row weights, α, were chosen as the optimal values given by (3.14).

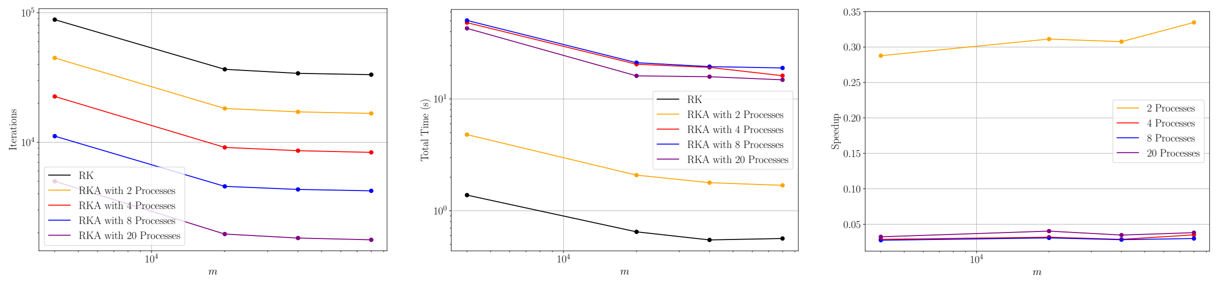
with a hypercube topology. However, communication time has other dependencies. In MPI, the time required to send messages between processors depends on the bandwidth, that is, the amount of data that can be transmitted over the network in one unit of time. In OPENMP, communication time depends on how fast threads can access memory. This means that, although communication in MPI has a smaller dependency on the number of processes, it does not necessarily mean that communication is faster.

The stopping criterion was chosen to be the same used for the shared memory implementation of RKA: we measure the necessary number of iterations to reach convergence by running the sequential and parallel algorithms until $\|x^{(k)} - x^*\| < 10^{-5}$ is verified; execution time is measured by rerunning the algorithm for the previously computed number of iterations.

Since a single node of the cluster has 2 central processing units with 10 cores each, we experimented with 3 configurations for distributed memory. In the first option, we have one MPI process per node. In the second, we have two MPI processes per node. Lastly, in the third option, we utilize all 20 cores in each node and have one process per core.

In Figure 6.1 we show the results for 20 MPI processes for systems with different dimensions. It is clear that, regardless of the dimension of the system, the last configuration has the smallest execution times. This is due to the communication cost being smaller between cores of the same node than between cores of different nodes. Since one of the goals of using MPI is to use several machines to process large data sets, we will further analyze the results for the second configuration since it has processes distributed amongst machines and is faster than the first configuration.

We now discuss the results of the RKA method by comparing it with the sequential RK present in Figure 6.2.



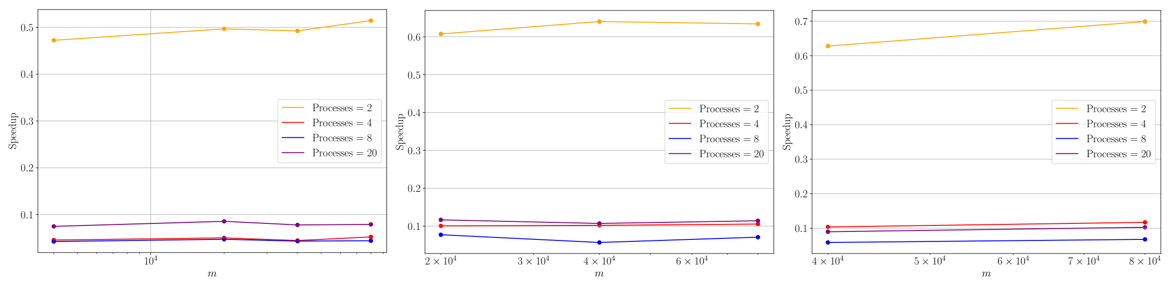
(a) Iterations. **(b)** Execution time. **(c)** Speedup.

Figure 6.2: Results for RK and RKA using 2, 4, 8, and 20 MPI tasks for several overdetermined systems with $n = 1000$ with a varying number of rows. Row weights, α, were chosen as the optimal values given by (3.14).

Figure 6.2a shows similar results to the ones using shared memory (Figure 5.2b): increasing the number of MPI processes decreases the number of iterations. However, the results for execution time in Figure 6.2b do not follow the same tendency as the number of iterations. Although iterations decrease for larger numbers of processes, time increases, meaning that the communication cost has a dominant factor in the execution time. As a consequence, the speedups in Figure 6.2c, computed as the quotient between the execution time of RK and the execution time of the parallel implementation of RKA, are very small, even smaller than the ones obtained for shared memory, present in Figure 5.3b.

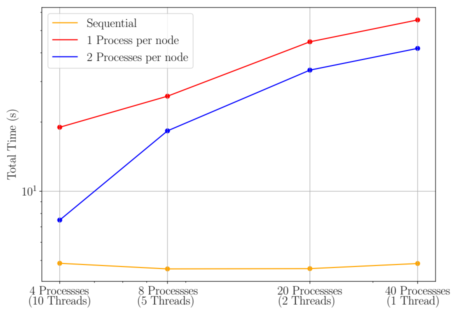
To conclude the analyses of the parallelization of RKA using distributed memory only, we calculated the speedup as the quotient between the total execution time of the sequential and parallel implementations of RKA. The results, shown in Figure 6.3, show that speedups are far from ideal. Furthermore, the speedups for the parallel implementation using shared memory only, in Figure 5.4, have higher values. In conclusion, using distributed memory only, RKA cannot be efficiently implemented.

To benefit from the advantages of both shared and distributed memory we developed an implementation of RKA using MPI and OpenMP. The pseudocode with the implementation details is presented in Algorithm 11 in Appendix F and was developed by putting together Algorithms 2 and 4 that contain the

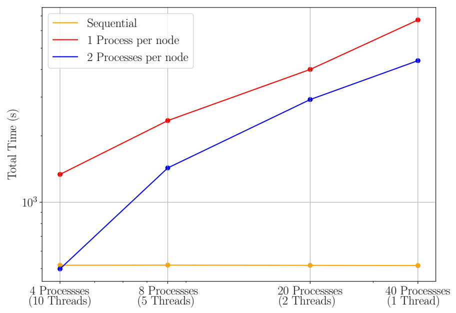


(a) Systems with $n = 1000$. **(b)** Systems with $n = 4000$. **(c)** Systems with $n = 10000$.

Figure 6.3: Speedup for RKA using 2, 4, 8, and 20 MPI tasks for several overdetermined systems using a fixed number of columns and a varying number of rows. Row weights, α, were chosen as the optimal values given by (3.14).



(a) System with dimensions 4000×1000.



(b) System with dimensions 40000×10000.

Figure 6.4: Execution time for a total of 40 MPI + OPENMP tasks for two different systems. Row weights, α, were chosen as the optimal values given by (3.14).

implementation of RKA for shared and distributed memory only.

We now analyze the results for 40 total tasks using different combinations of processes and threads. Figure 6.4 shows the results for two systems using one or two processes per node. Note that, as expected, two processes per node exhibit lower execution times than one process per node since the communication time is lower inside the same node. Furthermore, note that we only show results for the maximum of 10 threads since each CPU in the node has 10 cores. It is also expected that time decreases when we use fewer processes and more threads, once more due to the communication time. However, regardless of the dimension of the system, the parallel implementation using both distributed and shared memory is still slower than the sequential version of the RKA algorithm, leading us to conclude that this algorithm cannot be efficiently parallelized using a combination of distributed and shared memory.

6.2 Distributed Memory Implementation of the Randomized Kaczmarz with Averaging with Blocks Method

In this section, we discuss the implementation and results for RKAB for distributed memory only and for both distributed and shared memory combined. Similarly to the implementation of RKA (Section 6.1), the system will be partitioned between the several processes, meaning that the sequential implementation of RKAB for distributed memory can still be described by Algorithm 10 with the exception of lines 8 and 12 where the rows are sampled. Regarding the parallel implementation of RKAB for distributed memory, the details of a single iteration of RKAB for distributed memory are shown in Algorithm 5.

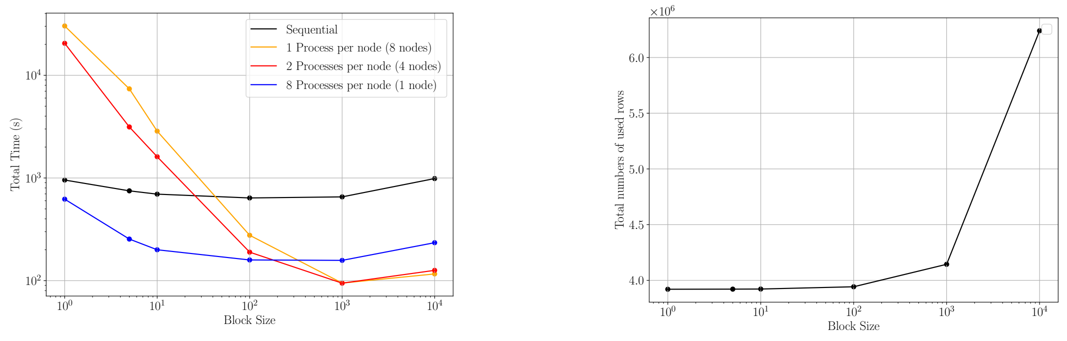
Algorithm 5 is much simpler than the parallel implementation for RKAB using shared memory (Algorithm 3) since we do not need the extra variable $x^{(thread)}$.

During the analyses of RKAB in Section 5.3, we concluded that a good option for the parameter $block\ size$ was the number of columns, n. However, since in this distributed memory implementation the matrix is partitioned amongst processes, this conclusion might not hold. For that reason, we will analyze the behavior of RKAB for several values of $block\ size$.

Algorithm 5 Pseudocode for an iteration of the parallel implementation of RKAB.

```
1:  $it \leftarrow it + 1$
2:  for $b = 0, ..., block\ size - 1$ do
3:      $row \leftarrow$ sampled from $\mathcal{D}$
4:      $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x \rangle}{\|A^{(row)}\|_2^2}$
5:      for $i = 0, ..., N$ do
6:          $x_i \leftarrow x_i + scale \times A_i^{(row)}$
7:  $row \leftarrow$ sampled from $\mathcal{D}$
8:  $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x \rangle}{\|A^{(row)}\|_2^2}$
9:  for $i = 0, ..., N$ do
10:     $x_i \leftarrow \frac{x_i + scale \times A_i^{(row)}}{np}$
11: MPI Allreduce $(x, +)$
```

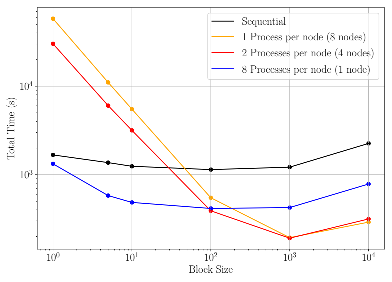
The stopping criterion is the same as the one chosen for RKA. The value of the row weights was chosen as $\alpha = 1$ (see Section 5.3.1). Furthermore, we tested the same 3 configurations also used for RKA. The results for 8 MPI processes for a system 80000×10000 for several values of $block\ size$ are shown in Figure 6.5. Firstly, contrary to the results for 8 threads using OpenMP (Figure 5.11b), time does not monotonically decrease for increasing block sizes up to the number of columns. Note that by partitioning the matrix by processes and using 8 processes, each will have a subsystem with dimensions 10000×10000. For $block\ size = 10000$, since rows are chosen accordingly to their norms, it is likely that some rows will be chosen more than once and that others will not be chosen at all. Using block sizes that are equal to or larger than the number of rows in each process can lead to reusing information and, consequently, decrease the rate of convergence. This can also be observed in Figure 6.5b where we see that the number of total used rows has a larger increase from $block\ size = 1000$ to $block\ size = 10000$. This means that the increase in the total amount of work needed for the algorithm to converge when



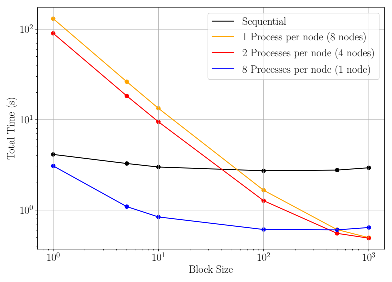
(a) Exection time.

(b) Total number of used rows.

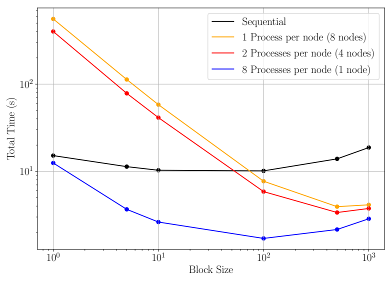
Figure 6.5: Results for RKAB using 8 MPI processes for a system with dimensions 80000×10000. We also represent the execution time for the sequential RKAB.



(a) Exection time for a system with dimensions 40000×10000.



(b) Exection time for a system with dimensions 80000×1000.



(c) Exection time for a system with dimensions 4000×1000.

Figure 6.6: Results for RKAB using 8 MPI processes for three systems. We also represent the execution time for the sequential RKAB.

using larger block sizes has a larger impact on execution time than the decrease in communication. As for the relationship between execution times for the three configurations, it is not always true that having all the processes in one node is faster than distributing processes among nodes, something that was observed for RKA. When the block size increases, the amount of communication decreases and so does the execution time. However, for large block sizes, communication between processes in the same node is slower than communication between processes in different nodes. This might mean that, for large block sizes, cache phenomenons have a dominant impact on communication. Nonetheless, just like for RKA, generally using two processes per node is faster than using only one.

Figure 6.6 shows the execution time for 8 processes for systems with other dimensions. Note that the system in Figure 6.6a has the same number of columns as the system in Figure 6.5a and both show an optimal block size of 1000. However, although the systems in Figures 6.6b and 6.6c both have $n = 1000$, it is not true that they have the same optimal parameter for $block\ size$. When we partition the systems in Figure 6.6b and 6.6c among processes, each will have a submatrix with dimensions 10000×1000 and 500×1000 respectively. Note that 10000×1000 is an overdetermined system that benefits from a larger block size, contrary to 500×1000 which is an underdetermined system. This leads us to conclude that the optimal block size for the implementation of RKAB for distributed memory not only depends on the number of columns but also on the relationship between the number of rows and columns of the submatrices that are owned by each process.

Apart from the distributed memory implementation of RKAB, we also developed an implementation using both MPI and OpenMP. The pseudocode with the implementation details, presented in Algorithm 12 in Appendix F, was developed by combining the shared and distributed implementations in Algorithms 3 and 5. Figure 6.7 shows the results for 40 tasks for a system 80000×10000 using one or two processes per node where the block size was chosen to be 1000 (see Figure 6.5a). Note that there is an unexpectedly large difference between the two configurations and that, contrary to what was expected, time does not necessarily decrease using more threads and fewer processes. However, the

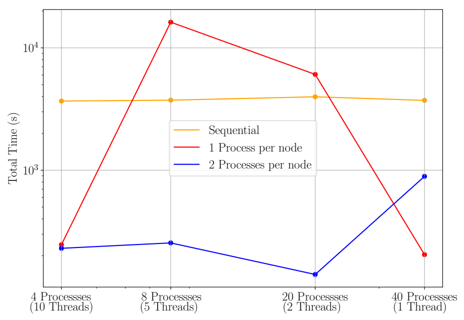


Figure 6.7: Execution time for a total of 40 MPI + OPENMP tasks for two different systems.

chosen block size is only optimal for 8 processes. Here we are using 40 total tasks where just one configuration uses 8 processes. A more thorough analysis is necessary to determine an effective way to choose the block size for each combination of processes and threads so that we can better analyze the performance for the RKAB method for distributed memory.

Chapter 7

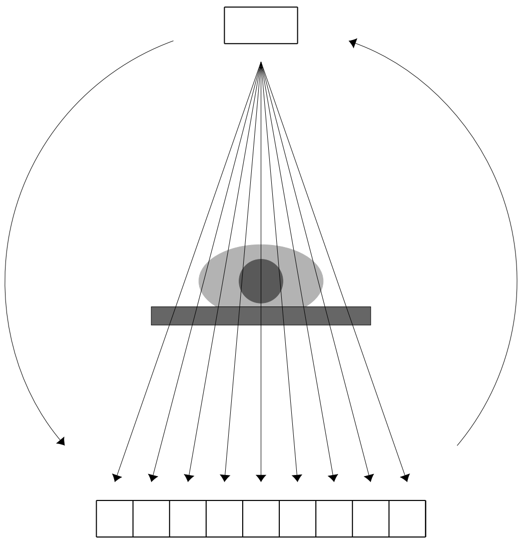
Application of the Randomized Kaczmarz Method to Computed Tomography

In Chapter 4 we analyzed how the RK method and its variations behave when solving artificial systems. However, we have yet to apply RK to a real-world problem. One of the many applications of RK is image reconstruction using projection data originating from Computed Tomography (CT). When experimental data is not noisy, linear systems derived from CT problems are consistent. However, most real-world problems involve physical quantities that are measured with some error, meaning that the systems that describe them are inconsistent.

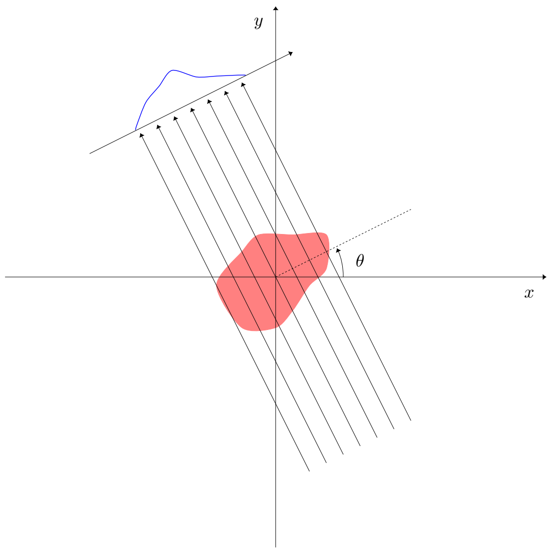
In this chapter, we discuss how we can use some of the variations of the Kaczmarz method to solve CT problems. We start by describing how to transform CT data into a linear system of equations in Section 7.1. Section 7.2 contains the details of generating CT data using simulators. In Section 7.3 we show how to use RK based methods to solve these systems. Furthermore, since CT problems have some additional restrictions for the solution, we discuss how we can integrate the Kaczmarz method with constraints that are described with linear inequalities. We finish that section by showing that RKA and RKAB can be used to reduce the reconstruction error of CT images. Note that in the presence of inconsistent systems, we cannot employ the stopping criteria used for the sequential version. For this reason, in Section 7.4, we discuss different stopping criteria with the goal of obtaining solutions that minimize the reconstruction error.

7.1 From projection data to linear systems

The geometry of a 2D CT scan, represented in Figure 7.1a, comprises an X-ray tube (radiation source), and, on the opposite side, a panel of detectors (target). The source and the detectors are spun around the area that is being scanned. For each position of the source and target, the radiation emitted by the X-ray tube is measured by the detectors. After collecting measurements from several positions, the data is combined and processed to create a cross-sectional X-ray image of the scanned object. So how does one translate measurements of radiation onto an image? We should start by explaining the physical



(a) Representation of the radiation source and detector panel in a CT imaging system.



(b) Spatial domain of a CT scan. The body that is being scanned is represented in red and the projections of each ray are represented in blue. θ is the rotation angle of the source.

Figure 7.1: Geometry of a CT scan.

meaning of a radiation measurement. Figure 7.1b represents the spatial domain of a CT scan.

Let $\mu(r)$ be the attenuation coefficient of the object along a given ray. The damping of that ray through the object is the line integral of $\mu(r)$ along the path of the ray, that is:

$$b = \int_{ray} \mu(r) dr \; . \tag{7.1}$$

This is the projection data that is obtained by each detector in the panel. In a CT problem the goal is, using the projection data b, to find $\mu(r)$ since a reconstructed image of the scanned body can be obtained using the attenuation coefficient of that object. To obtain $\mu(r)$ from b we make the following assumptions. First, we assume that the reconstructed image has P pixels on each side, meaning that it is a square of L^2 pixels. Secondly, we assume that $\mu(r)$ is constant in each pixel of the image. We can then denote the solution of the CT problem as a vector x with length L^2, that contains the attenuation coefficient of the scanned body in every pixel of the image. With these assumptions, we approximate the integral in (7.1) with a finite sum, such that

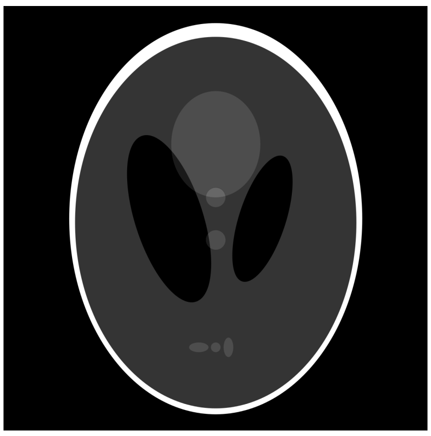
$$b = \sum_{l=1}^{L^2} a_l x_l \; , \tag{7.2}$$

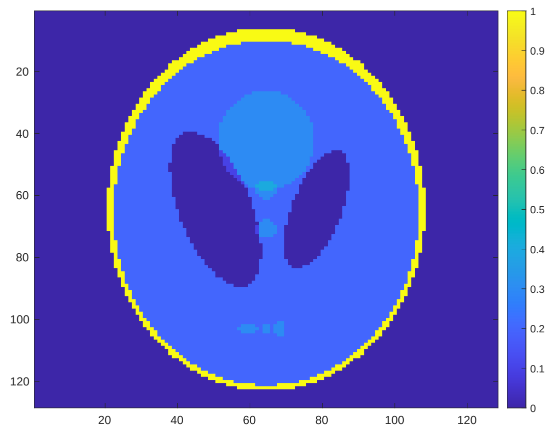
where a_l is the length of the ray in pixel l. Note that a_l is 0 for pixels that are not intersected by the ray. (7.2) corresponds to an approximation of the radiation intensity measured by a single detector. For a given position of source and target, d projections are obtained, where d is the number of detectors in the panel. Considering that measurements are obtained for Θ angles of rotation, the experimental data is made up of $d \times \Theta$ projections. In summary, there are $d \times \Theta$ equations like (7.2). With this setup, we are finally ready to write the CT problem as a linear system.

The system's matrix, $A^{m \times n}$, has dimensions $m = d \times \Theta$ and $n = L^2$. Each row describes how a given ray intersects the image pixels. Note that we are considering that there is one ray per detector. Calculating the entries of matrix A is not trivial and it can be a very time-consuming process. A vastly used method for computing A is the Siddon method [30]. Since matrix entries are only non-zero for the pixels that are intersected by a given ray, the system's matrices are very sparse, meaning that we will use the CSR during simulations. Regarding the other components of the system, the solution x has the pixel's values, and the constants vector b has the projection data.

7.2 Generation of CT data

The generation of linear systems from CT problems was achieved using the AIR TOOLS II software package for MATLAB [1] with a parallel beam geometry. To simulate projection data, we used the Shepp–Logan phantom [31], a standard test image for image reconstruction algorithms, represented in Figure 7.2a.


(a) Original spirit.



(b) Sampled phantom using 128 pixels for height and width.

Figure 7.2: Shepp–Logan phantom.

Throughout this chapter, we will discuss the two algorithms developed by Lith, Hansen, and Hochstenbach [32]. For this reason, we will use the same parameters that the authors used to generate their data set. The number of pixels was then selected to be $L = 128$ and the projection angles vary between 0º and 178.5º in increments of 1.5º, resulting in the sampled image in Figure 7.2b. The systems generated by the AIR TOOLS II package are consistent. To simulate measurement errors, the authors used a Gaussian distribution. However, since projection data comes from a photon counting process, using a Poisson distribution [33] to simulate errors in CT data is a more realistic approach. Therefore, we used

[1] `http://people.compute.dtu.dk/pcha/AIRtoolsII/`

both Gaussian and Poisson distributions so that we can validate the results for different error distributions. We will now describe the process of generating errors and transforming the consistent systems generated by the AIR TOOLS II package into inconsistent systems.

Let $\bar{b}$ be the error-free vector of constants and $\bar{x}$ be the respective solution. The consistent system can then be written as $A\bar{x} = \bar{b}$. We now introduce error e into the projections, such that $b = \bar{b} + e$.

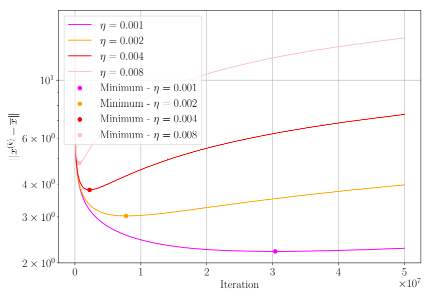
To add white Gaussian noise [32] we use a normal distribution with zero mean, that is, $e \sim N(0, \sigma)$. The standard deviation is chosen accordingly to the desired relative noise level, η, that can be computed using (7.3). The relative noise level was chosen as $\eta = \{0.001, 0.002, 0.004, 0.008\}$.

$$\eta^2 = \frac{\mathbb{E}(\|e\|^2)}{\|\bar{b}\|^2} = \frac{m\sigma^2}{\|\bar{b}\|^2} \Longrightarrow \sigma = \frac{\eta\|\bar{b}\|}{\sqrt{m}} \, . \tag{7.3}$$

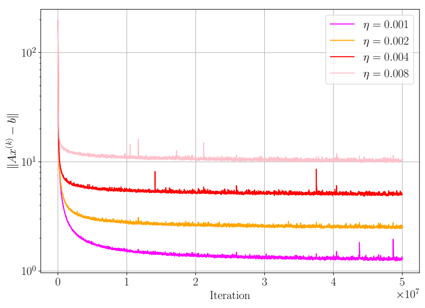
To generate b accordingly to a Poisson distribution, we follow the procedure used in Reference [34]. The error-free vector of constants $\bar{b}$ is used to compute the X-ray intensities at the detectors such that $\bar{I}_i = I_0 \, \exp -\bar{b}_i$, with $i = 1, .., m$, and where I_0 is the source's intensity. The noisy intensities are given by $I_i = \mathcal{P}(\bar{I}_i)$, where $\mathcal{P}(\lambda)$ is a Poisson distribution with expected value λ. The new vector of constants b can now be obtained by converting the intensities to noisy projection data such that $b_i = -log(I_i/I_0)$. Just like for Gaussian generated error where we had several values for η, for Poisson generated error we also chose several values for the sources intensity such that $I_0 = \{10^{13}, 4 \times 10^{13}, 8 \times 10^{13}, 10^{14}\}$. Having generated the error vector, e, the resulting inconsistent system can then be written as $A\bar{x}+e = Ax_{LS} \approx b$. Note that our goal is not to find the least-squares solution x_{LS}, but to find the solution of the error-free case $\bar{x}$.

From a physical perspective, it does not make sense to have negative values for the projection data. However, this can happen when adding noise. To avoid it, for a given entry in the projection data, when sampling from the Poisson and Gaussian distributions, if $b_i < 0$, we resample the value until the projection value is non-negative.

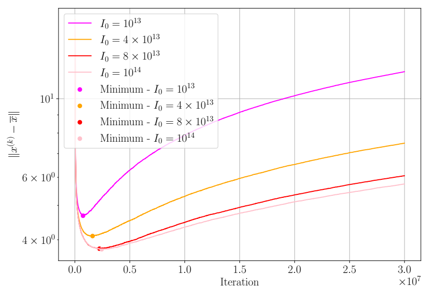
To use the Kaczmarz algorithm in these systems a few more modifications have to be made. First, rows that have all entries set to zero should be deleted, together with the corresponding entry in the vector of constants. Second, there can be null projection values such that $b_i = 0$. This can happen if a ray only intersects pixels where the attenuation coefficient is zero, that is, pixels that do not contain the scanned body. In this case, the entries of the solution corresponding to the non-zero entries of the $A^{(i)}$ should be set to zero and those columns should be deleted from the system. Furthermore, row for which $b_i = 0$ should be deleted. With these procedures, we generated 4 linear systems with noise sampled from a Poisson distribution and another 4 with noise sampled from a Gaussian distribution. All these systems have the same dimension, 19558×16384.



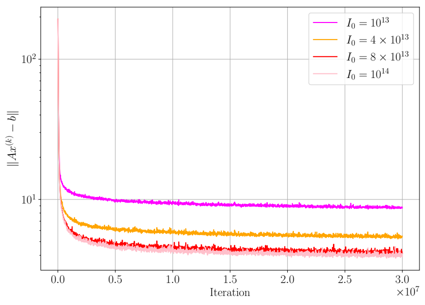
(a) Evolution of the reconstruction error for systems with Gaussian generated noise.



(b) Evolution of the residual for systems with Gaussian generated noise.



(c) Evolution of the reconstruction error for systems with Poisson generated noise.



(d) Evolution of the residual for systems with Poisson generated noise.

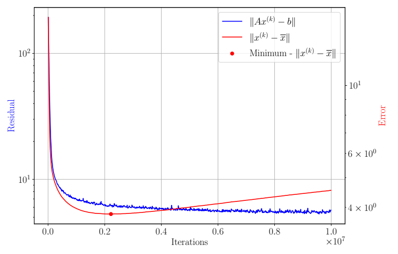
Figure 7.3: Results for linear systems originated from CT problems. Matrix A has dimensions 19558×16384. The values for the error and residual were sampled every 10000 iterations.

7.3 Solving Inconsistent System with RK Based Methods

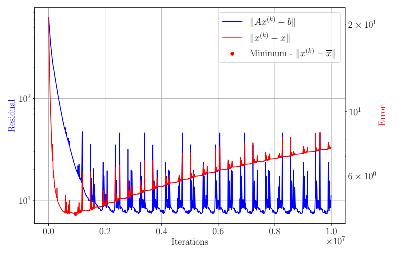
In this section, we describe the behavior of RK based methods for problems derived from Computed Tomography. It has been proved [32] that the cyclical version of the Kaczmarz method (CK), can outperform the Randomized version (RK). Furthermore, in Chapter 4, we showed that sampling without replacement (SRKWOR method) can outperform both CK and RK. Therefore, we will analyze how these three methods can be used to solve CT problems. We start by discussing the impact of parameters η and I_0 that control the noise level for Gaussian and Poisson-generated errors by analyzing the results for RK.

Figure 7.3 shows the evolution of the residual and reconstruction error, $\|x^{(k)} - \overline{x}\|$, for the RK method. Note that, in Figures 7.3a and 7.3c, the reconstruction error decreases until a minimum is reached (marked with a circle), after which it starts increasing. This is expected since the estimate of the solution given by RK will try to minimize the residual, $Ax^{(k)} - b$, which is obtained by the least-squares solution, not by the error-free solution, $\overline{x}$. However, to obtain a solution as close as possible to $\overline{x}$, we need to find the solution estimate at the minimum of the reconstruction error. Figures 7.3b and 7.3d show that the residual decreases and stabilizes around the convergence horizon.

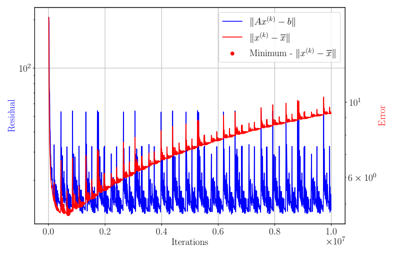
Figure 7.3a shows that, for Gaussian-generated noise, systems with larger relative noise (η) have a larger value for the minimum reconstruction error. However, that minimum is reached in fewer iterations when compared to systems with larger η. Furthermore, systems with a larger relative noise also have a



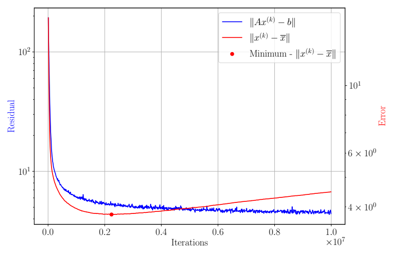
(a) Evolution of the residual and reconstruction error for a system with Gaussian generated noise ($\eta = 0.004$) using the RK method.



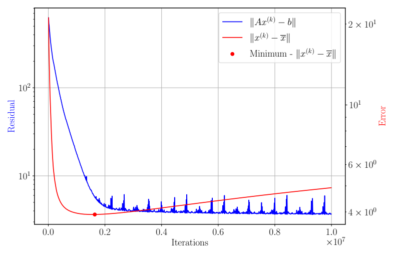
(b) Evolution of the residual and reconstruction error for a system with Gaussian generated noise ($\eta = 0.004$) using the CK method.



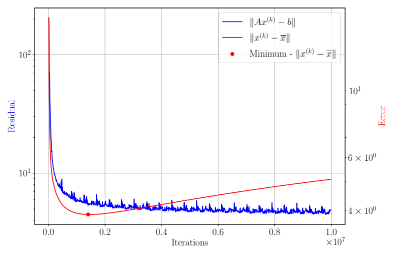
(c) Evolution of the residual and reconstruction error for a system with Gaussian generated noise ($\eta = 0.004$) using the SRKWOR method.



(d) Evolution of the residual and reconstruction error for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the RK method.



(e) Evolution of the residual and reconstruction error for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the CK method.



(f) Evolution of the residual and reconstruction error for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the SRKWOR method.

Figure 7.4: Results for two linear systems originated from CT problems. Matrix A has dimensions 19558×16384. The values for the error and residual were sampled every 10000 iterations.

larger value for the convergence horizon, as seen in Figure 7.3b.

Figures 7.3c and 7.3d show that systems with smaller I_0 are noisier than systems with larger I_0. For a Poisson distribution $\mathcal{P}(\lambda)$ with expected value λ, smaller values of I_0 will lead to smaller values of I_i. Consequently, since $-\log(x)$ is a decrescent function, smaller values of I_0 will lead to larger errors. Figure 7.3c shows that, for Poisson-generated data, although the value of the minimum reconstruction error is larger for more noisy systems, the number of iterations needed to reach that minimum is not as distinct between systems with different I_0 as it is for systems with different η (Figure 7.3a).

We now focus on two systems, one for each type of noise, and compare the RK method with the CK and SRKWOR method. We chose, for the Gaussian distribution, the system with $\eta = 0.004$ and, for the Poisson distribution, the system with $I_0 = 8 \times 10^{13}$ since they have similar values of the minimum of the reconstruction error and that minimum is obtained with a similar number of iterations. The evolution for the residual and reconstruction error is shown in Figure 7.4 and the values of the minimum of the reconstruction error are present in Table 7.1.

From the results in Figure 7.4 we can conclude that CK and SRKWOR have the same behavior as RK for inconsistent systems, that is, the reconstruction error exhibits a minimum value and the residual decreases until it stabilizes in the convergence horizon. Regarding the relative performance between the methods, Table 7.1 shows that RK is the method for which the reconstruction value has the smaller

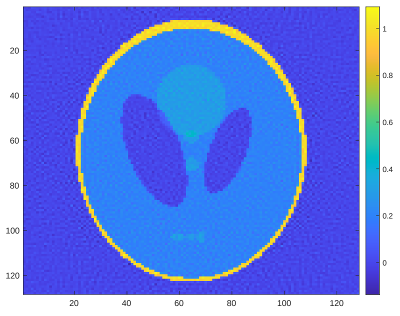
Table 7.1: Value of the minimum of the reconstruction error and respective iterations for the RK, CK and SRKWOR method for systems perturbed with noise sampled from Poisson and Gaussian distributions.

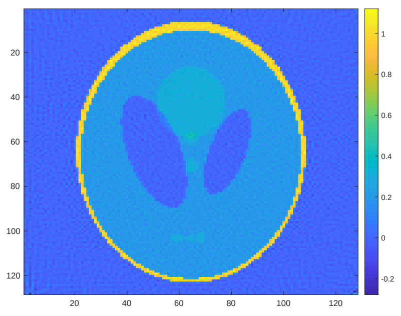
Method	Gaussian		Poisson	
	Iterations	Values	Iteration	Value
RK	2220000	3.80	2240000	3.77
CK	960000	4.45	1640000	3.91
SRKWOR	700000	4.67	1400000	3.88

value for both types of addictive noise. However, the other two methods require fewer iterations to reach their minimum error. For the Poisson generated noise, the values of the minimum error are quite similar. Is it worth it to use more iterations and have a smaller error value? Or is the error difference so small that it is not noticeable in the reconstructed images?

Figure 7.5 shows the reconstructed images for the three methods using Gaussian noise. Note that there is not a noticeable difference between the CK and SRKWOR methods (Figures 7.5b and 7.5c). However, the reconstructed image using the RK method is more similar to the original (Figure 7.2b) than the two previously discussed - the image scale for RK does not admit such low values has those for CK and SRKWOR. Using the image scale we also notice an important detail: although the pixels in the original image only admit values between 0 and 1, the solutions obtained with RK based methods have values outside that range. Furthermore, there are negative pixels, which do not make sense from a physical perspective. This is due to the fact that, just because the solution of the consistent problem does not have negative pixels, that does not mean that, after adding noise to the vector of constants, the least-squares solution of the inconsistent problem will have only positive values. To solve this problem we will introduce the restriction of no negative pixels into the problem. The reconstructed images using Poisson noise are very similar between the three methods so we will not show them here. This was expected since the difference between the minimum values of the reconstruction error is very small.

Requiring that the reconstruction image does not have negative pixels is equivalent to stating that


(a) Solution given by the RK method at iteration 2220000.


(b) Solution given by the CK method at iteration 960000.

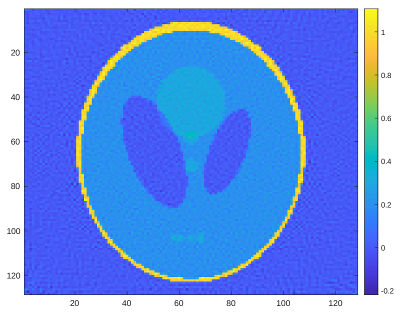

(c) Solution given by the SRKWOR method at iteration 700000.

Figure 7.5: Reconstructed images using the solution at the minimum of the reconstruction error for a system 19558×16384 perturbed with Gaussian generated noise with parameter $\eta = 0.004$.

$x \geq 0$. Our new goal is to solve the following problem:

$$Ax = b \quad \text{subject to} \quad x \geq 0\,. \tag{7.4}$$

This problem can be rewritten has

$$\begin{bmatrix} A \\ I \end{bmatrix} \quad x \quad \begin{matrix} = \\ \geq \end{matrix} \begin{bmatrix} b \\ 0 \end{bmatrix}\,, \tag{7.5}$$

Meaning that we have a linear system where the first m equations are equalities and the last n equations are inequalities. There is a simple way to adapt the Kaczmarz method to solve systems with inequalities [12]. Suppose that, in a given iteration, the chosen row corresponds to an inequality. If that inequality is satisfied by the current estimate of the solution, we move on to the next iteration; if the previous condition is not verified we project the current estimate of the solution onto the equation defined by the inequality, just like a normal iteration of the Kaczmarz method for equalities.

Table 7.2: Value of the minimum of the reconstruction error and respective iterations for the RK, CK and SRKWOR method for systems perturbed with noise sampled from Poisson and Gaussian distributions using the restriction $x \geq 0$.

Method	Gaussian		Poisson	
	Iterations	Values	Iteration	Value
RK	4470000	3.09	5060000	3.08
CK	2230000	1.64	2230000	1.41
SRKWOR	1150000	2.11	2370000	1.51

We implemented this restriction into the CK, RK, and SRKWOR methods, and the obtained results of the minimum values of the reconstruction error are present in Table 7.2. Note that, regardless of the method, the minimum values for the reconstruction error are smaller with the $x \geq 0$ restriction than without it (Table 7.1). This is in agreement with intuition since we are adding restrictions to the system that we know are satisfied by the error-free solution, making the solutions obtained with the restriction closer to $\overline{x}$. However, the decrease in the minimum values is larger for some methods than others. It is clear that the decrease is not as significant for RK as it is for the other methods. This is expected since, by adding rows with unit norms, if the norms of the rows corresponding to the equalities are larger, the rows encoding the restriction are sampled less frequently. On the other hand, for the other two methods, since rows are chosen cyclically, rows corresponding to the restrictions will mandatorily be used. Between CK and SRKWOR, CK appears to have the smalest values for the reconstruction error.

From the methods discussed so far, using the restriction $x \geq 0$, CK is the method for which the minimum of the reconstruction error has the smallest values for both types of noise. In Section 5.6 we have shown that RKA and RKAB can decrease the convergence horizon for noisy systems. However, RKA and RKAB use row selection criteria based on row norms, which we have just saw is not beneficial

Table 7.3: Value of the minimum of the reconstruction error and respective iterations for CK, SRKAWOR and SRKABWOR for systems perturbed with noise sampled from Poisson and Gaussian distributions using the restriction $x \geq 0$.

		Gaussian		Poisson	
		Iteration	Value	Iteration	Value
CK		2230000	1.64	2230000	1.41
SRKAWOR	$q = 1$	1150000	2.11	2370000	1.50
	$q = 2$	2030000	1.67	2820000	1.39
	$q = 4$	2480000	1.50	3090000	1.36
	$q = 8$	2790000	1.44	3230000	1.35
SRKABWOR	$q = 1$	79	2.11	153	1.50
	$q = 2$	85	1.96	183	1.40
	$q = 4$	103	1.93	179	1.39
	$q = 8$	106	1.98	197	1.39

when using the restriction $x \geq 0$. On the other hand, the variations of RKA and RKAB using sampling without replacement, SRKAWOR and SRKABWOR use every row of the system. For these reasons, we will now investigate if SRKAWOR and SRKABWOR can decrease the minimum of the reconstruction error and compare the results with the overall best method discussed so far, the CK method. We will not discuss the results for the case without the $x \geq 0$ restriction, since we have concluded that using it is always beneficial. For SRKABWOR, we use $block\ size = 15000$ since we concluded in Section 5.3 that the size of the blocks should be similar to the number of columns. Since we are only analyzing the numerical results, we used the sequential implementations of SRKAWOR and SRKABWOR with $q = \{1, 2, 4, 8\}$ (Algorithms 9 and 10). Furthermore, row weights were chosen as $\alpha = 1$, an option that we concluded in Section 5.3 was better for inconsistent systems. The results for SRKAWOR and SRKABWOR are shown in Table 7.3.

Note that the results for $q = 1$ are identical to those for SRKWOR in Table 7.2 since SRKAWOR and SRKABWOR are reduced to the SRKWOR method using one thread. For SRKAWOR note that the minimum reconstruction error does decrease for increasing numbers of threads. However, similarly to what happened for the shared memory implementation results for consistent systems using $\alpha = 1$ (Figure 5.2a), that decrease gets smaller when q increases. This might mean that the chosen row weights are inadequate for larger numbers of threads. Still, the SRKAWOR method with $q = 8$ has a smaller minimum than using the CK method. The performance of SRKABWOR is worse than that of SRKAWOR since the decrease for the minimum is much smaller than that for SRKAWOR. Once more, changing parameters like the block size or α could improve the results but that requires a more in-depth analysis than the one we do here. To conclude this analysis note that, although SRKAWOR and SRKABWOR are able to decrease the minimum of the reconstruction error, the iteration for which that minimum is obtained increases with the number of threads, meaning that, even if the SRKAWOR and SRKABWOR could be parallelized, there is a trade-off between execution time and the minimum of the reconstruction error.

So far we have shown how several methods compare in regards to the analytical value of the minimum of the reconstruction error. However, in the real-world situation, we do not have access to the original image that is being reconstructed. We need a stopping criterion that terminates these methods as close as possible to the minimum of the reconstruction error.

7.4 Stopping Criteria for Inconsistent Systems

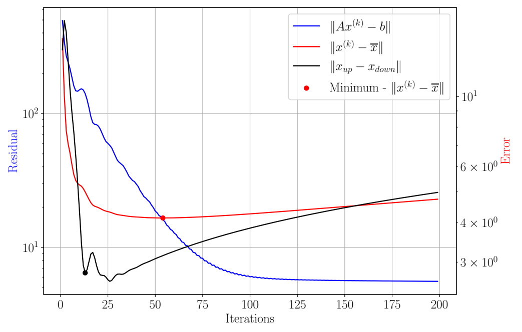
In this section, we tackle the problem of finding a stopping criterion that gives a solution that is close in iterations to the minimum of the reconstruction error. Furthermore, the stopping criterion should not be dependent on the statistical properties of the noise. One option would be to try to find where the value of the residual stabilizes. However this option is not viable for two reasons: the first is that the residual is computationally expensive and computing it many times would be counterproductive for the Kaczmarz algorithm which has a very small workload per iteration; the second is that the stabilization of the residual does not necessarily correspond to the minimum of the reconstruction error, shown by Figures 7.4b and 7.4e.

Lith, Hansen, and Hochstenbach [32] developed two methods to solve CT problems based on CK. Both methods use an error gauge that can be computed by using both a standard down-sweep Kaczmarz method (CK) and an up-sweep version. In an up-sweep Kaczmarz method, the system's matrix is still used ciclycally but rows are selected from the bottom to the top of the matrix. The stopping criterion is defined by testing if the error gauge has achieved a minimal value. Not only do these methods not require any information about the statistical properties of the noise, but they also do not require the computation of the residual.

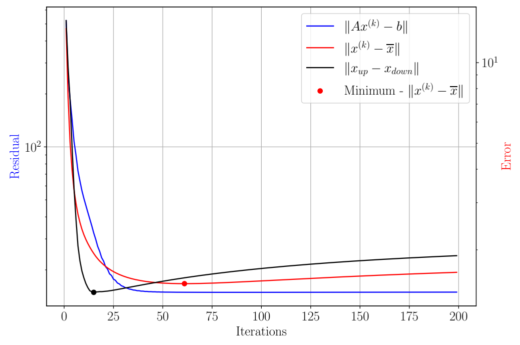
Let $\mathcal{K}^{\downarrow}(x)$ be a down-sweep Kaczmarz starting with x as the solution estimate and let $\mathcal{K}^{\uparrow}(\tilde{x})$ be the the analogous of $\mathcal{K}^{\downarrow}(x)$ for an up-sweep. The error gauge is given by $\|x - \tilde{x}\|$ and the Twin Algorithm can be described by Algorithm 6. In line 6 of Algorithm 6, the authors refer that finding a minimum can be done by checking if the values of the error gauge stopped decreasing for 7 consecutive iterations.

Algorithm 6 Twin Algorithm

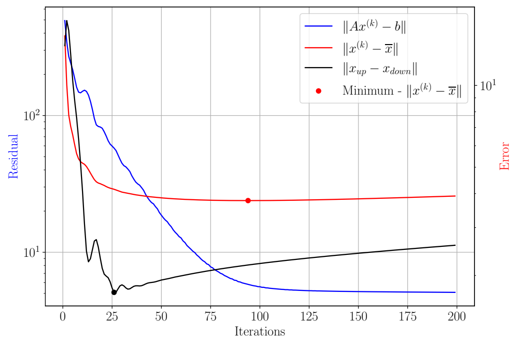
```
1: $x \leftarrow 0$
2: $\tilde{x} \leftarrow 0$
3: **for** $i = 0, ..., maxits$ **do**
4:     $x \leftarrow \mathcal{K}^{\downarrow}(x)$
5:     $\tilde{x} \leftarrow \mathcal{K}^{\uparrow}(\tilde{x})$
6:     **if** $\|x - \tilde{x}\|$ if at a minimum **then**
7:         **break**
8: **return** $\frac{1}{2}(x + \tilde{x})$
```



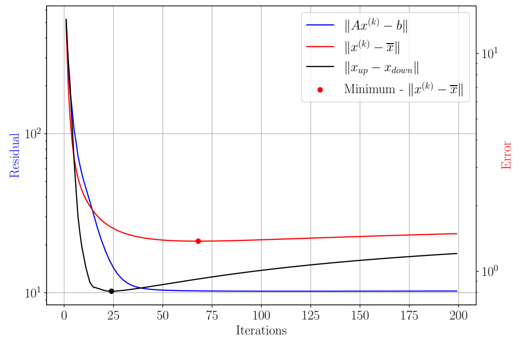
(a) Evolution of the residual, reconstruction error and error gauge for a system with Gaussian generated noise ($\eta = 0.004$) using the Twin Algorithm.



(b) Evolution of the residual, reconstruction error and error gauge for a system with Gaussian generated noise ($\eta = 0.004$) using the Twin Algorithm with the restriction $x \geq 0$.



(c) Evolution of the residual, reconstruction error and error gauge for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the Twin Algorithm.



(d) Evolution of the residual, reconstruction error and error gauge for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the Twin Algorithm with the restriction $x \geq 0$.

Figure 7.6: Results for two linear systems originated from CT problems. Matrix A has dimensions 19558×16384. The values for the error, residual, and error gauge are shown for every iteration.

We implemented two versions of Algorithm 6: with and without the restriction $x \geq 0$, just like for the Kaczmarz-based method discussed in the previous Section. The results are shown in Figure 7.6 where the black dot marks the stopping iteration for the Twin Algorithm, coincident with a minimum of the error gauge.

Note that, by comparing Figures 7.6a and 7.6b and Figures 7.6c and 7.6d, integrating the Twin Algorithm with the restriction $x \geq 0$ decreases the minimum of the reconstruction error. However, there seems to be a considerable difference in iterations between the iteration given by the stopping criterion of the Twin Algorithm and the minimum error value. Nonetheless, this difference in iterations might not translate into a large difference in the value of the reconstruction error. For a more rigorous analysis, we constructed a summary of the results for the Twin Algorithm in Table 7.4.

The first conclusion to draw from Table 7.4 is that the numerical values of the minimum of the reconstructor error are smaller for the Twin Algorithm than for CK, RK, and SRKWOR, whose results are in Tables 7.1 and 7.2 (with the exception of the value of RK for the Gaussian generated error without using

Table 7.4: Value of the minimum of the reconstruction error and respective iterations for the Twin Algorithm for systems perturbed with noise sampled from Poisson and Gaussian distributions using the restriction $x \geq 0$.

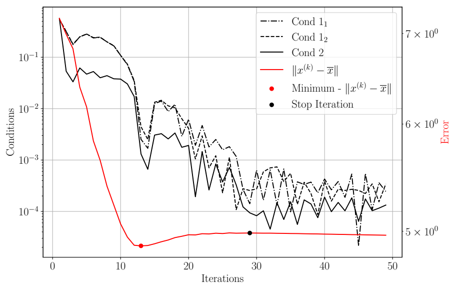
		w/o $x \geq 0$	w/ $x \geq 0$
Gaussian	Iteration (Minimum)	54	61
	Value (Minimum)	4.14	1.50
	Stop Iteration	13	15
	Stop Value	4.46	1.71
Poisson	Iteration (Minimum)	94	68
	Value (Minimum)	3.77	1.38
	Stop Iteration	26	24
	Stop Value	4.01	1.49

the $x \geq 0$ error). Furthermore, using the restriction $x \geq 0$ significantly decreases the reconstruction error. Finally, note that the difference for the reconstruction error at the stopping iteration and at the analytical minimum is not significant, especially using the restriction.

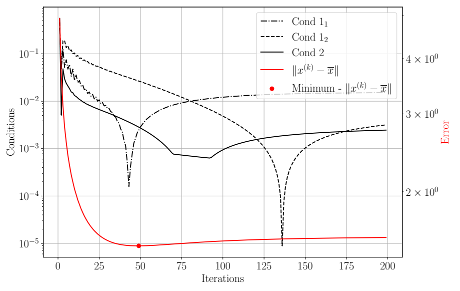
The second method developed in Reference [32], the Mutual-Step algorithm, uses the error gauge to determine approximately optimal step sizes for every iteration and eliminates the need for a stopping rule. The pseudocode is present in the Algorithm 7.

Algorithm 7 Mutual-Step Algorithm

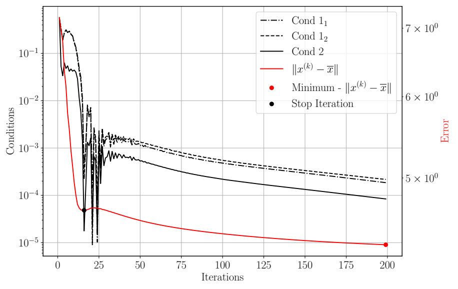
```
1:  $x_0 \leftarrow \mathcal{K}^{\downarrow}(0)$
2:  $\tilde{x}_0 \leftarrow \mathcal{K}^{\uparrow}(0)$
3:  $x \leftarrow x_0$
4:  $\tilde{x} \leftarrow \tilde{x}_0$
5:  for $i = 0, ..., maxits$ do
6:      $s \leftarrow \mathcal{K}^{\downarrow}(x) - x$
7:      $\tilde{s} \leftarrow \mathcal{K}^{\uparrow}(\tilde{x}) - \tilde{x}$
8:      Solve (7.6) to determine $\alpha$ and $\beta$
9:      cond1 $= \left\{ \frac{|s^T(x-\tilde{x})|}{\|s\|\,\|x-\tilde{x}\|} \leq \varepsilon_1$ and $\frac{|\tilde{s}^T(x-\tilde{x})|}{\|\tilde{s}\|\,\|x-\tilde{x}\|} \leq \varepsilon_1 \right\}$
10:     cond2 $= \left\{ |\alpha| \frac{\|s\|}{\|x\|} + |\beta| \frac{\|\tilde{s}\|}{\|\tilde{x}\|} \leq \varepsilon_2 \right\}$
11:     if cond1 or cond2 then
12:         break
13:     $x \leftarrow x + \alpha s$
14:     $\tilde{x} \leftarrow \tilde{x} + \beta \tilde{s}$
15: return $\frac{1}{2}(x + \tilde{x})$
```



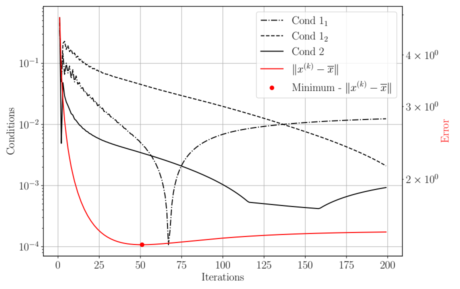
(a) Results for a system with Gaussian generated noise ($\eta = 0.004$) using the Mutual-Step algorithm.



(b) Results for a system with Gaussian generated noise ($\eta = 0.004$) using the Mutual-Step algorithm with the restriction $x \geq 0$.



(c) Results for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the Mutual-Step algorithm.



(d) Results for a system with Poisson generated noise ($I_0 = 8 \times 10^{13}$) using the Mutual-Step algorithm with the restriction $x \geq 0$.

Figure 7.7: Results for the Mutual-Step algorithm for two linear systems originated from CT problems. Matrix A has dimensions 19558×16384. The values of the quantities involved in the stopping conditions and the reconstruction error are shown for every iteration.

The paramters ε_1 and ε_2 were set to 10^{-4} to match the values used in [32]. Regarding line 8 of Algorithm 7, (7.6) can be easily solved using the formula for the inverse of a 2×2 matrix.

$$\begin{bmatrix} \|s\|^2 & -s^T\tilde{s} \\ -s^T\tilde{s} & \|\tilde{s}\|^2 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} -s^T(x-\tilde{x}) \\ -\tilde{s}^T(x-\tilde{x}) \end{bmatrix} \tag{7.6}$$

Just like for the Twin algorithm, we implemented the Mutual-Step algorithm with and without the restriction $x \geq 0$, whose results are shown in Figure 7.7. *Cond* 1_1 and *Cond* 1_2 correspond to the two quantities involved in the first stopping condition (line 9 of Algorithm 7); *Cond* 2 corresponds to the values used in the second stopping condition (line 10 of Algorithm 7). We will start by analyzing the results for the system perturbed with Gaussian noise presented in Figures 7.7a and 7.7b.

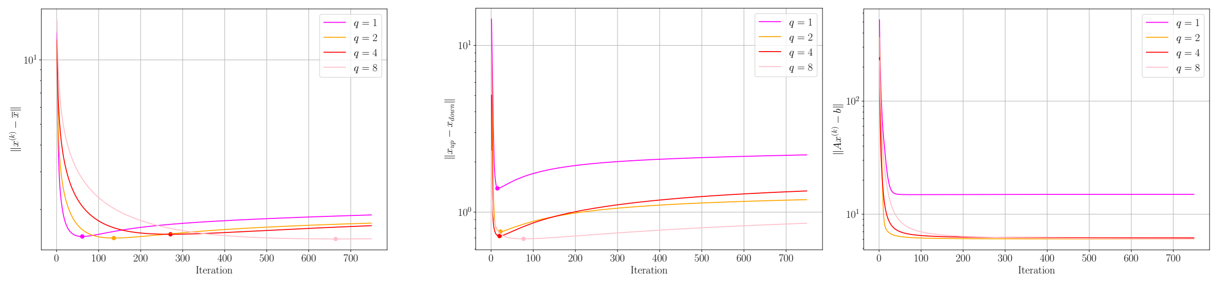
Figure 7.7a shows that the stopping iteration does not correspond exactly to the minimum of the reconstruction error. However, the value of the error at the stopping iteration, 4.98, is very close to the value at the minimum, 4.88. Still, both these values are higher than those given by the Twin algorithm at the minimum and at the stopping iterations (4.14 and 4.46). In Figure 7.7b we do not show the stopping iteration since this method did not converge using the parameters ε_1 and ε_2 used in the case without the restriction $x \geq 0$. Furthermore, the value of the minimum of the reconstruction error, 1.51, is not smaller than that for the Twin algorithm, 1.50. Overall, changing the parameters to $\varepsilon_1 = \varepsilon_2 = 10^{-3}$ for this system would have been a more adequate choice.

Regarding the results for the system perturbed with Poisson generated noise, for the case without the restriction $x \geq 0$ (Figure 7.7c), the stopping iteration corresponds to a local minimum of the reconstruction error instead of a global minimum. The results, using the restriction $x \geq 0$, shown in Figure 7.7d, are similar to the ones for the Gaussian noise (Figure 7.7b) where the method does not converge. Moreover, the values of the reconstruction error for the local minimum in Figure 7.7c and for the global minimum in Figure 7.7d, 4.65 and 1.39, are larger than those for the Twin algorithm, 3.77 and 1.38.

In conclusion, adding the restriction $x \geq 0$ improves the performance of the Twin algorithm by decreasing the minimum error of reconstruction. However, using $x \geq 0$, the Mutual-Step algorithm does not converge with the same parameters used for the case without restriction. Moreover, the reconstructor error is smaller for the Twin algorithm.

We finish this section by attempting to merge the Twin Algorithm, that has an effective stopping criteria, with the SRKAWOR algorithm, which can further decrease the reconstruction error. We can call this method the Twin Algorithm with Averaging. This algorithm can still be described by Algorithm 6, except that the downwards and upwards Kaczmarz sweeps are slightly different. We now explain the process of downwards Kaczmarz sweeps since the upwards sweep is analogous.

In a standard Kaczmarz sweep, there are m steps. In each step we use a row of the matrix, selected cyclically, to compute a new estimative of the solution. When we use averaging, instead of processing a single row per step, we process q rows whose results are averaged before moving on to the next step. This means that, if we are using the matrix sequentially, each Kaczmarz sweep of the Twin Algorithm with Averaging will have m/q steps. The assignment of each row to each thread is similar to the process shown in Figure 5.6a, except that we do not need a shuffled array of indices. Note that the Twin Algorithm with Averaging is equivalent to the Twin Algorithm if only one thread is used ($q = 1$).



(a) Reconstruction error.

(b) Error gauge.

(c) Residual.

Figure 7.8: Results for the Twin Algorithm with Averaging for a linear system originated from CT problems perturbed with Gaussian noise. Matrix A has dimensions 19558×16384. The values for the error, residual, and error gauge are shown for every iteration.

Figure 7.8 shows the results for the system with Gaussian noise (the results for Poisson noise are similar and we will not discuss them here). Figures 7.8a and 7.8b show a clear problem with this approach: when q increases, the minimum of the error gause gets progressively further (in iterations) from the minimum of the reconstruction error.

Table 7.5: Reconstruction error at its analytical minimum and at the minimum of the error gauge for the Twin Algorithm with Averaging for systems perturbed with noise sampled from a Gaussian distribution using the restriction $x \geq 0$.

q	Stopping Criteria		Analitically	
	Iterations	Value	Iteration	Value
1	15	1.71	61	1.50
2	22	2.01	136	1.47
4	20	2.85	271	1.53
8	77	2.44	664	1.46

Table 7.5 shows the value of the reconstruction error at the analytical minimum and at the minimum of the error gauge. Note that, although using $q > 1$ can decrease the analytical value of the minimum of the reconstruction error, using the number of iterations given by the stopping criteria gives progressively worse estimates for the point of semi-convergence. We conclude that this algorithm, at least with unitary uniform row weights ($\alpha = 1$), does not correctly estimate the number of iterations needed to have a good reconstructed image.

Chapter 8

Conclusion

8.1 Summary of Contributions

In this dissertation, we have implemented several algorithms based on the Kaczmarz method, including variations for consistent and inconsistent systems. We concluded that, although there are some variations that can outperform RK like the SRK method and sampling based on quasirandom numbers, the SRKWOR method is the fastest Kaczmarz-based method for consistent systems. For inconsistent systems, we concluded that, although RGS is faster than REK, the CGLS method is significantly faster than both methods.

We explored multiple approaches to parallelize the Kaczmarz method using shared and distributed memory and concluded that, in general, it is not possible to efficiently parallelize it. More specifically, we implemented the Randomized Kaczmarz with Averaging (RKA) algorithm and showed that its parallelization is not effective since, although the number of iterations is smaller for RKA, that decrease is not enough to make up for the cost of communication. Nevertheless, we introduced a new method, a blocked version of the RKA algorithm (RKAB), that can have the same effect that RKA has in decreasing the convergence horizon for inconsistent systems. Although the parallel implementation of RKAB cannot consistently beat the execution times of the sequential RK, the speedups regarding its sequential version are much improved compared to those of RKA. For the shared memory implementation of RKAB, we show that a good choice for the block size parameter is to use a number similar to the number of columns of the matrix of the system.

Lastly, we applied the Kaczmarz method to solving linear systems from CT problems. We show that the RK, CK, and SRKWOR methods exhibit semi-convergence, and that adding the constraint that all image pixels must have non-negative values decreases the minimum of the reconstruction error. However, these methods do not have a stopping criterion for inconsistent systems. For this reason, we implemented the Twin and Mutual-Step algorithms, based on the CK, that have a stopping criterion designed to get a solution close to that with the minimum reconstruction error. The incorporation of the previously discussed constraint with the Twin algorithm shows major improvements in the error of the reconstructed images.

8.2 Future Work

In Section 5.3.1, we analyzed how RKAB behaves for different uniform row weights for a consistent system. However, contrary to RKA, there is no formula for the optimal value of α. An improvement to the RKAB method would be to find this optimal α as a function of the block size.

When analyzing how the RKA method can be used to solve inconsistent systems in Section 5.6, we showed that using $\alpha = 1$ can decrease the convergence horizon proportionally to the number of threads. Moreover, we have shown that, although the optimal value of parameter α for consistent systems, α^*, is not optimal for inconsistent systems, it can still increase the rate of convergence of RKA. It would be interesting to find a parameter that can decrease the convergence horizon for an increasing number of threads, q, and the number of iterations needed to find that horizon.

To analyze the implementation of RKAB for both shared and distributed memory we need to first find the optimal block size for RKAB, which we concluded in Section 6.2 depends not only on the dimensions of the system but also on the number of processors. We leave a more comprehensive analysis of this behavior for future work.

In Chapter 7 we have compared how several different variations of the Kaczmarz method can be used to solve linear systems derived from CT problem. However, we focused on 2D tomographies with a single detector geometry (parallel beam) and used only one set of angles and a number of pixels. We leave a similar analysis for systems generated with other detection geometries, including 3D configurations, for future work.

Finally, in Section 7.4, we have shown that, for the Mutual-Step algorithm, using the parameters ε_1 and ε_2 defined by the authors, the method does not converge when integrated with the $x \geq 0$ constraint. We have yet to test this method with other values for these parameters.

Bibliography

[1] J. R. Senning, "Computing and estimating the rate of convergence," 2007.

[2] S. Kaczmarz, "Angenäherte auflösung von systemen linearer gleichungen (english translation by jason stockmann): Bulletin international de l'académie polonaise des sciences et des lettres," 1937.

[3] A. Ma, D. Needell, and A. Ramdas, "Convergence properties of the randomized extended gauss–seidel and kaczmarz methods," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 4, pp. 1590–1604, 2015.

[4] F. Deutsch, "Rate of convergence of the method of alternating projections," in *Parametric optimization and approximation*. Springer, 1984, pp. 96–107.

[5] F. Deutsch and H. Hundal, "The rate of convergence for the method of alternating projections, ii," *Journal of Mathematical Analysis and Applications*, vol. 205, no. 2, pp. 381–405, 1997.

[6] A. Galántai, "On the rate of convergence of the alternating projection method in finite dimensional spaces," *Journal of mathematical analysis and applications*, vol. 310, no. 1, pp. 30–44, 2005.

[7] F. Natterer, "Mathematics of computerized tomography.- john wiley & sons ltd," *New York*, 1986.

[8] G. T. Herman and L. B. Meyer, "Algebraic reconstruction techniques can be made computationally efficient (positron emission tomography application)," *IEEE transactions on medical imaging*, vol. 12, no. 3, pp. 600–609, 1993.

[9] H. G. Feichtinger, C. Cenker, M. Mayer, H. Steier, and T. Strohmer, "New variants of the pocs method using affine subspaces of finite codimension with applications to irregular sampling," in *Visual Communications and Image Processing'92*, vol. 1818. International Society for Optics and Photonics, 1992, pp. 299–310.

[10] T. Strohmer and R. Vershynin, "A randomized kaczmarz algorithm with exponential convergence," *Journal of Fourier Analysis and Applications*, vol. 15, pp. 262–278, 2007.

[11] D. Needell, "Randomized kaczmarz solver for noisy linear systems," *BIT Numerical Mathematics*, vol. 50, no. 2, pp. 395–403, 2010.

[12] X. Chen, "The kaczmarz algorithm, row action methods, and statistical learning algorithms," *Frames and harmonic analysis*, vol. 706, pp. 115–127, 2018.

[13] M. Schmidt, "Notes on randomized kaczmarz," *Randomized Algorithms*, 2015.

[14] I. H. Sloan, I. Sloan, and S. Joe, *Lattice methods for multiple integration*. Oxford University Press, 1994.

[15] J. H. Halton, “On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals,” *Numerische Mathematik*, vol. 2, pp. 84–90, 1960.

[16] I. M. Sobol, “Uniformly distributed sequences with an additional uniform property,” *USSR Computational Mathematics and Mathematical Physics*, vol. 16, no. 5, pp. 236–242, 1976.

[17] D. Leventhal and A. S. Lewis, “Randomized methods for linear constraints: convergence rates and conditioning,” *Mathematics of Operations Research*, vol. 35, no. 3, pp. 641–654, 2010.

[18] D. Needell, R. Zhao, and A. Zouzias, “Randomized block kaczmarz method with projection for solving least squares,” *Linear Algebra and its Applications*, vol. 484, pp. 322–343, 2015.

[19] A. Zouzias and N. M. Freris, “Randomized extended kaczmarz for solving least squares,” *SIAM Journal on Matrix Analysis and Applications*, vol. 34, no. 2, pp. 773–793, 2013.

[20] V. Rokhlin and M. Tygert, “A fast randomized algorithm for orthogonal projection,” *arXiv preprint arXiv:0912.1135*, 2009.

[21] Z.-Z. Bai and W.-T. Wu, “On greedy randomized kaczmarz method for solving large sparse linear systems,” *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. A592–A606, 2018.

[22] Y. Yaniv, J. D. Moorman, W. Swartworth, T. Tu, D. Landis, and D. Needell, “Selectable set randomized kaczmarz,” *arXiv preprint arXiv:2110.04703*, 2021.

[23] J. D. Moorman, T. K. Tu, D. Molitor, and D. Needell, “Randomized kaczmarz with averaging,” *BIT Numerical Mathematics*, vol. 61, no. 1, pp. 337–359, 2021.

[24] D. Gordon and R. Gordon, “Component-averaged row projections: A robust, block-parallel scheme for sparse linear systems,” *SIAM Journal on Scientific Computing*, vol. 27, no. 3, pp. 1092–1117, 2005.

[25] J. Liu, S. J. Wright, and S. Sridhar, “An asynchronous parallel randomized kaczmarz algorithm,” *arXiv preprint arXiv:1401.4780*, 2014.

[26] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent,” *Advances in neural information processing systems*, vol. 24, 2011.

[27] Y. Nesterov, “Efficiency of coordinate descent methods on huge-scale optimization problems,” *SIAM Journal on Optimization*, vol. 22, no. 2, pp. 341–362, 2012.

[28] J. Liu, S. Wright, C. Ré, V. Bittorf, and S. Sridhar, “An asynchronous parallel stochastic coordinate descent algorithm,” in *International Conference on Machine Learning*. PMLR, 2014, pp. 469–477.

[29] T. Wallace and A. Sekmen, “Deterministic versus randomized kaczmarz iterative projection,” *arXiv preprint arXiv:1407.5593*, 2014.

[30] R. L. Siddon, “Fast calculation of the exact radiological path for a three-dimensional ct array,” *Medical physics*, vol. 12, no. 2, pp. 252–255, 1985.

[31] L. A. Shepp and B. F. Logan, “The fourier reconstruction of a head section,” *IEEE Transactions on nuclear science*, vol. 21, no. 3, pp. 21–43, 1974.

[32] B. S. van Lith, P. C. Hansen, and M. E. Hochstenbach, “A twin error gauge for kaczmarz’s iterations,” *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. S173–S199, 2021.

[33] Y. Zhu, M. Zhao, Y. Zhao, H. Li, and P. Zhang, “Noise reduction with low dose ct data based on a modified rof model,” *Optics express*, vol. 20, no. 16, pp. 17 987–18 004, 2012.

[34] P. C. Hansen, J. S. Jørgensen, and P. W. Rasmussen, “Stopping rules for algebraic iterative reconstruction methods in computed tomography,” in *2021 21st International Conference on Computational Science and Its Applications (ICCSA)*. IEEE, 2021, pp. 60–70.

[35] D. Needell, R. Ward, and N. Srebro, “Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm,” *Advances in neural information processing systems*, vol. 27, 2014.

[36] D. Needell and J. A. Tropp, “Paved with good intentions: analysis of a randomized block kaczmarz method,” *Linear Algebra and its Applications*, vol. 441, pp. 199–221, 2014.

[37] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, “Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing,” in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–11.

Appendix A

Mathematical Definitions

Throughout this document, we will refer to the ith row of a matrix A as $A^{(i)}$ and, to the jth column, as $A_{(j)}$. The **conjugate transpose** of a $m \times n$ matrix with complex entries is the $n \times m$ matrix obtained by taking the transpose and then taking the complex conjugate of each entry. It is denoted by

$$A^* = (\overline{A})^T \,. \tag{A.1}$$

The **euclidean norm** or L^2 norm of a vector in $\mathbb{R}^n$ can be written as $\|x\|_2 = \left(\sum_i^n x_i^2\right)^{1/2}$, and measures how much a vector extends in space. The **spectral norm** of a matrix measures how much the matrix can "scale" or "stretch" a vector and is defined by

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \max_{\|x\|_2=1} \|Ax\|_2 \,. \tag{A.2}$$

Matrix A can be decomposed using singular value decomposition such that $A = U\Sigma V^T$. Using the fact that U is an unitary matrix, meaning that $\|Ux_0\|_2 = \|x_0\|_2$, we have that

$$\|A\|_2 = \max_{\|x\|=1} \|Ax\|_2 = \max_{\|x\|_2=1} \|U\Sigma V^T x\|_2 = \max_{\|x\|_2=1} \|\Sigma V^T x\|_2 \,. \tag{A.3}$$

We will now define that $y = V^T x$, and, since V^T is also a unitary matrix,

$$\|y\|_2 = \|V^T x\|_2 = \|x\|_2 = 1 \,. \tag{A.4}$$

The definition of the spectral norm can now be rewritten as $\|A\|_2 = \max_{\|y\|_2=1} \|\Sigma y\|_2$. Note that $\Sigma = diag(\sigma_1, ..., \sigma_n)$, where σ_1 is the largest singular value of matrix A. Maximizing $\|\Sigma y\|_2$ subject to $\|y\|_2 = 1$ yields $y = (1, 0, 0, ..., 0)^T$ and, subsequently, $\|A\|_2 = \sigma_{max}(A) = \sqrt{\lambda_{max}(A^* A)}$. In summary, the spectral norm of a matrix A is given by the **largest singular value of** A, that is, the square root of the largest eigenvalue of matrix $A^* A$, where A^* denotes the conjugate transpose defined in (A.1). It is also important to measure how much a matrix can shrink vectors, that is

$$\min_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \min_{\|x\|_2=1} \|Ax\|_2 \,. \tag{A.5}$$

Using the same reasoning as before, we can easily conclude that

$$\min_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sigma_{min}(A) = \sigma_{max}(A^{-1}) = \frac{1}{\|A^{-1}\|_2} \tag{A.6}$$

The spectral norm measures how much can a matrix possibly increase the norm of a vector, similar to a ”worst-case” norm. The **Frobenius norm** describes an ”average case” norm such that

$$\|A\|_F = \sqrt{\mathbb{E}\,\frac{\|Ax\|_2^2}{\|x\|_2^2}}\,. \tag{A.7}$$

The Frobenius norm is often easier to compute than the spectral norm since it is just the Euclidean norm of the matrix, that is,

$$\begin{aligned} \|A\|_F &= \sqrt{\sum_{i=1}^{m} \|A_i\|_2^2} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2} = \sqrt{\sum_i \Big(\sum_j a_{ji}^T a_{ij}\Big)} = \sqrt{\sum_i \Big[A^*A\Big]_{ii}} \\ &= \sqrt{tr(A^*A)} = \sqrt{\sum_{i=1}^{min\{m,n\}} \sigma_i^2(A)}\,. \end{aligned} \tag{A.8}$$

A useful property satisfied by the L^2 norm is that, if the product Ax exists, then

$$\|Ax\|_2 = \|b\|_2 \leq \|A\|_2\,\|x\|_2\,. \tag{A.9}$$

Since $x = A^{-1}b$, another result is that

$$\|x\|_2 \leq \|A^{-1}\|_2\,\|b\|_2\,. \tag{A.10}$$

A.1 Least-Squares Solution of an Overdetermined Linear System

In this section we show how to obtain the normal equations that satisfy (2.2). Let us begin by simplifying the cost function, such that

$$\|Ax-b\|_2^2 = (Ax-b)^T(Ax-b) = (Ax)^T(Ax)-(Ax)^Tb-b^T(Ax)+b^Tb = x^TA^TAx-2b^TAx+b^Tb. \tag{A.11}$$

If u and x are two column vectors, then

$$\frac{\partial}{\partial x}(u^Tx) = u^T\,. \tag{A.12}$$

If x is a column vector and A is a matrix, then

$$\frac{\partial}{\partial x}(x^TAx) = x^T(A+A^T)\,. \tag{A.13}$$

With both (A.12) and (A.13) we can calculate the gradient of the cost function:

$$\nabla\|Ax-b\|_2^2 = x^T(A^TA+(A^TA)^T) - 2b^TA = 2x^TA^TA - 2b^TA\,. \tag{A.14}$$

The x value for which the cost function is minimum can be computed by finding where the gradient of the cost function is 0, that is,

$$\nabla\|Ax-b\|_2^2 = 0 \iff x^TA^TA = b^TA \iff A^TAx = A^Tb \Rightarrow x_{LS} = (A^TA)^{-1}A^Tb. \tag{A.15}$$

A.2 Least Euclidean Norm Solution of an Underdetermined Linear System

Using Lagrange multipliers, the optimization problem in (2.3) can be redefined, such that x is the value that minimizes the quantity

$$L(x,\lambda) = x^Tx + \lambda^T(Ax-b)\,. \tag{A.16}$$

To find the minimum, the zeros of the partial derivatives of $L(x,\lambda)$ must be computed:

$$\frac{\partial}{\partial x} = 2x + A^T\lambda = 0 \Rightarrow x = -A^T\lambda/2\,; \tag{A.17}$$

$$\frac{\partial}{\partial \lambda} = Ax - b = 0\,. \tag{A.18}$$

Substituting (A.17) onto (A.18), we get that

$$\lambda = -2(AA^T)^{-1}b\,, \tag{A.19}$$

and, subsequently,

$$x_{LN} = A^T(AA^T)^{-1}b \tag{A.20}$$

A.3 Condition Number and Scaled Condition Number

The solution of a linear squared system $Ax = b$ can be simply computed as $x = A^{-1}b$. If there is some induced error in b due to imprecisions in the measurements or due to errors in previous calculations, it is useful to know how much those errors will affect the solution. This is where the condition number comes in: its goal is to measure how much the output value changes when small changes are made to the input value. Suppose that b is perturbed by Δb, such that $b \to b + \Delta b$. The new solution will be $x + \Delta x$ and the change in x is given by:

$$\Delta x = A^{-1}\Delta b\,. \tag{A.21}$$

If small changes in b result in small changes in x, we say that the system is well-conditioned. Otherwise, if a small Δb results in a large Δx, the system is said to be ill-conditioned. From (A.10), we get that

$$\|\Delta x\|_2 \leq \|A^{-1}\|_2 \, \|\Delta b\|_2 \, , \tag{A.22}$$

and, from (A.9), one can obtain

$$\frac{1}{\|x\|_2} \leq \frac{\|A\|_2}{\|b\|_2} \, . \tag{A.23}$$

Finally, both (A.22) and (A.23) yield

$$\frac{\|\Delta x\|_2}{\|x\|_2} \leq \|A\|_2 \, \|A^{-1}\|_2 \frac{\|\Delta b\|_2}{\|b\|_2} \, . \tag{A.24}$$

In summary, if $\|A\|_2 \|A^{-1}\|_2$ is small, $\|\Delta x\|_2/\|x\|_2$ will be small when $\|\Delta b\|_2/\|b\|_2$ is small; on the contrary, if $\|A\|_2 \, \|A^{-1}\|_2$ is large, $\|\Delta x\|_2/\|x\|_2$ can be much larger than $\|\Delta b\|_2/\|b\|_2$. It is now useful to define the **condition number** as

$$k(A) = \|A\|_2 \, \|A^{-1}\|_2 \, . \tag{A.25}$$

When $k(A)$ is small, the matrix A is well-conditioned and, when $k(A)$ is large, the matrix A is badly conditioned or ill-conditioned. The **scaled condition number** uses the Frobenius norm of the matrix instead of the spectral norm, meaning that

$$\kappa(A) = \|A\|_F \, \|A^{-1}\|_2 \, . \tag{A.26}$$

The relationship between both numbers is

$$1 \leq \frac{\kappa(A)}{\sqrt{n}} \leq k(A) \, . \tag{A.27}$$

Appendix B

Relationship with Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) is a particular case of the more general gradient descent method. The (Gradient Descent (GD)) method is a first-order iterative algorithm for finding a local minimum of a differentiable function, the objective function, by taking repeated steps in the opposite direction of the gradient, that is, the direction of the steepest descent. Each iteration of the gradient descent method can be written as

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla F(x^{(k)}) \tag{B.1}$$

where α_k is the step and $F(x)$ is the objective function to minimize. $F(x)$ is a sum of functions associated with each data entry, represented by $f_i(x)$. In the context of machine learning, the GD method is used to minimize an objective function that usually depends on a very large set of data, which makes it very computationally expensive. The difference between SGD and GD is that in SGD the gradient of the objective function is estimated by using a random subset of data, instead of using the entire data set. The size of the subset is called the minibatch size. If each function, $f_i(x)$, has an associated weight, $w(i)$, that influences the probability of that data entry being chosen, we say that [1]

$$f_i^{(w)}(x) = \frac{1}{w(i)} f_i(x) \;, \tag{B.2}$$

and the objective function is written as an expectation

$$F(x) = \mathbb{E}^{(w)}[f_i^{(w)}(x)] \;. \tag{B.3}$$

If the minibatch size is one, that is, if we only use one data entry per iteration, the gradient of the objective function can be written as $\nabla F(x) \approx \nabla f_i^{w}(x) = \nabla \frac{1}{w(i)} f_i(x)$, and the algorithm can now be written as

$$x^{(k+1)} = x^{(k)} - \frac{\alpha_k}{w(i_k)} \nabla f_{i_k}(x_k) \tag{B.4}$$

[1] For more a more detailed explanation see Section 3 of [35].

where i_k is the chosen data entry in iteration k. Chen [12] showed that the Randomized Kaczmarz method is a special case of SGD using one data entry per iteration. If the objective function is $F(x) = \frac{1}{2}\|Ax - b\|^2$, then $f_i(x) = \frac{1}{2}(b_i - \langle A^{(i)}, x^{(k)}\rangle)^2$. The associated weight is the probability of choosing row i, that is, $\|A^{(i)}\|^2/\|A\|_F^2$. Substituting this definition in (B.4), we have that

$$x^{(k+1)} = x^{(k)} + \alpha_k \|A\|_F^2 \frac{b_i - \langle A^{(i)}, x^{(k)}\rangle}{\|A^{(i)}\|^2} A^{(i)T} , \tag{B.5}$$

and the (2.4) is recovered if the step is set to $\alpha_k = \frac{1}{\|A\|_F^2}$.

Appendix C

Variations of the Kaczmarz Method with Blocks

C.1 Randomized Block Kaczmarz Method

Needell and Tropp [36] were the first to develop a randomized block version of the Kaczmarz method, the RBK method. Each block is a subset of rows of matrix A chosen with a randomized control scheme. The idea behind the Randomized Block Kaczmarz method is to speed up the process by enforcing many constraints at once. Let us consider that, in each iteration k, we select a subset of row indices of A, τ_k. The current iterate $x^{(k+1)}$ is computed by projecting the previous iterate $x^{(k)}$ onto the solution of $A_{\tau_k}x = b_{\tau_k}$, that is

$$x^{(k+1)} = x^{(k)} + (A_{\tau_k})^{\dagger}(b_{\tau_k} - A_{\tau_k}x^{(k)}) \, , \tag{C.1}$$

where $(A_{\tau_k})^{\dagger}$ denotes the Moore–Penrose inverse (pseudoinverse) and the block τ_k is chosen from a partition $T = \{\tau_1, ..., \tau_l\}$, where l is the number of blocks. If $l = m$, that is, if the number of blocks is equal to the number of rows, each block consists of a single row and we recover the Randomized Kaczmarz method. The block τ_k can be chosen from the partition using one of two methods: it can be chosen randomly from the partition independently of all previous choices, or it can be sampled without replacement, an alternative that Needell and Tropp found to be more effective. In the latter, the block can only contain rows that have yet to be selected. Only when all rows are selected can there be a reusage of rows. The partition of the matrix into different blocks is called row-paving. The calculation of the pseudoinverse $(A_\tau)^{\dagger}$ in each iteration is a computationally expensive step. But, if the submatrix (A_τ) is well-conditioned, we can use algorithms like CGLS to efficiently calculate it. Similarly to RK, the Randomized Block Kaczmarz method exhibits an expected linear rate of convergence. It is important to determine in which cases it is beneficial to use the Randomized Block Kaczmarz method to detriment of the Randomized Kaczmarz method. The first case is if we have matrices with good row paving, that is, the blocks are well conditioned. One example of matrices with good row paving is row-standardized matrices. In the presence of good row paving, the matrix-vector multiplication and the computation of the pseudoinverse can be very efficient, meaning that one iteration of the block method has roughly the same cost as one iteration of the standard method. A second case is connected with the implementation of the algorithm: the simple Kaczmarz method transfers a new equation into working memory in each iteration, making the total time spent in data transfer throughout the algorithm excessive; on the other hand, not only does the block Kaczmarz algorithm move large data blocks into working memory at a time, but it also relies on matrix-vector multiplication that can be accelerated using basic linear algebra subroutines (BLASx) [37].

C.2 Randomized Double Block Kaczmarz Method

We have seen in Section 3.2 that the Randomized Extended Kaczmarz method is a variation of the Randomized Kaczmarz method that converges to the least squares solution x_{LS}. In Section C.1 we described situations where the Randomized Block Kaczmarz for consistent systems can outperform the Randomized Kaczmarz method. Needell, Zhao, and Zouzias [18] developed a new algorithm called Randomized Double Block Kaczmarz (RDBK) method that combines both the REK and the RBK algorithm with the goal of developing a method that solves inconsistent systems with accelerated convergence. Recall that in REK there is a projection step that makes use of a single column of matrix A and there is the Kaczmarz step that utilizes a single row. This means that, in a block version of REK, we need both a column partition for the projection step and a row partition for the Kaczmarz step, hence the name ”Double Block”. In each iteration k we select a subset of row indices of A, τ_k, and a subset of column indices of A, v_k. The projection step and the Kaczmarz step can then be written as

$$z^{(k+1)} = z^{(k)} - A_{v_k}(A_{v_k})^{\dagger} z^{(k)} , \quad \text{and} \tag{C.2}$$

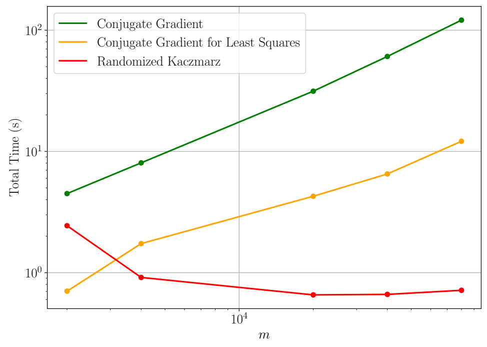
$$x^{(k+1)} = x^{(k)} + (A_{\tau_k})^{\dagger}(b_{\tau_k} - z^{(k+1)}_{\tau_k} - A_{\tau_k}x^{(k)}) . \tag{C.3}$$

The initialization variables are $x^{(0)} = 0$ and $z^{(0)} = b$. This method can then have an accelerated convergence when compared to other methods that converge to the least-squares solution of a given system, if the block matrices are well-conditioned and fast multiplication for matrix-vector can be used; even if the blocks are not well-conditioned, transferring large data blocks at a time can improve the runtime.

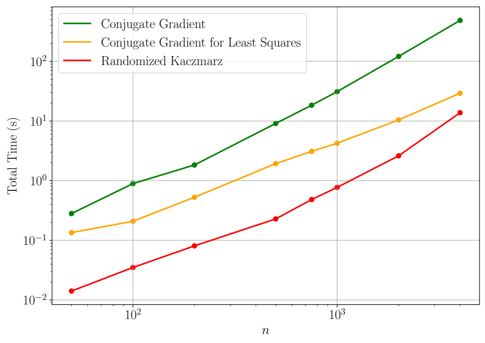
Appendix D

Results for the Sequential Version Using the Second Dense Data Set

In this appendix, we discuss the results of the variations of the Kaczmarz method for the second dense data set. This data set contains matrices with rows with similar norms, in contrast with the first dense data set that has rows with very distinct norms, whose results were analyzed in Chapter 4. The goal of this analysis is to determine how row norms affect the relative performance between methods since many of them use row selection criteria based on probability distribution dependent on row norms.



(a) Computational time until convergence for several overdetermined systems using a fixed number of columns and a varying number of rows.



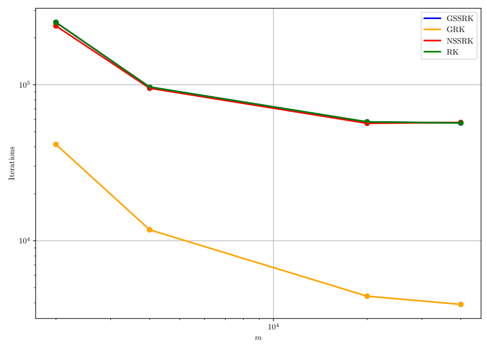
(b) Computational time until convergence for several overdetermined systems using a fixed number of rows and a varying number of columns.

Figure D.1: Comparison between RK, CG and CGLS for dense overdetermined systems.

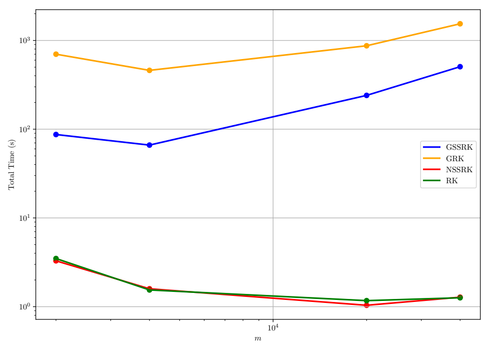
We start with the comparison between RK, CG and CGLS. Figure D.1 shows very similar results to the ones from Figure 4.7, meaning that RK can also outperform CG and CGLS for dense systems that have rows with similar norms.

Figure D.2 shows the results for methods GSSRK, NSSRK, GRK, and RK. Just like before, results are very similar to Figure 4.9. This is expected since all these methods rely on the same probability distribution for row sampling, so their relative performance should be the same for datasets with different distributions of row norms.

Figure D.3 shows the results for methods RK, CK, SRK, and , SRKWOR. These results are the same as the ones from Figure 4.9 with one exception: for this dataset, RK and SRK have similar iterations

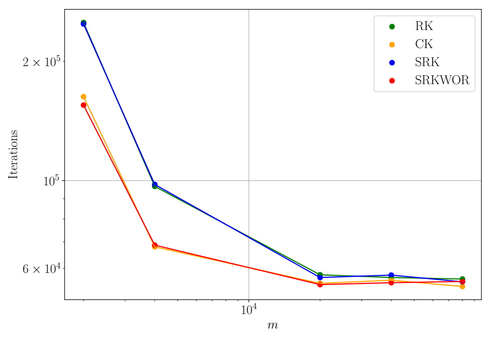


(a) Number of iterations for several overdetermined systems as a function of the number of rows. Note that the iterations for GSSRK, NSSRK, and RK and overlapped.

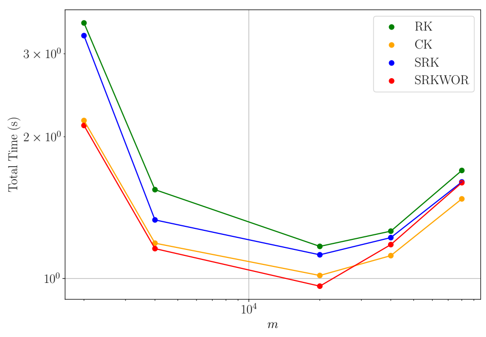


(b) Computational time until convergence for several overdetermined systems as a function of the number of rows.

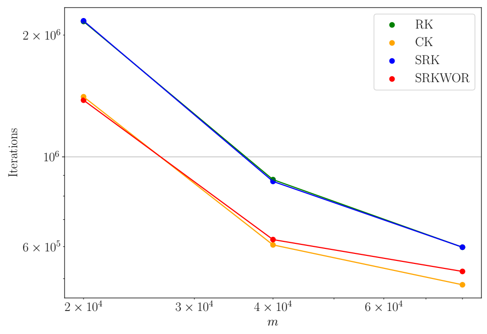
Figure D.2: Results for some variations of the Kaczmarz algorithm for dense systems using a fixed number of columns $n = 1000$ and a varying number of rows.



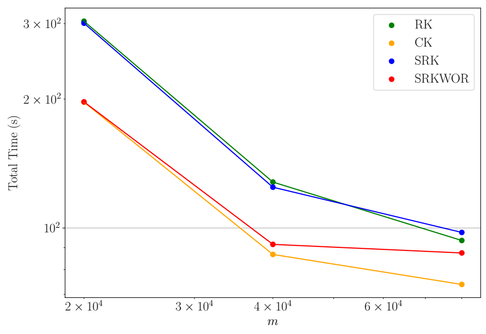
(a) Number of iterations for several overdetermined systems with $n = 1000$ as a function of the number of rows.



(b) Computational time until convergence for several overdetermined systems with $n = 1000$ as a function of the number of rows.

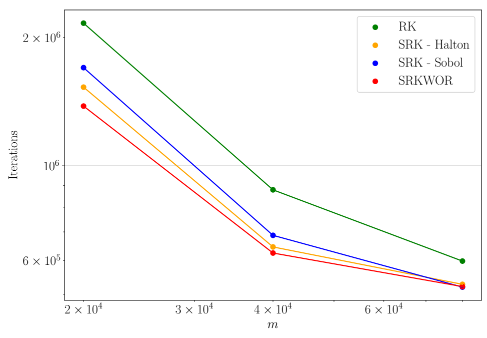


(c) Number of iterations for several overdetermined systems with $n = 10000$ as a function of the number of rows.

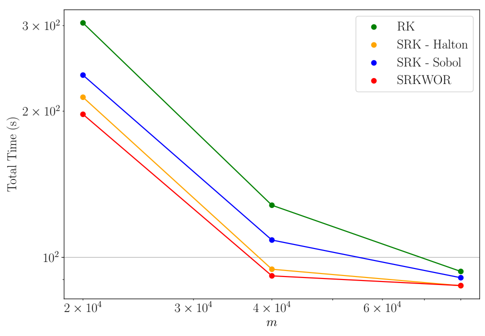


(d) Computational time until convergence for several overdetermined systems with $n = 10000$ as a function of the number of rows.

Figure D.3: Results for some variations of the Kaczmarz algorithm for dense systems using a fixed number of columns and a varying number of rows.



(a) Number of iterations for several overdetermined systems as a function of the number of rows.



(b) Computational time until convergence for several overdetermined systems as a function of the number of rows.

Figure D.4: Results for some variations of the Kaczmarz algorithm for dense systems using a fixed number of columns $n = 10000$ and a varying number of rows.

and execution times. This is in accordance with intuition since, having rows with very similar norms, the probability distribution used by RK will be very similar to a uniform probability distribution, the one used by SRK.

Finally, Figure D.4 shows the results for the quasirandom numbers. These results are once more similar to the ones for the first dense data set, shown in Figure 4.13, with a slight difference in the relative performance between both sequences of quasirandom numbers: for this dataset, the Halton sequence seems faster in both iterations and time than the Sobol sequence.

Appendix E

Pseudocode of Sequential Implementations

Algorithm 8 Pseudocode for an iteration of the sequential implementation of RK. x corresponds to the estimate of the solution in the current iteration, it. $A_i^{(row)}$ corresponds to the i-th column of a given row of matrix A. $\mathcal{D}$ is a probability distribution that samples row indices with probability proportional to their norms.

```
1: $it \leftarrow it + 1$
2: $row \leftarrow$ sampled from $\mathcal{D}$
3: $scale \leftarrow \frac{b_{row} - \langle A^{(row)}, x^{(prev)} \rangle}{\|A^{(row)}\|_2^2}$
4: for $i = 0, ..., N$ do
5:     $x_i \leftarrow x_i + scale \times A_i^{(row)}$
```

Algorithm 9 Pseudocode for an iteration of the sequential implementation of RKA. $x^{(prev)}$ and x correspond to the estimates of the solution in the previous and current iteration, it. $A_i^{(row)}$ corresponds to the i-th column of a given row of matrix A. The number of threads is given by q. $\mathcal{D}$ is a probability distribution that samples row indices with probability proportional to their norms. Row weights are uniform and given by α.

```
1: $it \leftarrow it + 1$
2: for $i = 0, ..., N$ do
3:     $x_i^{(prev)} \leftarrow x_i$
4: for $k = 0, ..., q$ do
5:     $row \leftarrow$ sampled from $\mathcal{D}$
6:     $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(prev)} \rangle}{\|A^{(row)}\|_2^2}$
7:     for $i = 0, ..., N$ do
8:         $x_i \leftarrow x_i + \frac{scale \times A_i^{(row)}}{q}$
```

Algorithm 10 Pseudocode for an iteration of the sequential implementation of RKAB. $x^{(thread)}$ corresponds to the estimate of the solution in each thread. $x^{(prev)}$ and x correspond to the estimate of the solution in the previous and current iterations. $A_i^{(row)}$ corresponds to the i-th column of a given row of matrix A. The number of threads is given by q. $\mathcal{D}$ is a probability distribution that samples row indices with probability proportional to their norms.

```
1:  $it \leftarrow it + 1$
2:  for $i = 0, ..., N$ do
3:      $x_i^{(prev)} \leftarrow x_i$
4:  for $k = 0, ..., q$ do
5:      for $i = 0, ..., N$ do
6:          $x_i^{(thread)} \leftarrow x_i^{(prev)}$
7:      for $b = 0, ..., block\ size - 1$ do
8:          $row \leftarrow$ sampled from $\mathcal{D}$
9:          $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(thread)} \rangle}{\|A^{(row)}\|_2^2}$
10:         for $i = 0, ..., N$ do
11:             $x_i^{(thread)} \leftarrow x_i^{(thread)} + scale \times A_i^{(row)}$
12:     $row \leftarrow$ sampled from $\mathcal{D}$
13:     $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(thread)} \rangle}{\|A^{(row)}\|_2^2}$
14:     for $i = 0, ..., N$ do
15:         $x_i^{(thread)} \leftarrow x_i^{(thread)} + scale \times A_i^{(row)} - x_i^{(prev)}$
16:     for $i = 0, ..., N$ do
17:         $x_i \leftarrow x_i + \frac{x_i^{(thread)}}{q}$
```

Appendix F

Pseudocode of Parallel Implementations Combining Shared and Distributed Memory

Algorithm 11 Pseudocode for an iteration of the parallel implementation of RKA using both shared and distributed memory. $x^{(prev)}$ and x correspond to the estimates of the solution in the previous and current iteration, it. $A_i^{(row)}$ corresponds to the i-th column of a given row of matrix A. The number of threads is given by q and the number of processes/tasks is given by np. $\mathcal{D}$ is a probability distribution that samples row indices with probability proportional to their norms. Row weights are uniform and given by α.

```
1: $it \leftarrow it + 1$
2: OMP for $i = 0, ..., N$ do
3:     $x_i^{(prev)} \leftarrow x_i$
4:     $x_i \leftarrow 0$
5: $row \leftarrow$ sampled from $\mathcal{D}$
6: $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(prev)} \rangle}{\|A^{(row)}\|_2^2}$
7: OMP critical
8:     for $i = 0, ..., N$ do
9:         $x_i \leftarrow \frac{x_i^{(prev)} + scale \times A_i^{(row)}}{np \times q}$
10: OMP barrier
11: OMP single
12:     MPI Allreduce $(x, +)$
```

Algorithm 12 Pseudocode for an iteration of the parallel implementation of RKAB using both shared and distributed memory. $x^{(thread)}$ corresponds to the estimate of the solution in each thread. x corresponds to the estimate of the solution in the current iteration, it, computed using the results from all threads. $A_i^{(row)}$ corresponds to the i-th column of a given row of matrix A. The number of threads is given by q and the number of processes/tasks is given by np. $\mathcal{D}$ is a probability distribution that samples row indices with probability proportional to their norms.

```
1:  $it \leftarrow it + 1$
2:  $row \leftarrow$ sampled from $\mathcal{D}$
3:  $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x\rangle}{\|A^{(row)}\|_2^2}$
4:  for $i = 0, ..., N$ do
5:      $x_i^{(thread)} \leftarrow x_i + scale \times A_i^{(row)}$
6:  for $b = 0, ..., block\ size - 1$ do
7:      $row \leftarrow$ sampled from $\mathcal{D}$
8:      $scale \leftarrow \alpha \times \frac{b_{row} - \langle A^{(row)}, x^{(thread)}\rangle}{\|A^{(row)}\|_2^2}$
9:      for $i = 0, ..., N$ do
10:         $x_i^{(thread)} \leftarrow x_i^{(thread)} + scale \times A_i^{(row)}$
11: for $i = 0, ..., N$ do
12:     $x_i^{(thread)} \leftarrow x_i^{(thread)} - x_i$
13: OMP barrier
14: OMP critical
15:     for $i = 0, ..., N$ do
16:         $x_i \leftarrow x_i + \frac{x_i^{(thread)}}{q}$
17: OMP barrier
18: OMP for $i = 0, ..., N$ do
19:     $x_i \leftarrow \frac{x_i}{np}$
20: OMP single
21:     MPI Allreduce $(x, +)$
```