Parallelization of the Kaczmarz Algorithm

Inês Alves Ferreira *Electrical and Computer Engineering Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento*
Instituto Superior Técnico, Universidade de Lisboa
ines.alves.ferreira@tecnico.ulisboa.pt

***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.

***Index Terms*—Linear systems; Iterative algorithms; Parallel and distributed computing; Rate of convergence; Least-Squares problem**

I. 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 that can accurately solve linear systems, 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 gets improved 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 where each equation corresponds to a data entry. 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 iterative methods 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.

A. 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.

B. Organization of the Document

The organization of this document is as follows. In Section II 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 Section III we introduce some of the methods inspired by the Randomized Kaczmarz algorithm. In Section IV we present the implementation details of the sequential version of the algorithm and some of its variations, together with the experimental results. In Sections V and VI we discuss several approaches to parallelizing the Kaczmarz method using shared and distributed memory, respectively. Finally, in Section VII, we conclude this dissertation and discuss future work.

II. SYSTEMS OF LINEAR EQUATIONS: PROPERTIES AND ALGORITHMS

We start this section by outlining the categorization of linear systems based on factors such as matrix dimensions and the existence

of a solution. We then elaborate on the methods used to solve linear systems and the corresponding computational algorithms employed. Finally, we present the Kaczmarz algorithm, along with its key attributes, and subsequently introduce some randomized variants.

A. Types of Linear Systems

A linear system of equations can be written as $Ax = b$, 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**, if there is at least one solution for the system, or **inconsistent**, if there is no solution. When considering the relationship between the number of equations and variables, systems can be classified as **overdetermined**, if $m \geq n$, or **underdetermined**, if $m < n$. For overdetermined consistent systems, we are interested in finding the exact solution of the system, x^*. However, most real-world overdetermined systems are inconsistent and, in that case, we are interested in finding the least-squares solution, that minimizes the residual. For underdetermined systems, there are more degrees of freedom than equations, meaning that systems often have infinite solutions. For these cases, we are interested in finding the least Euclidean norm solution.

B. 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, the computational cost and memory usage can be high for large matrices when using these methods. Iterative methods generate a sequence of approximate solutions that get increasingly closer to the exact solution with each iteration. They are generally less computationally demanding than direct methods and the precision of the solutions given by iterative methods can be controlled by external parameters. Therefore, if high precision is not a requirement, iterative methods can outperform direct methods, especially for large and sparse matrices. A particular class of iterative methods is row-action methods. These use one row of matrix A in each iteration. The Kaczmarz method, that will now be introduced, is one example.

C. Kaczmarz Method

The Kaczmarz method [1] 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 i-th coordinate of b. Each iteration $x^{(k+1)}$ can be thought of as the projection of $x^{(k)}$ onto the hyperplane defined by $\langle A^{(i)}, x \rangle = b_i$. The original version of the algorithm can then be written as

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

with $i = k \bmod m$ and where α_i is a relaxation parameter that is set to 1. Each iteration of the Kaczmarz method can be thought of as the projection of $x^{(k)}$ onto H_i, the hyperplane defined by $H_i = x : \langle A^{(i)}, x \rangle = b_i$. 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 highly coherent matrices, that is, matrices for which the angle between consecutive rows is small, the convergence of the cyclic Kaczmarz method can be slow. However, convergence can be accelerated if the rows of the matrix are used in a random fashion. This was experimentally observed and led to the development of randomized versions of the Kaczmarz method.

D. Randomized Kaczmarz Method

Kaczmarz [1] showed that, if the linear system is solvable, the method converges to the solution x^*, but the rate at which the method converges is difficult to quantify. Furthermore, it has been observed [2]–[4] that, instead of using the rows of A in a cyclic manner, choosing rows randomly can improve the rate of convergence of the algorithm. Therefore, to tackle these two questions, Strohmer and Vershynin [5] 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

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

This version of the algorithm is called the Randomized Kaczmarz (RK) method. Strohmer and Vershynin proved that RK has exponential error decay. Moreover, they showed that in extremely overdetermined systems the Randomized Kaczmarz method outperforms all other known algorithms. They reignited not only the research on the Kaczmarz method but also triggered much investigation into developing and analyzing randomized linear solvers. Later, Needell [6] extended these results for inconsistent systems, showing that RK reaches an estimate that is within a fixed distance from the solution, called the convergence horizon.

E. Simple Randomized Kaczmarz Method

Apart from the comparison between the original Kaczmarz method with the Randomized Kaczmarz method, Strohmer, and Vershynin [5] also compared both these methods with the Simple Randomized Kaczmarz (SRK) method. In this method, rows are sampled using a uniform probability distribution. Schmidt [7] proved the convergence rate for this method and showed that RK should be at least as fast as SRK.

F. Sampling rows using quasirandom numbers

We previously mentioned that using consecutive rows from highly coherent matrices can make convergence slow and that sampling rows in a random fashion can improve the rate of convergence. However, when sampling random numbers, these can form clumps, meaning that consecutively sampled rows can have small angles between them. This is where quasirandom numbers come in: they are sequences of numbers that are evenly distributed and have been shown to improve the convergence rate of Monte-Carlo-based methods. Several low-discrepancy sequences can be used to generate quasirandom. Here we will work with two sequences that are widely used: the Halton sequence [8] and the Sobol sequence [9].

III. The Revival of Row and Column Action Methods

The work by Strohmer and Vershynin in [5] motivated other developments in row/column action methods by randomizing classical algorithms. In this section, we present several iterative methods, some of which are modifications of the RK method.

A. Randomized Coordinate Descent Method

Leventhal and Lewis [10] developed the 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}. RGS is a column-action method where the columns are sampled with probability proportional to their norms. Leventhal and Lewis also showed that, like RK, RGS converges linearly in expectation.

B. 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 therefore, are inconsistent. To extend the work developed by Strohmer and Vershynin [5] to inconsistent systems, Zouzias and Freris [11] introduced the Randomized Extended Kaczmarz (REK) method. This method is a mixture of a row and column action method since, in each iteration, we use one row and one column of matrix A, and it converges linearly in expectation to the least-squares solution, x_{LS}. Rows are chosen with probability proportional to the row norms and columns are chosen with probability proportional to column norms.

C. Greedy Randomized Kaczmarz Method

The Greedy Randomized Kaczmarz (GRK) method introduced by Bai and Wu [12] 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. The Greedy Randomized Kaczmarz method presents a faster convergence rate when compared to the Randomized Kaczmarz method, meaning that it is excepted for GRK to outperform RK.

D. Selectable Set Randomized Kaczmarz Method

Just like the Greedy Randomized Kaczmarz method, the Selectable Set Randomized Kaczmarz (SSRK) method [13] 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 aren't yet solved is referred to as the selectable set, which is updated in each iteration. There are two ways to update the selectable set, described by the variations of the SSRK method known as the Non-Repetitive Selectable Set Randomized Kaczmarz (NSSRK) method and the Gramian Selectable Set Randomized Kaczmarz (GSSRK) method. In the NSSRK method, any row can be chosen except the row that was used in the previous iteration. In the GSSRK, rows that are orthogonal to the rows used in the previous iteration should not be chosen in the current iteration, which can be verified by checking null entries of the Gramian matrix $G = A^T A$.

E. 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 [14] was introduced by Moorman et al. 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

$$x^{(k+1)} = x^{(k)} + \frac{1}{q} \sum_{i \in \tau_k} w_i \frac{b_i - \langle A^{(i)}, x^{(k)} \rangle}{\|A^{(i)}\|_2^2} A^{(i)T} \quad (3)$$

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. Furthermore, that decrease is proportional to the number of threads. For the special case of uniform row weights, that is $w_i = \alpha$, the authors give some insight into how to choose α to improve convergence. For a consistent system, the optimal value for α is:

$$\alpha^* = \begin{cases} \dfrac{q}{1 + (q-1)s_{min}}, & s_{max} - s_{min} \leq \dfrac{1}{q-1} \\ \dfrac{2q}{1 + (q-1)(s_{min} + s_{max})}, & s_{max} - s_{min} > \dfrac{1}{q-1} \end{cases} \quad (4)$$

where $s_{min} = \sigma_{min}^2(A)/\|A\|_F^2$ and $s_{max} = \sigma_{max}^2(A)/\|A\|_F^2$. 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.

IV. Sequential Version

In this section, we evaluate the performance of sequential implementations of the Kaczmarz method and some of its variants. Section IV-A presents the implementation details for the considered variants of the Kaczmarz method, as well as a description of the experimental setup. Section IV-B discusses the obtained experimental results.

A. Implementation

We implemented all the sequential methods previously discussed. Furthermore, since several authors have found that sampling without replacement can be an effective row selection criterion [15], we also developed the Simple Randomized Kaczmarz Without Replacement method (SRKWOR). In sampling without replacement, after an observation is selected, it cannot be selected again. This can be accomplished by working with an array of row indices that is shuffled during pre-processing.

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 Conjugate Gradient (CG) and Conjugate Gradient for Least-Squares (CGLS) methods from the Eigen [1] linear algebra library.

Simulations were implemented in the C++ programming language; its source code and corresponding documentation are publicly available [2]. All experiments were carried out on the Accelerates Cluster [3]. 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.

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:

- 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

[1] https://eigen.tuxfamily.org/index.php?title=Main_Page
[2] Code available here: https://github.com/inesalfe/Thesis-Kaczmarz.git
[3] http://epp.tecnico.ulisboa.pt/accelerates/

$r = b - Ax^{(k)}$ 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 the computation of the residual in the second condition is a very computationally expensive task. 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 the first condition of the stopping criteria is only verified every 1000 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. This can be accomplished by using $\varepsilon_1 = 10^{-25}$ and $\varepsilon_2 = 10^{-10}$.

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.

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. Similarly to the previous case, we compute the number of iterations for RK, CG, and CGLS to reach the error upper bound $\|x^{(k)} - x_{LS}\|^2 < \epsilon_3 = 10^{-10}$ and then measure the execution time of those iterations.

2) Datasets

The Kaczmarz method and its variants were tested for dense matrices using artificial datasets generated using C++. Two main datasets were generated: one with contrasting row norms and a second one with coherent rows. The goal of the first dataset was to evaluate how different row selection criteria perform against matrices with contrasting row norms. The goal of the second dataset was to compare the randomized versions with the original version of the Kaczmarz method for systems with coherent rows.

In the first dataset 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. 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 goal of the second dataset is to have coherent rows. This can be achieved by having consecutive rows with few changes between them. Entries are sampled from a normal distribution $N(2, 20)$ but each two consecutive rows only differ in 5 elements.

Finally, a dataset 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 dataset sampled from a normal distribution $N(0, 1)$.

3) The 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 10 times with different seeds, and the solution x is calculated as the average of the outputs from those runs.

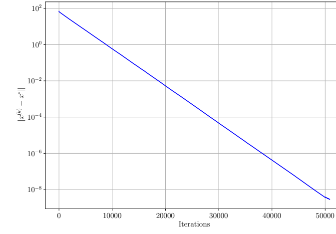


Fig. 1. Error as a function of the number of iterations for an 80000×1000 overdetermined system for the Randomized Kaczmarz method.

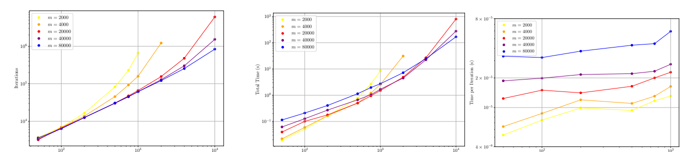
B. Results

C. Randomized Kaczmarz Algorithm

The simulations for the RK method in this section used the first dataset. We start with the analysis of the method's convergence, shown in Figure 1. 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.

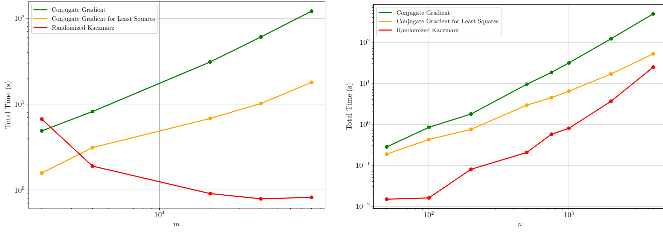
The following analysis focuses on the number of iterations and execution time as functions of the number of rows, m, and columns, n. From Figure 2a it is clear that the number of iterations increases with n. Increasing n while maintaining m makes systems harder to solve since there are more variables for the same number of restrictions. However, for a given value of n, there isn't 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 2b 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 2c. 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 excepted 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 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.

To finish the analysis of RK, we show how this method can outperform CG and CGLS. From both Figures 3a and 3b 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. From Figure 3a we can see that, except for the smallest system, RK is faster than CGLS. Figure 3b shows that, although RK is faster for the matrix dimensions shown in the plot, the difference between this method and CGLS decreases when n increases, that is, when m and



(a) Number of iterations. (b) Total execution time. (c) Execution time per iteration.

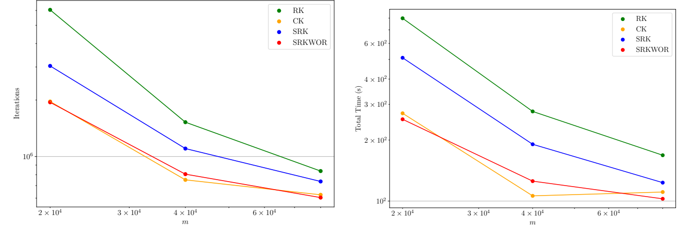
Fig. 2. Results for the Randomized Kaczmarz algorithm using a fixed number of rows and a varying number of columns.



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

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

Fig. 3. Comparison between RK, CG and CGLS for overdetermined systems.



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

(b) Execution time until for systems with $n = 10000$.

Fig. 5. Results for some variants of the Kaczmarz algorithm for systems using a fixed number of columns and a varying number of rows.

n get closer. In summary, RK should be used to the detriment of CG and/or CGLS for very overdetermined systems.

1) 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 dataset using a fixed column number. We start with the methods introduced in Section III that, like RK, select rows based on their norms. Figure 4a 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 4a, 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. The RK, GSSRK, and NSSRK methods have an indistinguishable number of iterations. In terms of time (Figure 4b), 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.

When Strohmer and Vershynin in [5] 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 5. 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, exhibits a lower number of iterations and time than the RK method, which shows that, for this dataset, 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 in iterations and time. These results show that sampling without replacement is indeed an efficient way to choose rows.

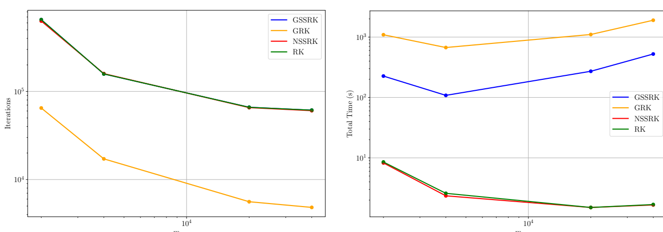
Contrary to the intuition provided by Strohmer and Vershynin [5], the RK inspired method from Figure 5 does not outperform CK. As Wallace and Sekmen [16] refer, choosing rows in a random way should only outperform the CK method for matrices where the angle between consecutive rows is very small, that is, highly coherent matrices. To confirm this we also compare the results of the methods used in Figure 5 for the second dataset. Figure 6 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. So far, it seems that, for the first dataset, 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 7, 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 datasets 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.

D. Variants of the Kaczmarz Algorithm for Least-Squares Systems

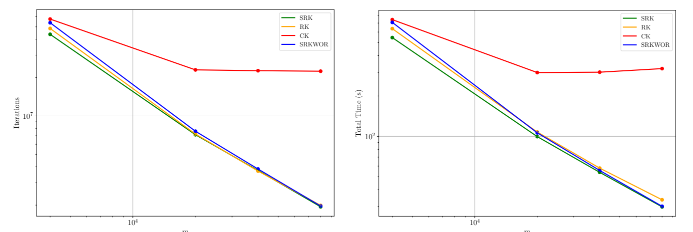
In this section, we discuss the results for the REK and RGS methods. We will also show that, for the dataset that we generated, other benchmark methods are faster. In Section IV-C, we compared the RK method with CG and CGLS and concluded that the CGLS is faster than the CG method for overdetermined systems. For this reason, we will compare REK and RGS with CGLS. The execution



(a) Number of iterations.

(b) Execution time.

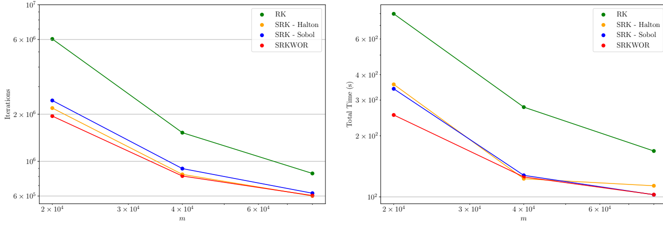
Fig. 4. Results for some variants of the Kaczmarz algorithm for systems using a fixed number of columns $n = 1000$ and a varying number of rows.



(a) Number of iterations.

(b) Execution time.

Fig. 6. Results for some variants of the Kaczmarz algorithm for the coherent dataset using a fixed number of columns $n = 1000$ 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.

Fig. 7. Results for some variants of the Kaczmarz algorithm for systems using a fixed number of columns $n = 10000$ and a varying number of rows.

time of the three methods, shown in Figure 8, 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.

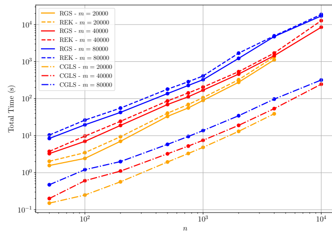


Fig. 8. Execution time for REK, RGS and CGLS for inconsistent systems using a fixed number of rows and a varying number of columns.

V. Parallel Implementations Using Shared Memory

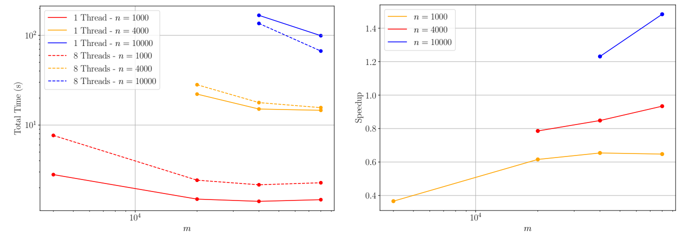
It is not a trivial task to parallelize the Kaczmarz algorithm, as it is an iterative algorithm where each iteration depends on the previous one. There are, nonetheless, two main strategies for the parallelization of iterative algorithms: block-sequential, where we parallelize the work inside each iteration; and block-parallel, where several iterations are distributed and computed in parallel, after which the results are combined. In this section, we present several shared memory approaches to the parallelization of RK using both block-sequential and block-parallel strategies.

The implementations of the several methods were accomplished using the OpenMP C++ API. All the experiments and results reported throughout this section were conducted on the Accelerates Cluster, also used for the sequential simulations. Regarding the datasets that were used, the simulations in Sections V-A to V-C used consistent systems taken from the first dataset discussed in Section IV-A2. Only for Section V-D, did we use inconsistent systems from the least-squares problems dataset.

A. Parallelization of Each Iteration

In a first attempt at parallelization, we show that using a block-sequential approach that parallelizes the work inside each iteration doesn't always exhibit speedup and, when it does, it is far from ideal, which was somewhat expected due to the small workload per iteration.

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. 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.



(a) Execution time.

(b) Speedup.

Fig. 9. Results for the parallel implementation of RK for 8 threads using a block-sequential approach.

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 implementation of the parallelization of RK was run for 8 threads. Figure 9 shows the execution time and speedup, where speedup is defined as the quotient of the sequential execution time and the parallel time. 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 and that they increase with the number of columns, which was anticipated since increasing the workload per iteration attenuates the overhead of parallelization. Although we don't show here the results, the speedups for 2 and 4 threads are also not ideal.

B. Randomized Kaczmarz with Averaging

We now move on to block-parallel implementations with the RKA algorithm. 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) such that

$$x^{(k+1)} = x^{(k)} + \frac{\alpha}{q} \sum_{i \in \tau_k} \frac{b_i - \langle A^{(i)}, x^{(k)} \rangle}{\|A^{(i)}\|_2^2} A^{(i)T} . \quad (5)$$

where q is the number of threads. Using the previous expression, we implemented a sequential version of RKA so that we could validate the results obtained by the authors of RKA and evaluate the effectiveness of the parallelization. We will now go through the details of the computations involved in each iteration of the parallel implementation of RKA, presented in Algorithm 1.

In lines 3 and 4 of Algorithm 1 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 1 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.

In lines 7 to 9 of Algorithm 1 the results are combined. To ensure that all threads update the estimate of the solution and that no two

Algorithm 1 Pseudocode for an iteration of the parallel implementation of RKA. $A_i^{(row)}$ 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.

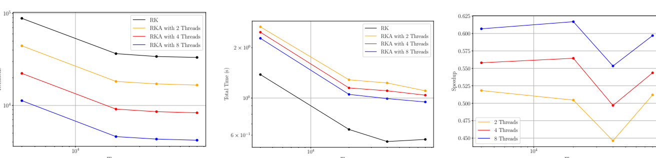
1: $it \leftarrow it + 1$
2: **OMP barrier**
3: **OMP for** $i = 0, ..., N$ **do**
4: $\quad x_i^{(prev)} \leftarrow x_i$
5: $\quad row \leftarrow$ sampled from $\mathcal{D}$
6: $\quad scale \leftarrow \alpha \times \dfrac{b_{row} - \langle A^{(row)}, x^{(prev)} \rangle}{q \, \|A^{(row)}\|_2^2}$
7: **OMP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

threads are updating x simultaneously, a critical section must be created, meaning that the gathering of results is done sequentially. 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 must be done sequentially.

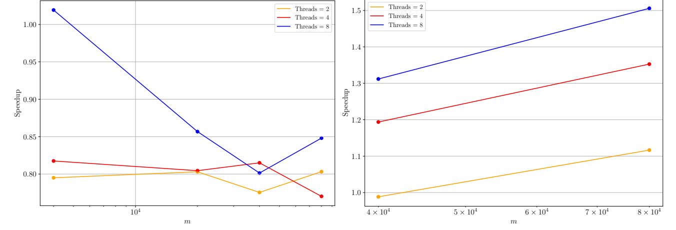
As in Section V-A, we do not consider the stopping criteria when measuring execution times. For that, we measure the necessary number of iterations to reach convergence by running the sequential and parallel algorithms until $\|x^{(k)} - x^*\| < 10^{-5}$ is verified. We then measure the execution time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

We now discuss the results for the RKA method. We will use the optimal values for parameter α given by (4). The sequential and parallel versions of RKA were tested for 2, 4, and 8 threads, whose results are shown in Figure 10. From Figure 10a it is clear that 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. Although Figure 10a shows that RKA requires fewer iterations to converge than RK, Figure 10b 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 1). 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 10c represents the speedup computed as the quotient between the execution time of RK and the execution



(a) Number of iterations. (b) Execution time. (c) Speedup.

Fig. 10. 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.



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

Fig. 11. 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.

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 11, 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 datasets 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.

C. Randomized Kaczmarz with Averaging with Blocks

Efficient parallelization of RKA isn't 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 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. Just like for RKA, we developed a sequential version of RKAB so that we can evaluate the effectiveness of the parallelization. Here, we discuss a detailed explanation of the work inside a single iteration of the parallel implementation of RKAB presented in Algorithm 2.

In Algorithm 2, instead of saving the previous iteration (Algorithm 1), 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, x, to calculate the scale factor, needed to compute $x^{(thread)}$ (lines 3 to 6 of Algorithm 2). 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 2). 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. Note that we need the barrier in line 14 of Algorithm 2 so that we don't have one thread updating x in the critical region while another thread is behind computing $x^{(thread)}$ in line 13.

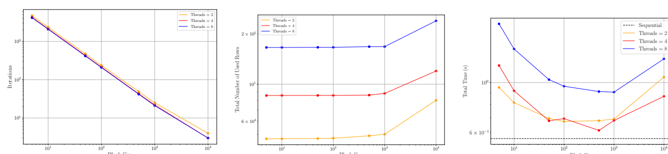
The stopping criterion will be the same as the one used for RKA. During the implementation of RKA we used the optimal value for the row weights, α^*. However, since RKAB is a different algorithm than RKA and there is no calculated optimal value for α, we will use $\alpha = 1$.

We will start by analyzing how the execution time of RKAB depends on the size of the blocks by setting $block\ size$ $=$

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

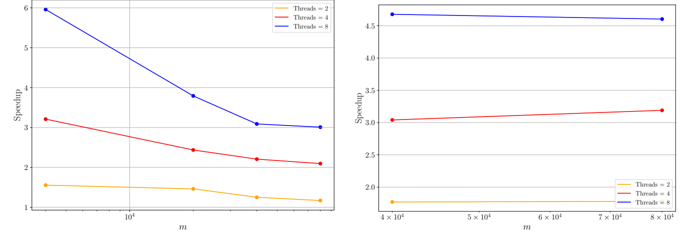
1: $it \leftarrow it + 1$
2: OMP **barrier**
3: $row \leftarrow$ sampled from $\mathcal{D}$
4: $scale \leftarrow \alpha \times \dfrac{b_{row} - \langle A^{(row)}, x\rangle}{\|A^{(row)}\|_2^2}$
5: **for** $i = 0, ..., N$ **do**
6: $\quad x_i^{(thread)} \leftarrow x_i + scale \times A_i^{(row)}$
7: **for**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

$\{5, 10, 50, 100, 500, 1000, 10000\}$. Similarly to the previous parallelization attempts, the sequential and parallel versions of RKAB were tested for 1, 2, 4, and 8 threads. Figure 12 shows the results for RKAB for a system with dimensions 80000×1000. Figure 12a 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 12b, increases when more threads are used. Figure 12b also shows that, for a given number of threads, regardless of *block size*, the total number of used rows stays the same (with the exception of the *block size* $= 10000$). The results in Figure 12c 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 big 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 12b), the number of times that the threads have to communicate to average the results is smaller. The larger block size is special 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, meaning 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 also close between themselves, meaning that there is little to no benefit in averaging similar results. 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.



(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.

Fig. 12. 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) Systems with $n = 1000$. (b) Systems with $n = 10000$.

Fig. 13. 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.

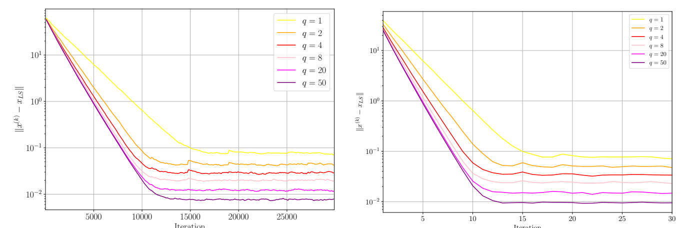
We will now evaluate the parallelization of the RKAB algorithm by comparing it with its sequential version. Figure 13, contains the speedup calculated as the quotient between the total execution time of the sequential and parallel implementations of RKAB for several threads using *block size* $= n$. The results show much higher speedups than the ones obtained with RKA (see Figure 13).

In conclusion, although the parallelization of the RKAB can be done more effectively than the parallelization of RKA, the RKAB method, just like RKA, is not faster than the sequential RK for most systems in our dataset.

D. Application of RKA and RKAB to inconsistent systems

In Sections V-B and V-C 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. In this section, we show that RKAB is also able to achieve this. To show how the convergence horizon can change for several numbers of threads we will show the evolution of the error, $\|x^{(k)} - x_{LS}\|$ for RKA and RKAB using $\alpha = 1$. The inconsistent system used in this section has dimensions 80000×1000 and was taken from the least-squares dataset that was generated to test REK and RGS.

Figure 14 shows the error evolution for RKA and RKAB. Note that Figures 14a and 14b are very similar: using a higher number of threads, q, the error value around which the method stabilizes



(a) Error for RKA. (b) Error for RKAB using *block size* $= 1000$.

Fig. 14. Results for RKA and RKAB (with $\alpha = 1$) for a system 80000×1000. Here we show the first 30000 iterations and the error was stored every 100 iterations.

decreases. Furthermore, the values around the error stabilizes are similar for both methods, meaning that the RKAB method, like RKA, can decrease the convergence horizon.

VI. Parallel Implementations Using Distributed Memory

In the previous section, 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, 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.

In this section, we implement and discuss the results for the RKA and RKAB methods using distributed memory. The implementations using distributed memory were accomplished using the C++ API MPI [4]. 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 dataset and execution times correspond to the total time of 10 runs of the algorithm.

A. Distributed Memory Implementation of the RKA Method

In this section, we discuss the implementation and results for RKA for distributed memory.

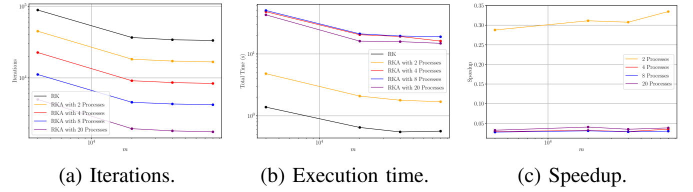
Since one of the advantages of using distributed memory is to be able to process data sets that are not able to be stored in a single machine, in this implementation we divide matrix A and vector b between the several available machines. We developed a new sequential implementation of RKA that simulates the partition of the system amongst the several processes.

The implementation of the parallel implementation of RKA for distributed memory is much simpler than that for shared memmory. 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 1). The averaging of the results was accomplished with the *Allreduce* command with the sum operation.

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, simulations were run with two MPI processes per node.

We now discuss the results of the RKA method by comparing it with the sequential RK present in Figure 15. Figure 15a shows similar results to the ones using shared memory (Figure 10a): increasing the number of MPI processes decreases the number of iterations. However, the results for execution time in Figure 15b 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

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



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

Fig. 15. 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 (4).

time. As a consequence, the speedups in Figure 15c, 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 10c. In conclusion, using distributed memory only, RKA cannot be efficiently implemented.

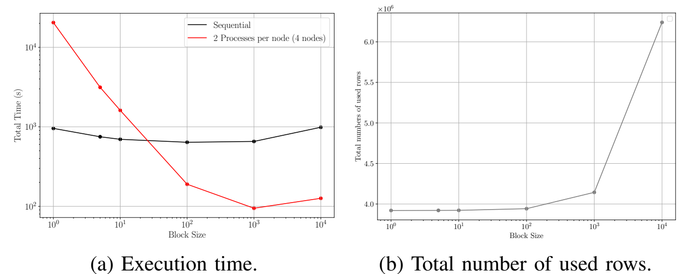
B. Distributed Memory Implementation of the RKAB Method

In this section, we discuss the implementation and results for RKAB for distributed memory. Similarly to the implementation of RKA (Section VI-A), the system will be partitioned between the several processes.

Once more, the implementation of the parallel implementation of RKAB for distributed memory is much simpler than that for shared memory (Algorithm 2) since we do not need the extra variable $x^{(thread)}$ and, just like for RKA, the averaging of the results was accomplished with the *Allreduce* command with the sum operation.

During the analyses of RKAB in Section V-C, 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*.

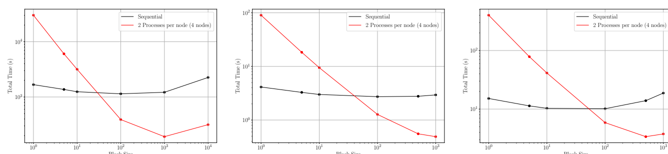
The stopping criterion is the same as the one chosen for RKA. The value of the row weights was chosen as $\alpha = 1$ (see Section V-C) and we used the configuration of two processes per node. The results for 8 MPI processes for a system 80000×10000 for several values of *block size* are shown in Figure 16. Firstly, contrary to the results for 8 threads using OpenMP (Figure 12c), 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 16b where we see that the number of total used rows has a larger increase from *block size* $= 1000$



(a) Execution time. (b) Total number of used rows.

Fig. 16. Results for RKAB using 8 MPI processes for a system with dimensions 80000×10000. We also represent the execution time for the sequential RKAB.

to *block size* = 10000. This means that the increase in the total amount of work needed for the algorithm to converge when 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.



(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.

Fig. 17. Results for RKAB using 8 MPI processes for three systems. We also represent the execution time for the sequential RKAB.

Figure 17 shows the execution time for 8 processes for systems with other dimensions. Note that the system in Figure 17a has the same number of columns as the system in Figure 16a and both show an optimal block size of 1000. However, although the systems in Figures 17b and 17c 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 17b and 17c 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.

VII. Conclusion

A. 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 RKlike 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 the cost of communication does not make up for the decrease in the number of iterations in comparison with the RK method. 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.

B. Future Work

In Section V-C, we analyzed how RKAB behaves for unitary uniform row weights ($\alpha = 1$) 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 V-D, we showed that using $\alpha = 1$ can decrease the convergence horizon proportionally to the number of threads. It would be interesting to find an optimal value for α for inconsistent systems, just like α^* for consistent systems.

We concluded in Section VI-B that the optimal block size for RKAB 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.

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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