ANKARA YILDIRIM BEYAZIT UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

Predicting Sparse Linear Systems Partition Number using Machine Learning

M.Sc. Thesis by

Mohamed Abdiaziz Hassan

Department of Computer Engineering

July, 2024

ANKARA

Predicting Sparse Linear Systems Partition Number using Machine Learning

A Thesis Submitted to the

Graduate School of Natural And Applied Sciences of

Ankara Yildirim Beyazit University

In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering, Department of Computer Engineering

by

Mohamed Abdiaziz Hassan

July, 2024

ANKARA

M.Sc. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**Predicting Sparse Linear Systems Partition Number using Machine Learning**" completed by **MOHAMED ABDIAZIZ HASSAN** under supervision of **ASSIST. PROF. DR. FAHREDDİN ŞÜKRÜ TORUN** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Fahreddin Şükrü TORUN

Supervisor

Assoc. Prof. Dr. Fatih NAR

Jury Member

Assist. Prof. Dr. Mucahid KUTLU

Jury Member

Prof. Dr. Sadettin ORHAN

Director

Graduate School of Natural and Applied Sciences

ETHICAL DECLARATION

I hereby declare that, in this thesis which has been prepared in accordance with the Thesis Writing Manual of Graduate School of Natural and Applied Sciences,

- All data, information and documents are obtained in the framework of academic and ethical rules,
- All information, documents and assessments are presented in accordance with scientific ethics and morals,
- All the materials that have been utilized are fully cited and referenced,
- No change has been made on the utilized materials,
- All the works presented are original,

and in any contrary case of above statements, I accept to renounce all my legal rights.

Date: 2024, 3 July **Signature:** _____

Name & Surname: Mohamed Abdiaziz Hassan

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Assist. Prof. Dr. Fahreddin Şükrü Torun, for giving me the incredible opportunity to work with him on this project. His tremendous support and unwavering motivation were invaluable throughout my studies. His valuable recommendations and deep understanding constituted the milestones of this study, guiding me at every step of the way.

I am also profoundly grateful to Assoc. Prof. Dr. Fatih Nar for his invaluable support and assistance with the machine learning problems we encountered. His expertise and guidance have been instrumental in overcoming numerous challenges and in advancing the quality of this research.

I would like to extend my sincere gratitude to Assist. Prof. Dr. Mücahid Kutlu for his valuable feedback on my thesis. His insightful comments and suggestions significantly improved the quality of my work.

Finally, I would like to extend my heartfelt thanks to my parents for their constant love, encouragement, and support. Their unwavering belief in me was a source of strength and inspiration throughout this journey.

July, 2024 **Mohamed Abdiaziz Hassan**

Predicting Sparse Linear Systems Partition Number using Machine Learning

ABSTRACT

This thesis proposes a machine learning approach to solve sparse linear systems, a common problem in science and engineering. Our method uses machine learning to predict the optimal number of partitions needed for the block cimmino method. Traditional methods typically rely on trial and error or choosing the maximum number of cores as partitions to determine the best number of blocks. However, using a trained machine learning model eliminates this process. In this work, we present two models: one predicts two partition numbers with an AUC score of 89%, and the other predicts multiple partition numbers with an AUC score of 76% for the block cimmino method. We achieve this by using previously identified features in the literature and applying them in a novel context. We trained and tested our models using a diverse set of matrices, incorporating feature selection, demonstrating the effectiveness of leveraging established features in new applications and the robustness of our model in addressing the challenges of partitioning sparse linear systems. Our proposed method is faster and more effective than traditional methods.

Keywords: Machine learning, Sparse linear systems, Scientific Computing, block cimmino.

Makine Öğrenmesi Kullanarak Seyrek Doğrusal Sistemler için Bölüm Sayısını Tahmin Etme

ÖZ

Bu tez, bilim ve mühendislikte yaygın bir problem olan seyrek doğrusal sistemleri çözmek için bir makine öğrenimi yaklaşımı önermektedir. Yöntemimiz, blok Cimmino yöntemi için gereken optimal bölümlerin sayısını tahmin etmek amacıyla makine öğrenimini kullanmaktadır. Geleneksel yöntemler genellikle deneme yanılma yöntemine veya en iyi blok sayısını belirlemek için maksimum çekirdek sayısını bölüm olarak seçmeye dayanır. Ancak, eğitimli bir makine öğrenimi modeli kullanmak bu süreci ortadan kaldırır. Bu çalışmada, iki model sunuyoruz: biri, blok Cimmino yöntemi için AUC skoru %89 olan iki bölüm sayısını, diğeri ise AUC skoru %76 olan birden fazla bölüm sayısını tahmin ediyor. Bunu, literatürde daha önce tanımlanmış özellikleri kullanarak ve bunları yeni bir bağlamda uygulayarak başarıyoruz. Modellerimizi çeşitli matris setlerini kullanarak eğittik ve test ettik, özellik seçimini de dahil ederek, yerleşik özelliklerin yeni uygulamalarda kullanılmasının etkinliğini ve modelimizin seyrek doğrusal sistemlerin bölümlenmesi zorluklarını ele almaktaki sağlamlığını gösterdik. Önerdiğimiz yöntem, geleneksel yöntemlerden önemli ölçüde daha hızlı ve daha etkilidir.

Anahtar kelimeler: Makine öğrenimi, Seyrek doğrusal sistemler, Bilimsel Hesaplama, blok Cimmino.

CONTENTS

NOMENCLATURE

Acronyms

ABCD	Augmented Block Cimmino Distributed
ADASYN	Adaptive Synthetic Sampling
AUC	Area Under the Curve
BiCGStab	Biconjugate Gradient Stabilized
GMRES	Generalized Minimal Residual
IDR	Induced Dimension Reduction
KNN	K-Nearest Neighbors
LSQR	Least Squares QR
ML	Machine Learning
PCA	Principal Component Analysis
PETSc	Portable, Extensible Toolkit for Scientific Computation
SMOTE	Synthetic Minority Over-sampling Technique
WKNN	Weighted K-Nearest Neighbors

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Sparse linear systems are encountered in various fields, including network theory [1], data analysis [2], structural analysis [3], and chemical engineering [4]. Efficiently solving these systems is essential for understanding complex networks, processing high-dimensional data, predicting the behavior of physical structures, and optimizing chemical processes. Therefore, robust and efficient numerical methods are needed to solve these systems. These systems are characterized by linear equations with a sparse coefficient matrix, meaning most of its elements are zero. The relationship between variables in these systems is often expressed as

$$Ax = b \tag{1.1}$$

where A is the coefficient matrix, x is the vector of unknowns, and b is the vector of constants. When A is a sparse matrix, we refer to the system as a linear sparse system.

When working with linear sparse systems, different methods are used based on the system's features. Direct solvers, like LU and Cholesky, give exact solutions in a fixed number of steps, but they can require a lot of resources for very large systems. Iterative solvers, like Conjugate Gradient and GMRES, start with an initial guess and improve it step by step. They use less memory and work well for big systems but may need preconditioning to converge faster. Preconditioning changes the original linear system using a matrix that approximates the inverse of the original matrix, leading to faster convergence of the iterative method. Hybrid solvers use both direct and iterative methods. They use direct methods for some parts of the system and iterative methods for others. This approach balances efficiency and accuracy, especially in complicated systems where direct or iterative methods alone are not ideal.

The block cimmino method [5–9] is a widely recognized and highly effective approach for solving sparse linear systems. The method employs a hybrid row-block projection approach, where a direct solution can be utilized in each iteration to obtain

the projections. In the block cimmino method, the linear system **1.1** is partitioned into p block rows, where $p \leq n$, as follows:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} \tag{1.2}$$

Algorithm 1 outlines the steps of the classical block Cimmino method. This algorithm can be parallelized easily since independent underdetermined linear systems (at line 4 of Algorithm 1) can be solved by performing row block projections in each processor. Only summing up those partial solutions requires communication among processors. Finally the solution is updated in the last step of the algorithm.

Algorithm 1 Block Cimmino Method

1: Choose $x^{(0)}$
2: **while** t=0,1, 2,. . . . , until Convergence **do**
3: **for** $i = 1$ to p **do**
4: $\delta_i = A_i^+(b_i - A_i x^{(t)})$
5: **end for**
6: $x^{(t+1)} = x^{(t)} + w \sum_{i=1}^{p} \delta_i$
7: **end while**

The efficiency of partitioning methods is significantly influenced by the number of blocks used. While a larger number of blocks can enhance parallelization and speed up computation, it can also worsen the conditioning of the iteration matrix. This complication makes convergence more difficult and requires more iterations. Therefore, determining the optimal number of blocks is crucial for achieving the best parallel performance. Determining the optimal number of partitions is complex and depends on the problem's specifics, the system's characteristics, and the available computational resources.

There are two common methods for selecting the number of partitions. The first method is the trial-and-error method, which relies on a time-consuming process to determine the ideal number of partitions. The second method is straightforward and selects the maximum number of cores as the number of partitions. Both methods have limitations

and are not entirely reliable. In response to these limitations, we propose leveraging machine learning to accurately predict the optimal number of partitions for efficiently solving sparse linear systems. By analyzing the details of each problem and training a machine-learning model on various examples, we can make precise predictions. This approach accelerates the solution process and identifies crucial features that impact results, providing a quicker and more effective way to solve sparse linear systems in science and engineering. Machine learning is increasingly recognized as a powerful tool for solving various scientific and engineering problems.

Table 1.1 Matrix and solution time

Matrix	**Kind**	**Solution Time (p=4)**	**Solution Time (p=112)**
nxp1	Circuit Simulation Problem	**135.9 s**	3,682.7 s
ASIC_320k	Circuit Simulation Problem	67.7 s	**4.3 s**

Table 1.1 shows that for the matrix nxp1, using 4 partitions was 27.1 times faster than using 112 partitions. For the matrix ASIC_320k, using 112 partitions was 15.5 times faster than using 4 partitions. Thus, selecting the optimal number of partitions is crucial for achieving maximum parallel efficiency.

This study introduces machine learning models to predict partition numbers for the parallel block cimmino method by leveraging features previously identified in the literature and applying them in a new context. Through extensive analysis and careful selection of the most significant features, our approach achieved an impressive test Area Under the Curve (AUC) of 89% for the model that predicts two partition numbers and 76% for the model that predicts multiple partition numbers. This demonstrates the effectiveness of using established features in new applications and the robustness of our models in addressing the challenges of partitioning sparse linear systems.

CHAPTER 2

LITERATURE REVIEW

Funk, Götz, and Anzt [10] have proposed a deep learning approach to predict the best iterative solver for sparse linear systems. Their method uses a fully connected feedforward neural network trained on various features which include: matrix density, symmetry and non-zero element distribution. Their neural network is optimized by a genetic algorithm which achieves a result of 60% top-1 and 82% top-3 accuracy. BiCGStab, GMRES and IDR are the top 3 main solvers, which solve more than 80% of the problems. The researchers also have suggested that expanding on the feature set and preconditioning techniques help to improve the accuracy of the predictions. However, it's important to understand that while this is study seems to show great potential the simplicity of the feature set and the focus on Krylov solvers limits the generality.

In [11] Sood had developed a machine learning model that was able to predict the optimal iterative solver for sparse linear systems using Portable, Extensible Toolkit for Scientific Computation (PETSc). This was done by extracting features from the linear systems and training the classifiers with algorithms such as Bayesian networks, decision trees, k-nearest neighbors and support vector machines. To add to this, the study managed to achieve an 87% accuracy. Also, the best solver-preconditioner combinations were: Least Squares QR (LSQR) with Jacobi, BiCG with ASM (0) and BCGS with ILU (0). Overall, the study is quite extraordinary and its focus on PETSc and the computational cost of some features might limit its applicability and real-time usage.

Nisa et al. have investigated the use of machine learning to optimize Sparse matrix vector multiplication (SpMV) on GPUs by predicting the best sparse matrix format and execution times [12]. investigation indicates that they use: XGBoost, SVM, MLP, and decision trees to train models on features extracted from over 2300 matrices. Plus, it demonstrates that the format selection accuracy is very high (up to 88%) and it has a low relative mean error ($\approx 10\%$) in performance prediction. Thus, it demonstrates that a minimal set of features is more than enough for accurate predictions and the performance

is fairly consistent across different GPU architectures and precision levels. Although they get high accuracy and efficiency, it is crucial to remember that the computational cost of feature extraction and the need for broader dataset evaluation as potential areas for improvement in the future.

Dufrechou, Ezzatti and Quintana-Ortí offered their suggestion on a machine learning approach to select the best sparse triangular solver for linear systems on GPUs, to reduce runtime based on matrix properties [13]. They done this by using features like matrix size, number of nonzeros, and arithmetic precision and classification algorithms like Decision Trees, Weighted K-Nearest Neighbors (KNN), and Bagged Trees, which managed to achieve over 80% accuracy. The Bagged Trees demonstrated the best accuracy and generalization. Plus the results showed, that machine learning can effectively predict the best solver for sparse triangular linear systems and can reduce runtime. However, it should be noted that the cost of some features and the dataset used may not allow for real-time application and generality.

Tang, Zhang and Chen [14]. have suggested using graph neural networks (GNNs) to automate the selection of optimal preconditioners and Krylov solvers for large sparse linear systems, a process that at the moment currently relies on expert knowledge and a lot of trial and error. the representing sparse matrices as graphs and combining node and graph features through the use of graph convolutions, this study evaluates four models which include: Graph Attention Networks (GATs), Graph Isomorphism Networks (GINEs), Graph Convolutional Networks (GCNs) and GraphSAGE. These four models process features that are cheap to compute, therefore balancing effectiveness and cost. To be more specific, the GNN models outperform traditional machine learning methods by at least 25% in the NDCG metric of the best traditional method. Additionally, the GNN models are accurate and efficient in predicting the best solver-preconditioner combinations. Also, the computational overhead of GNN inference is found to be justified compared to the solving time of the linear systems. Despite, the approach having a high performance and being innovative, the complexity might be a challenge for implementation and integration with existing systems, and the evaluation is specific to this dataset and might not generalize to other datasets.

Sood, Norris, and Jessup present a performance modeling approach for parallel preconditioned Krylov methods which are used for solving large sparse linear systems in scientific and engineering computations [15]. This study goes into detail by evaluating the scalability of these methods using a combination of machine learning and analytical communication-based models to predict the best solver-preconditioner combinations for different problem sizes without the need for extensive empirical measurements. Furthermore, this approach utilizes machine learning to be able to classify solvers based on their convergence behavior with an analytical model to rank solvers according to their communication costs using features such as matrix size, non-zero elements and computational performance metrics. To add to this, the results show that this hybrid approach can predict the best solver-preconditioner combinations accurately and it provides a communication-based ranking that matches with empirical performance data. The strengths of this study are the novel combination of machine learning and analytical modeling, high accuracy of predictions and scalability across different problem sizes and processor counts.

Wei et al. [16] has recommended a machine learning approach to predict the best sparse general matrix-matrix multiplication (SpGEMM) algorithm for GPUs. This algorithm has been suggested to address the performance variability issue across a variety of different matrix shapes. So, by extracting features from input matrices and training models with LightGBM and XGBoost, they manage to gain a high accuracy with minimal inference overhead. Experiments on three GPUs (Tesla P100, Tesla V100, RTX 3090) show that this method outperforms state-of-the-art algorithms such as: cuSPARSE, NSparse, and spECK. These algorithms do this by both accuracy and speed, approximately at around 85% accuracy and significant performance improvements, especially with spECK and TileSpGEMM algorithms. This approach is useful because it reduces the need for matrix thumbnails and generalizes well across different GPU architectures, a big win for SpGEMM on GPUs.

Grementieri and Galeone [17] research proposes a neural sparse linear solution algorithm framework that can solve large sparse symmetric linear systems which have been traditionally encountered in scientific and engineering applications. Likewise,

the use of graph neural networks (GNNs) considering these systems are undirected weighted graphs, and it's not a new approach specifically permutation invariant and scale invariant. On top of that, the NSLS framework receives this graph which is sent through a graph convolution network (GCN). And, this is done until it is treated as a regression of nodes to approximate the solution. Similarly, adopting various characteristics enhancement techniques like diagonal, Jacobi Edge Augmentation, conjugate gradients for nodes with fixed patterns and end-to-end scale mechanisms helps to ensure accurate directional prediction. More importantly, NSLS models when applied to both synthetic datasets and real-world structural engineering data-coil, the results imply that there is great potential for GPU utilization and hardware independence. The research also shows that NSLS models can efficiently produce approximate solutions which is particularly beneficial for real-time simulations where exact accuracy is less critical, feature augmentations especially those based on iterative methods like the conjugate gradient can further improve model performance.

Kuefler and Chen address the application of reinforcement learning to choose the preconditioned solvers for sparse linear systems without the need for labeled data as in supervised learning. [18] Their method compares different preconditioners based on factors such as the running time to help improve performance. These include: equilibrating, reordering, preconditioning and solving the matrix. Furthermore, state representations are defined by matrix features such as bandwidth and norms. In addition to this, on 664 matrices, the system was tested, and the result was 81.56% success rate when the training and testing sets were different. However, the result was 8% success rate when the training and testing sets were the same and also 4% for a more diverse set. Moreover, the median time ratio for solved matrices was near optimal, which indicated the method's effectiveness and applicability in suggesting suitable preconditioned solvers.

In [19] Dufrechou, Ezzatti, Freire, and Quintana-Ortí, the authors proposed a machine learning-based method for selecting sparse triangular linear system (SpTRSV) solvers on GPUs. This is possible by building upon previous works such as refining the selection process to increase efficiency and accuracy in selecting the most suitable

solver for a specific matrix, to improve the performance in high-performing computing applications. In addition to this, the authors research is based on level-set and self-scheduled methods for sptrsv on GPUs and the machine learning model is trained with features such as matrix dimension, number of nonzeros and computational performance. The researchers compared several machine learning approaches to determine which were the most effective and applied data augmentation to expand the dataset. Moreover, heuristics were also suggested to estimate some costly features in order to improve the efficiency of the selection step. The findings show that the use of the proposed machine learning method leads to a considerable reduction in the time taken without a corresponding reduction in the accuracy of the results. Additionally, Decision Trees and wKNN were found to be the most accurate classifiers with up to 84% accuracy. To add to this, 6% of New solvers from the latest cuSparse library versions were also incorporated into the study, which helped to enhance the performance of the method in predicting the optimal sptrsv routine.

Chen et al. [20] tackles the problem of sparse signal recovery from noisy linear measurements in wireless communications using a deep learning architecture with adaptive depth for the sparse linear inverse problem. This is much better than fixed-depth neural networks because using the same depth for all tasks is not efficient when their sparsity differs. Also, the method shows that iterative algorithms in the form of neural networks have an adaptive halting score which allows the network to change the depth during inference based on the task complexity. Moreover, in the adaptive-depth DL architecture, the number of iterations is task-dependent and the architecture has been tested with synthetic data for real-world applications like random access in massive MTC and MIMO channel estimation. Also, the results show a significant gain in efficiency and performance. The results show that the proposed architecture has better reconstruction performance and fewer layers than fixed-depth DL methods, a more efficient solution for sparse linear inverse problems in communication systems.

CLA is a new technique presented by Elgohary et al. [21] for improving large-scale ML algorithm performance. To add to this, CLA uses lightweight database compression methods to compress matrices, which in return allows for efficient in-memory linear algebra operations on compressed data. This approach solves the problem of attaining both a high compression ratio and a fast decompression rate. The methodology involves column compression schemes and a sampling-based compression algorithm which are implemented within Apache SystemML for testing. The experiments show that CLA can achieve in-memory operation speeds close to the uncompressed matrix, while also at the same time cutting down the memory usage by up to 26 times. In contrast to general-purpose (Gzip, Snappy) and specialized (CSR-VI) compression methods it is more effective. Furthermore, CLA can be easily used by data scientists through its integration into Apache SystemML, and it brings significant performance enhancements to a range of ML algorithms including linear regression, logistic regression, and principal component analysis (PCA). And, the trade-off between the compression ratio and operation speed is optimized well in the research, also the proposed method yields compression ratios of 2x to 27x on real-world datasets.

CHAPTER 3

BACKGROUND

In this chapter, we will cover sparse linear systems, including how they are represented and the various methods used to solve them (such as direct, iterative, and hybrid methods). We will also introduce the basics of partitioning, describe the types of machine learning and algorithms used in this study, and provide an overview of the tools used.

3.1 Parallel Computing

Parallel computing is a technique of solving a problem by dividing it into two or more sub-problems and solving them in parallel. In traditional computing, only one operation is executed at a time in a step-by-step process. However, in parallel computing, a large problem is decomposed into several sub-problems to be solved at the same time. These parts are then worked on by different processors at the same time; these processors are also referred to as cores.

This approach can make computers solve problems much faster, especially where the problem is complicated. Let's say you have a big puzzle to solve. As a single person, it is time-consuming when one is working alone. But if many friends help you, each working on a different part, you finish much quicker. Parallel computing is similar to this in its operation.

There are several types of parallel computing. Some use many processors in a single computer, while others make use of a network of interconnected computers. Regardless of the method, the main goal is always the same: to solve problems faster by working on many parts at once.

The Message Passing Interface (MPI) is crucial in this context, providing a standardized system for communication among processes running on different nodes or cores. MPI supports efficient message exchange, process synchronization, and parallel I/O operations, making it essential for large-scale applications like scientific

simulations, machine learning, and data analysis. By enabling effective coordination and resource utilization, MPI ensures that the benefits of parallel computing—solving complex problems faster and more efficiently are fully realized.

3.2 Sparse Linear Systems

Sparse linear systems are a type of linear system where most of the elements in the matrix are zero. The basic form of a sparse linear system is

$$Ax = b \tag{3.1}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{3.2}$$

where A is the coefficient matrix, x is the vector of unknowns, and b is the vector of constants. When A is a sparse matrix, we refer to the system as a linear sparse system.

In a sparse matrix, the majority of its elements are zero. This means that we can store and work with these matrices more efficiently. For example, in a large $n \times n$ matrix, if only a small number of elements are non-zero, we don't need to waste space storing all the zero elements. Instead, we can just keep track of the non-zero elements and their positions.

Working with sparse matrices is faster because there are fewer elements to process. This is especially useful for very large systems where working with every single element would take too much time and memory. Sparse matrices often appear in problems where interactions or relationships are limited to a small number of elements.

Sparse matrices are created by storing only the non-zero elements and their positions, which significantly reduces memory usage compared to dense matrices. This process involves identifying and recording the non-zero elements and indexing their locations. Sparse matrices are crucial in various scientific and engineering applications, as they

enable faster computations and reduced memory overhead.

3.2.1 Formats

Regular dense matrices store every single value, even when most values are zero. This approach becomes inefficient for sparse linear systems, where the majority of elements are zero. To address this inefficiency, sparse matrix formats are used, which store only the nonzero elements along with the necessary information to access them efficiently. Among the specialized storage formats developed for this purpose, the most commonly used are the Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate List (COO).

- **Compressed Sparse Row (CSR) format:** compresses the rows of a sparse matrix. It uses three arrays: one for the values, one for the column indices of non-zero elements in each row, and another to store starting positions of each row's non-zeros in the value and column index arrays. It allows fast row access and efficient matrix-vector multiplication.
- **Compressed Sparse Column (CSC) format:** similar to CSR, but stores information by column instead of row. Useful for operations needing fast column access.
- **Coordinate List (COO) format:** it stores three arrays: one for the values, one for the row indices, and one for the column indices of each non-zero element. It's simple to implement but can be inefficient for memory and computations.

3.2.2 Solvers

When solving sparse linear systems, different solution methods can be used, each suited to a specific type of problem. These methods include direct solvers, iterative solvers, and hybrid solvers.

1. **Direct solvers** : aim to find the exact solution to the linear system $Ax = b$ in a finite number of steps. They involve factorizing the matrix A into simpler components that can be easily manipulated. Common direct solvers include LU

decomposition, which breaks down A into a lower triangular matrix L and an upper triangular matrix U, and Cholesky decomposition, which is used for symmetric positive-definite matrices, decomposing A into LL^T. Direct solvers are highly accurate and provide exact solutions within the limits of machine precision, but they can be computationally expensive and memory-intensive, especially for very large matrices.

2. **Iterative solvers**: generates a sequence of approximations to the solution. These methods are particularly useful for large-scale problems where direct methods become impractical due to their computational and memory requirements. Iterative solvers include methods like the Conjugate Gradient (CG) for symmetric positive-definite matrices, Generalized Minimal Residual (GMRES) for non-symmetric matrices, and Bi-Conjugate Gradient (BiCG) for general non-symmetric systems. These solvers are typically less memory-intensive and can handle much larger systems than direct solvers, though their convergence rate can be highly dependent on the properties of the matrix, such as its condition number.

3. **Hybrid solvers**: combine the strengths of both direct and iterative methods to achieve better performance and robustness. A common approach is to use a direct solver to precondition the system, improving the condition number and making the iterative process converge faster. For example, a sparse matrix can be initially reduced using a direct solver on a smaller or more manageable part of the system, followed by an iterative method to refine the solution. This combination can lead to significant improvements in efficiency, especially for very large or complex systems where neither direct nor iterative methods alone would be optimal.

3.2.3 Block Cimmino Method and Partitioning

The Block Cimmino method is a hybrid solver for large sparse linear systems. It combines direct and iterative approaches, making it efficient and particularly effective in parallel computing environments, by distributing computations across multiple processors.

3.2.3.1 Block Cimmino Method

The block cimmino method is a variant of the cimmino algorithm, which is a row projection method. In the block cimmino method, the matrix A is partitioned into blocks of rows. Each block of rows is used to form a subproblem that can be solved independently, making it highly suitable for parallel processing.

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} \tag{3.3}$$

Given a linear system $Ax = b$, where A is an $m \times n$ matrix, the matrix is divided into p blocks of rows, $A_1, A_2, \ldots, A_p$, with corresponding sub-vectors $b_1, b_2, \ldots, b_p$. Each block $A_i x = b_i$ represents a subproblem. The block cimmino method iteratively updates the solution vector x by projecting onto the solution spaces of these subproblems.

Mathematically, the update step in the block cimmino method can be described as:

$$x^{(k+1)} = x^{(k)} + \alpha \sum_{i=1}^{p} A_i^T (A_i A_i^T)^{-1} (b_i - A_i x^{(k)}) \tag{3.4}$$

where $x^{(k)}$ is the solution at the k-th iteration, and α is a relaxation parameter that controls the convergence rate.

The method leverages the parallelism by allowing each subproblem $A_i x = b_i$ to be solved independently, and the results are combined to update the solution vector x.

3.2.3.2 Partitioning

Partitioning is a crucial step in the block cimmino method as it directly affects the efficiency and performance of the algorithm. The goal of partitioning is to divide the matrix A into blocks in a way that balances the computational load and minimizes inter-processor communication.

Several strategies can be used for partitioning:

- **Row Partitioning:** The matrix A is divided into contiguous blocks of rows. This is the simplest form of partitioning but may not always result in balanced computational loads.
- **Block Partitioning:** The matrix is divided into blocks that may not be contiguous. This allows for more flexibility in balancing the load among processors.
- **Graph-based Partitioning:** The matrix is represented as a graph, with rows and columns as nodes and non-zero entries as edges. Graph partitioning algorithms are then used to divide the graph into subgraphs with balanced computational loads and minimal edge cuts, reducing communication overhead.

Effective partitioning aims to ensure that each processor has an approximately equal amount of work and that the communication between processors is minimized. This is critical for achieving high performance and scalability in parallel computing environments

3.3 Augmented Block Cimmino Distributed (ABCD)

Augmented Block Cimmino Distributed (ABCD) [22] solver is an advanced tool which is designed to solve large sparse linear systems efficiently. It provides two main implementations: The Augmented Block Cimmino method and a classical Block Cimmino method with an additional Conjugate Gradient (CG) method. The solver works in such a way that the sparse matrix is divided into several row blocks and can be processed separately and in parallel on different processors or nodes. This division allows for the distributed processing and the MPI is used to manage the processors. Moreover, the best linear algebra libraries are utilized for matrix computations to obtain the maximum performance. The solver also takes into account both shared and distributed memory systems in parallelism which helps in load balancing and reduction of cost of communication. These features contribute to the solver's high scalability and efficiency, making it a flexible and powerful solution for handling

complex and large-scale computational problems in various scientific and engineering fields.

3.4 Machine Learning

Machine learning is a part of artificial intelligence, often called AI. AI is the ability of a computer or robot to do tasks that usually need human intelligence, like recognizing speech, understanding language, making decisions, and solving problems. Machine learning helps computers learn from data and improve their performance over time without being explicitly programmed. It works by using algorithms that can find patterns in data and make predictions or decisions based on that data. For example, if you give a computer a lot of pictures of cats and dogs, it can learn to tell the difference between them. This learning process makes it possible for machines to handle tasks that are complex and varied, making our lives easier and more efficient.

3.4.1 Types of Machine Learning and Ensemble Learning Methods

ML has different types based on how the learning process happens. The main types of ML are supervised learning, unsupervised learning, and reinforcement learning.

3.4.1.1 Supervised Learning

Supervised Learning is a machine learning method where a computer is trained with labeled data, meaning it knows the correct output for given inputs. For example, to recognize fruits, the computer is given images of fruits along with their names. Supervised learning includes two main tasks: classification and regression. Classification predicts a category for input data, like labeling emails as "spam" or "not spam." Regression predicts a continuous value, like estimating house prices based on features such as location and size. Overall, supervised learning enables computers to make accurate predictions and decisions using labeled data.

Ensemble Learning is a technique that combines multiple machine learning models to make better predictions. Instead of relying on a single model, ensemble learning uses several models and combines their results to improve accuracy and robustness. There are different types of ensemble learning methods:

1. **Bagging Bootstrap Aggregating** involves training multiple models on different random samples of the training data and then combining their predictions. An example is the Random Forest algorithm, which builds multiple decision trees and averages their predictions.

2. **Boosting:** trains models sequentially, each one focusing on the errors made by the previous model. It combines the models' predictions to form a strong predictor. An example is the Gradient Boosting algorithm, which builds models that correct the errors of the previous ones.

3. **Stacking:** involves training multiple models and then using another model, called a metalearner, to combine their predictions. The base models make predictions, and the meta-learner uses these predictions as input to make the final prediction.

These ensemble methods help improve the performance of machine learning models by reducing errors and increasing accuracy, making them more reliable for various tasks.

3.4.1.2 Unsupervised Learning

In unsupervised learning, the computer is given data without the correct output. It has to find patterns and relationships in the data on its own. For example, it can group similar items together, like clustering customers with similar buying habits.

3.4.1.3 Reinforcement Learning

In reinforcement learning, the computer learns by making decisions and receiving feedback in the form of rewards or penalties. It learns to make better decisions over time to maximize the rewards. For example, a robot can learn to navigate a maze by receiving a reward when it reaches the end.

3.5 Algorithms

We are predicting the Sparse Linear Systems Partition Number. Our input is a sparse matrix; the output is the partition number. The partition numbers we have are 4, 6, 8, 10, 12, and 112. This makes the problem a classification problem, and we used these algorithms.

3.5.1 Logistic regression

Logistic regression is a type of statistical method used to predict the outcome of a binary variable, which means it can only be one of two possible outcomes. For example, it can be used to predict if an email is spam or not spam, or if a student will pass or fail a test. Unlike linear regression, which predicts a continuous number, logistic regression predicts probabilities. It uses a special function called the logistic function, or sigmoid function, which takes any number and turns it into a number between 0 and 1. This helps in estimating the probability that a certain event will happen. If the probability is greater than 0.5, the model predicts the event will happen; if it is less than 0.5, it predicts the event will not happen. Logistic regression is popular because it is simple, easy to use, and often gives good results for many kinds of problems.

3.5.2 Decision Tree

Decision Tree is a simple and intuitive machine learning algorithm used for both classification and regression tasks. It works by splitting the data into subsets based on feature values, creating a tree-like model of decisions. Starting with the entire dataset, the algorithm selects the best feature to split the data into subsets, based on criteria like Gini impurity or entropy for classification, and variance reduction for regression. Each branch of the tree represents a decision rule, and each leaf node represents an outcome or prediction. The process continues recursively, creating a tree structure where decisions are made by following a path from the root to a leaf. Decision Trees are easy to understand and interpret, making them popular for applications where model transparency is important. However, they can be prone to overfitting, especially with very deep trees. Techniques like pruning, setting a maximum depth, or using ensemble methods like Random Forests can help mitigate this issue.

3.5.3 Random Forest

Random Forest is a powerful machine learning algorithm used for both classification and regression tasks. It creates a "forest" of many decision trees, with each tree trained on different random subsets of the data and features. For classification, each tree votes for a class, and the class with the most votes is the final prediction. For regression, the

predictions from all the trees are averaged. This approach reduces overfitting, making the model robust and accurate. Random Forest handles large datasets, high-dimensional data, and missing values well, and it can estimate feature importance. It's widely used due to its flexibility, ease of understanding, and reliable performance.

3.5.4 K-Nearest Neighbors (KNeighbors)

KNeighbors algorithm is a versatile supervised learning method used for classification and regression tasks. It identifies predictions based on similarity: by finding the K closest instances in the training data to a new data point, using metrics like Euclidean distance. For classification, it predicts the majority class among these neighbors; for regression, it averages their target values. KNeighbors is straightforward to implement and understand, making it suitable for initial data exploration. However, it can be sensitive to outliers and the choice of distance metric. Techniques such as feature scaling can help improve its robustness. Overall, KNeighbors is valued for its simplicity, effectiveness with smaller datasets, and ability to capture nonlinear relationships.

3.5.5 Support Vector Classifier (SVC)

SVC is a reliable machine learning algorithm mainly used for classification tasks. It finds an optimal hyperplane that maximizes the margin between different classes in the data space. This hyperplane is determined by support vectors those data points closest to each class—which help define where the decision boundary lies. SVC can handle both linear and non-linear separations using different kernel functions like linear, polynomial, or radial basis function (RBF), chosen based on the data's complexity. Proper tuning of parameters like regularization (C parameter) and kernel-specific settings is crucial for improving model accuracy. Despite being sensitive to outliers, SVC is effective for managing high-dimensional data and complex decision boundaries, making it widely applicable in various fields needing precise classification.

3.5.6 Adaptive Boosting (AdaBoost)

AdaBoost is a machine learning ensemble method primarily used for classification tasks. It combines multiple weak learners, typically decision trees with limited depth, to create a strong learner. During training, AdaBoost assigns higher weights to misclassified

data points, allowing subsequent weak learners to focus more on the difficult cases and sequentially correct errors. The final prediction in AdaBoost is a weighted sum of predictions from all weak learners, with weights based on their accuracy. This approach ensures that more accurate models have a greater influence. AdaBoost is versatile and less prone to overfitting compared to individual weak learners, although it can be sensitive to noisy data and outliers, which may affect performance. Overall, AdaBoost is valued for achieving high accuracy with relatively simple base classifiers, making it popular in robust classification applications.

3.5.7 Gradient Boosting

Gradient Boosting is a powerful machine learning technique used for regression and classification tasks. It builds an ensemble of weak learners, usually decision trees, sequentially correcting errors made by each previous learner. This iterative process minimizes a chosen loss function, like mean squared error for regression or log-loss for classification, using gradient descent optimization. Starting with an initial learner fitted to the data, Gradient Boosting improves model accuracy with subsequent learners that refine predictions by focusing on residual errors left by previous iterations. It requires careful tuning of hyperparameters such as learning rate and tree depth to prevent overfitting, and regularization techniques help maintain model generalizability. Despite its sensitivity to parameter settings, Gradient Boosting is favored for its ability to capture complex data relationships effectively. It is widely used in applications such as web analytics, finance, and healthcare for its capacity to deliver accurate predictions across various domains, making it a versatile and robust machine learning approach.

3.5.8 Light Gradient Boosting Machine (LightGBM)

LightGBM is an advanced machine learning framework designed for efficient handling of large-scale datasets in classification and regression tasks. Unlike traditional gradient boosting methods, LightGBM employs a novel tree-based learning algorithm that grows trees vertically (leaf-wise) rather than horizontally (level-wise). This approach enhances training speed and reduces memory usage, making it particularly effective for datasets with millions of instances and high-dimensional features. LightGBM is well-suited for tasks where interactions between features are

critical, such as in finance and online advertising, due to its ability to capture complex relationships. It supports hyperparameter tuning, enabling users to optimize model performance effectively. However, careful parameter adjustment is essential to prevent overfitting, especially when dealing with smaller or noisy datasets. Despite these considerations, LightGBM is highly regarded for its speed, scalability, and accuracy in uncovering intricate patterns within data across various domains. Its efficient handling of large datasets and ability to capture nuanced feature interactions make it a preferred choice in industries requiring robust and scalable machine learning solutions.

3.5.9 Extreme Gradient Boosting (XGBoost)

XGBoost is an advanced machine learning algorithm known for its efficiency and performance in both classification and regression tasks. It belongs to the family of gradient boosting algorithms and is designed to optimize speed and model performance. XGBoost builds upon traditional gradient boosting techniques by incorporating regularization and parallel computing, making it highly scalable and suitable for large datasets.

The algorithm works by sequentially adding decision trees to an ensemble, each tree correcting errors made by its predecessors. XGBoost uses a technique called gradient boosting, which minimizes a chosen loss function by iteratively improving model predictions. It supports various objectives and evaluation metrics, allowing flexibility in model optimization.

XGBoost is favored for its ability to handle complex interactions and nonlinear relationships in data, making it suitable for a wide range of applications, including finance, healthcare, and online advertising. Despite its power, proper parameter tuning is crucial to prevent overfitting and achieve optimal performance. Overall, XGBoost is recognized for its speed, accuracy, and effectiveness in delivering high-quality predictions across diverse domains.

3.6 Data Augmentation and Oversampling

When working with imbalanced datasets, two methods that are used to improve the quantity of data available for training machine learning models are data augmentation and oversampling. Data augmentation is the process of adding additional data points by changing preexisting data such as in image processing when images are rotated or flipped. while oversampling aims to balance the dataset by creating new samples or repeating existing samples to increase the number of samples in the minority class.

3.6.1 Synthetic Minority Over-sampling Technique (SMOTE)

creates synthetic samples by selecting two or more similar instances in the minority class and generating new instances that are combinations of these instances. This helps to balance the class distribution without simply duplicating the minority class samples.

3.6.2 Adaptive Synthetic Sampling (ADASYN)

ADASYN is similar to SMOTE but focuses on generating more synthetic samples for the minority class instances that are harder to classify. It adapts to the data distribution by considering the density of minority class samples, thus improving the classifier's performance on difficult cases.

3.6.3 Random Over Sampler

It balances the dataset by randomly duplicating instances from the minority class until the desired class distribution is achieved. It is simple but can lead to overfitting as it doesn't create new, informative samples.

3.6.4 Borderline SMOTE

It is an extension of SMOTE that generates synthetic samples only for minority class instances that are near the decision boundary. This helps to strengthen the minority class in critical areas, improving classifier performance on these challenging points.

CHAPTER 4

METHODOLOGY

This chapter explores methods for predicting partition numbers in sparse linear systems and emphasizes the key features required to identify the optimal partition number for a given sparse matrix. This is essential for effectively applying machine learning techniques to solve sparse linear systems.

4.1 Dataset

The dataset for this thesis was obtained from the Suite Sparse Matrix Collection [23]. We chose matrices with at least 10,000 rows and columns, leaving smaller ones to use parallelism better. Because of memory limitations, we selected matrices with no more than 20 million nonzero elements. We focused on unsymmetric matrices because the block cimmino method works well with them. Using these criteria, we found 282 real-valued sparse matrices in our study.

We used the ABCD Solver to solve these linear systems and noted the key performance metrics. After reviewing the results, we prioritized the matrices with a maximum of 10,000 iterations and the shortest solution time. This included measuring the BCG Time, Factorization Time, and Mumps Initialization Time in seconds. This process enabled us to obtain 120 matrices.

We created three datasets:

- Dataset 1: 120 matrices with partition numbers 4 and 12
- Dataset 2: 120 matrices with partition numbers 4, 6, 8, 10, and 12
- Dataset 3: 10 largest matrices from Dataset 1 with partition numbers 4 and 112

We divided our datasets into 70% for training and 30% for testing. Figures 4.1, 4.3, and 4.2 illustrate the distribution of classes in our datasets, highlighting the existing imbalance. These figures give a detailed view of how classes are spread across the

dataset, showing instances of over-representation or under-representation within certain classes.

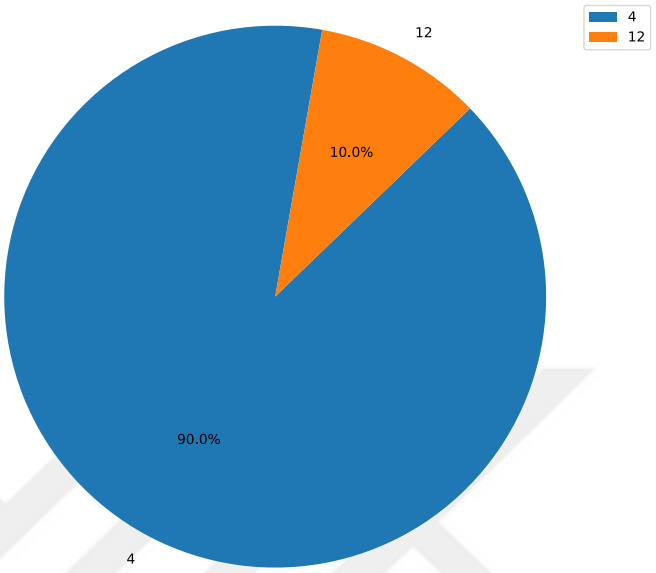

Figure 4.1 Dataset 1 distribution

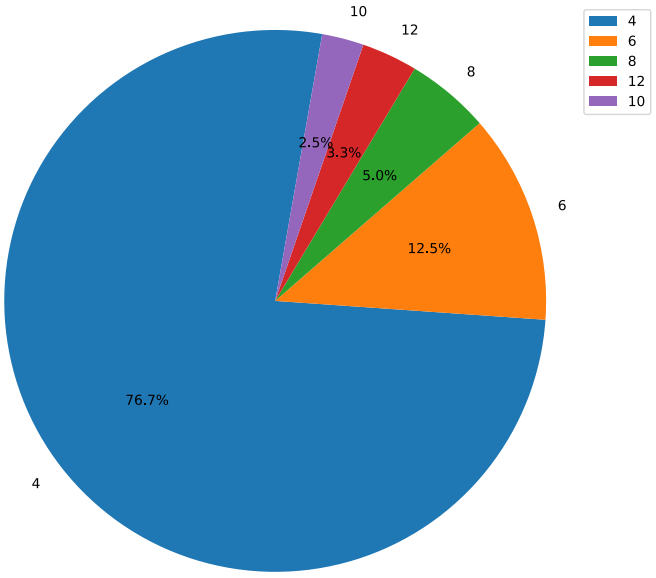

Figure 4.2 Dataset 2 distribution

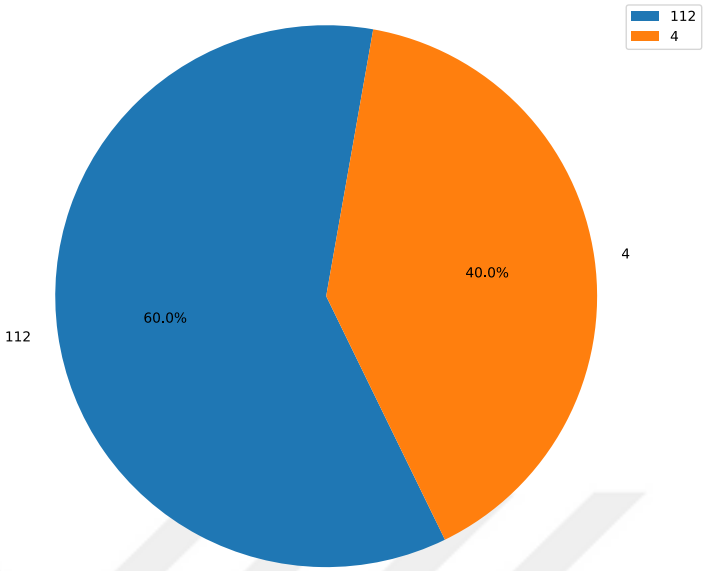

Figure 4.3 Dataset 3 distribution

4.2 Features

Table 4.1 Features

Feature	Description
M	The number of rows
NNZ	The number of nonzero elements.
Density	The density of nonzero elements. NNZ/(M*N)
Row Max	The maximum number of nonzero elements per row
Row Min	The minimum number of nonzero elements per row.
Row Mean	The average number of nonzero elements per row.
Row Median	The median number of non-zero elements per row.
Row STD	The Standard deviation of nonzero elements per row.
Group	Group of the matrix
Psym	Pattern Symmetry.
Kind	Kind of the matrix
Column Max	The maximum number of nonzero elements per Column
Column Min	The minimum number of nonzero elements per Column
Column Mean	The average number of nonzero elements per Column.
Column Median	The median number of non-zero elements per Column
Column STD	The Standard deviation of nonzero elements per Column.
Nsym	Numerical Symmetry.

We discovered that the features provided by the Suite Sparse Matrix Collection, including rows, columns, number of nonzero elements, kind, group, Pattern Symmetry, and Numerical Symmetry, were insufficient for our purposes. Therefore, we included additional features such as minimum, maximum, mean, median, and standard deviation for each matrix's rows and columns. Although this required extra computation, it contributed to better partitioning and reduced overall execution times. As for categorical features like Kind and Group, we grouped them into three general categories and utilized label encoding to manage them.

4.3 Experiments

In this thesis, we conducted several experiments using different partition numbers. Most experiments utilized two partition numbers: the minimum and the maximum. We selected the minimum to be 4 and the maximum based on the computer's maximum number of cores. Additionally, we conducted another experiment that involved using multiple partition numbers, specifically numbers between the minimum and maximum partition numbers.

4.3.1 Experiment 1: Analysis of a Dual-Partition Shared Memory System

In this experiment, we used Dataset 1, which included all 120 matrices. These matrices were analyzed with partition number of 4 and 12. Our shared-memory system consists of a 2-socket configuration with 64 GB of memory. Each socket housed a 6-core Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz processor, creating distinct non-uniform memory access (NUMA) domains. We employed the block Cimmino method to partition the sparse linear system. Figure 4.4 presents the performance profile of 120 matrix instances, summarizing the results of the experiment. A performance profile is a graphical tool used to evaluate the effectiveness of various parameters or algorithms. The vertical axis represents parallel running times as a performance metric, whereas the horizontal axis shows the performance ratio relative to the best performance achieved across the two partition counts for each problem.

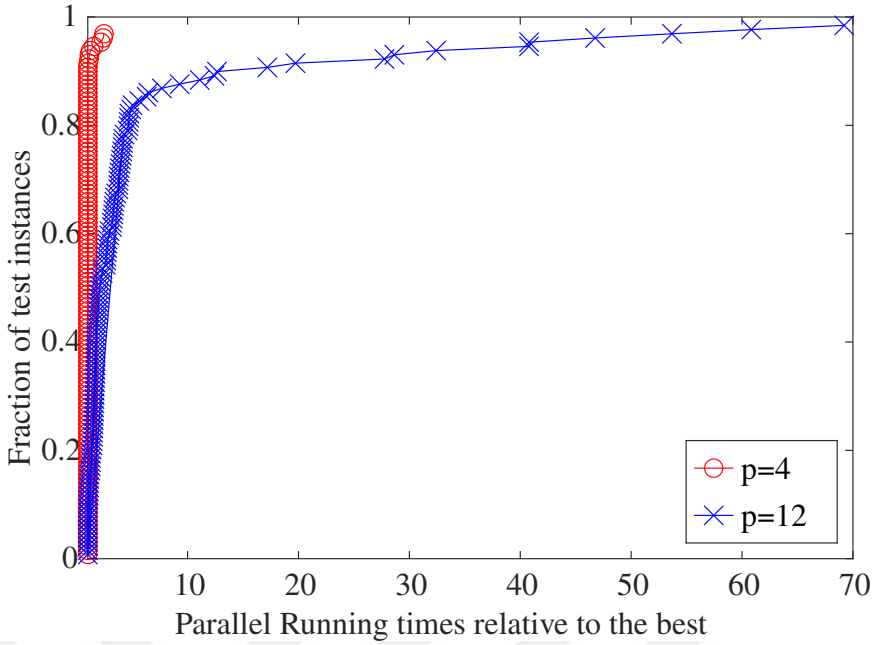

Figure 4.4 Comparison of two different partition numbers as performance profiles.

4.3.2 Experiment 2: Analysis of Shared Memory Performance with Multiple Partitions

In the previous experiment, the target was limited to two partition numbers, such as 4 and 12. However, in this experiment, we use Dataset 2, which involves multiple partition numbers: 4, 6, 8, 10, and 12. This experiment was conducted using a shared-memory system, similar to Experiment 1.

4.3.3 Experiment 3: Analysis of Dual-Partition Distributed Memory for Large Matrices

We focused on analyzing the 10 largest matrices from our dataset using partition numbers 4 and 112. Our computing tasks were performed on the National Center for High-Performance Computing (UHeM) [24] system, which consists of four distributed nodes, each equipped with 28 cores, totaling 112 cores.

Because of limited core hours, we selected these specific matrices for analysis. The details of the selected matrices, along with their optimal partition numbers highlighted in bold, are listed in Table 4.2. Our analysis found that setting the partition to 4 resulted in faster solutions using the parallel block cimmino method. However, for two matrices

(Hamrle3 and rajat30), the method failed because of insufficient memory.

Table 4.2 Matrix properties and solution time

Matrix Name	Rows	NNZ	Solution Time (p=4)	Solution Time (p=112)
circuit5M_dc	3523317	14865409	12.60	**3.87**
Hamrle3	1447360	5514242	Failed	**808.20**
rajat30	643994	6175244	Failed	**193.30**
cage13	445315	7479343	45.43	**8.98**
Largebasis	440020	5240084	**5.69**	596.67
nxp1	414604	2655880	**135.90**	3,682.72
language	399130	1216334	**52.70**	97.21
ASIC_320k	321821	1931828	67.72	**4.36**
ASIC_320ks	321671	1316085	2.42	**2.44**
ss1	205282	845089	4.06	**2.93**

4.4 Feature Importance

We used Random Forest to identify the important features because it works well with different data types and can find complex patterns. Since our datasets are non-linear, we chose Random Forest. Random Forest allowed us to determine the relative importance of each feature by examining how often and how significantly each feature was used in the trees' decision-making processes. This method assisted us in prioritizing the features that have the most significant impact on our model's predictions. After obtaining each feature's relative value, we sorted them and plotted each feature and its values using a horizontal bar chart. We performed this process for all datasets, and the results are shown in Figures 4.5 , 4.7 and 4.6

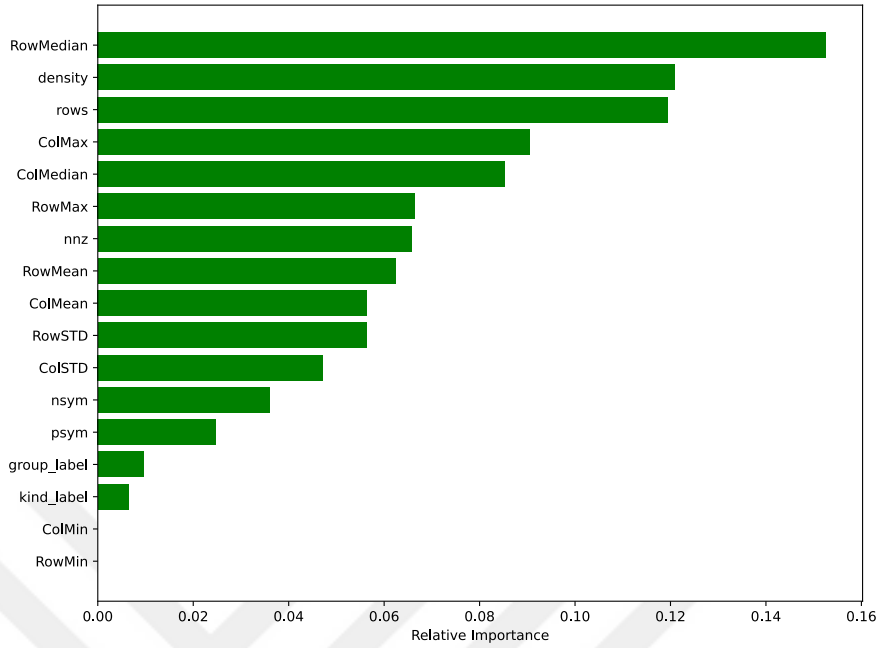

Figure 4.5 Dataset 1 feature importance

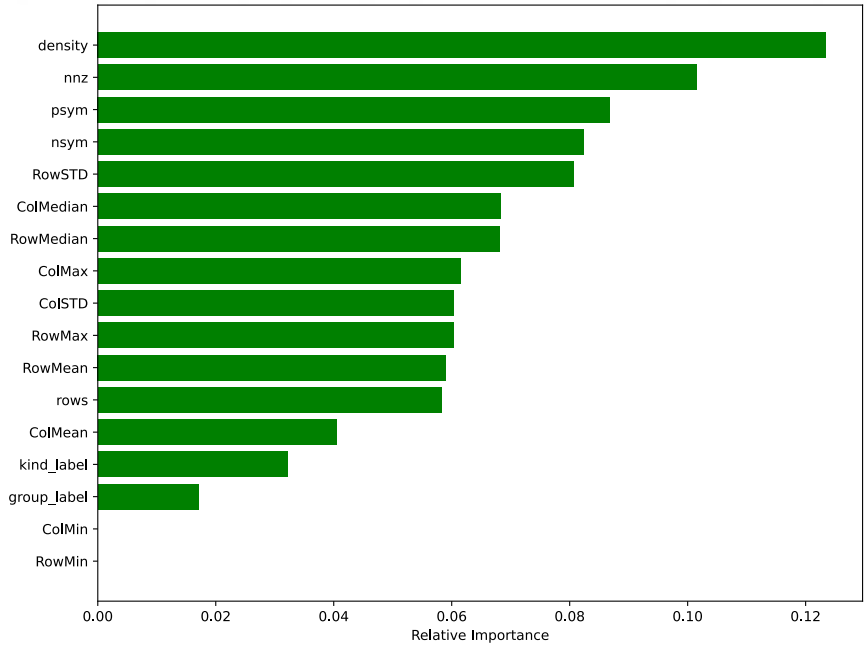

Figure 4.6 Dataset 2 feature importance

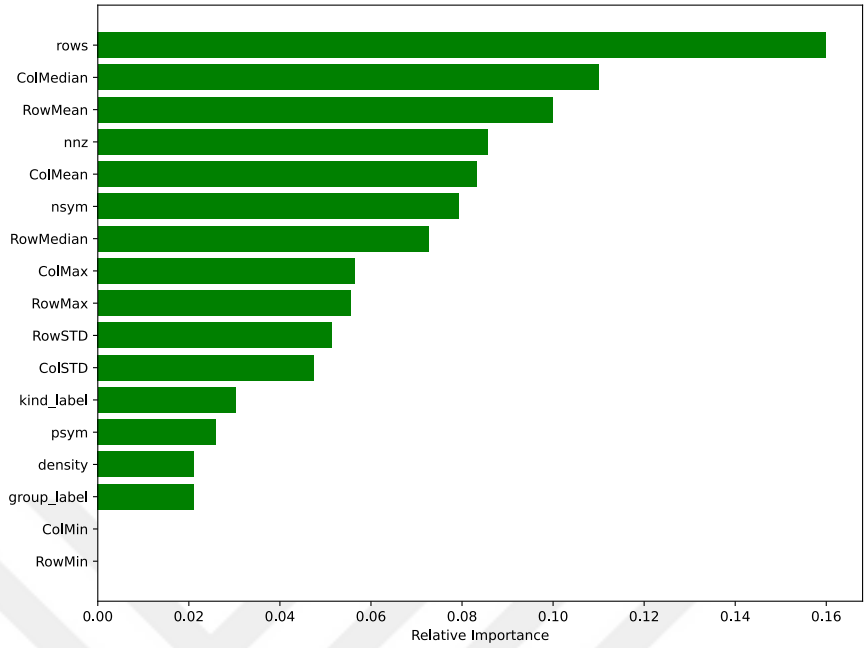

Figure 4.7 Dataset 3 feature importance

4.5 Feature Selection

To choose the most important features from our datasets, we used Feature Importance scores from the random forest classifier. We started by incrementally combining features, beginning with the top 1, 2, and 3 features and continuing until all features were included. This process is crucial for preventing overfitting and finding the right balance between bias and variance when using a variety of models such as Logistic Regression, Random Forest Classifier, Decision Tree Classifier, K-Neighbors Classifier, Support Vector Machine, AdaBoost Classifier, Gradient Boosting Classifier, LightGBM, and XGBoost Classifier. We then selected the model with the highest validation score and tested it on the test dataset using the AUC metric, which is suitable for unbalanced datasets.

4.5.1 Partition 4 and 12 Experiment

4.5.1.1 Baseline Experiment

Table 4.3, shows the best model for the top features based on feature importance, validation, and test scores. We observed that the Support Vector Classifier (SVC)

emerged as the best model and started to converge after including 10 features. The dataset used here is Dataset 1.

Table 4.3 Feature selection and model performance using Dataset 1

Model	**#Features**	**Validation (%)**	**Test (%)**
Gradient Boosting	1	82.3	78.5
AdaBoostClassifier	2	90.00	73.80
AdaBoostClassifier	3	90.00	73.80
AdaBoostClassifier	4	90.00	73.80
AdaBoostClassifier	5	90.00	73.80
AdaBoostClassifier	6	90.0	73.80
AdaBoostClassifier	7	90.0	73.80
AdaBoostClassifier	8	88.3	73.80
LGBMClassifier	9	88.30	78.10
SVC	10	90.00	89.00
SVC	11	90.00	89.00
SVC	12	90.00	89.00
SVC	13	90.00	89.00
SVC	14	90.00	89.00
SVC	15	90.00	89.00
SVC	16	90.00	89.00
SVC	17	90.00	89.00

4.5.1.2 Oversampling Techniques Comparison

We have a data imbalance problem, As shown in Figure 4.1. To fix this, we used different oversampling techniques, which are listed below. Each technique and its result are presented. We only applied oversampling techniques to our training data. Also, we used oversampling techniques after we did feature importance.

1. SMOTE

Table 4.4 Model performance with SMOTE and feature selection

Model	#Features	Validation (%)	Test (%)
LGBMClassifier	1	90.86	76.56
LGBMClassifier	2	94.98	76.95
LGBMClassifier	3	95.73	71.48
LGBMClassifier	4	95.73	71.48
LGBMClassifier	5	95.69	71.48
LGBMClassifier	6	95.69	71.48
LGBMClassifier	7	95.56	72.66
LGBMClassifier	8	95.98	74.22
LGBMClassifier	9	95.72	74.22
LGBMClassifier	10	95.59	74.61
LGBMClassifier	11	95.51	71.88
GradientBoostingClassifier	12	96.52	69.92
GradientBoostingClassifier	13	97.04	60.16
GradientBoostingClassifier	14	97.31	57.81
GradientBoostingClassifier	15	97.64	57.03
GradientBoostingClassifier	16	97.56	57.81
GradientBoostingClassifier	17	97.64	58.59

2. ADASYN

Table 4.5 Model performance with ADASYN and feature selection

Model	# Features	Validation (%)	Test (%)
LogisticRegression	1	91.18	72.66
LGBMClassifier	2	94.21	73.44
LGBMClassifier	3	96.23	75.39
LGBMClassifier	4	96.23	75.39
AdaBoostClassifier	5	95.72	73.83
AdaBoostClassifier	6	95.76	73.83
GradientBoostingClassifier	7	94.94	74.61
LGBMClassifier	8	94.98	80.47
LGBMClassifier	9	94.57	80.47
LGBMClassifier	10	94.76	83.98
LGBMClassifier	11	95.19	80.47
GradientBoostingClassifier	12	96.13	68.36
LGBMClassifier	13	97.04	71.88
GradientBoostingClassifier	14	97.61	60.16
GradientBoostingClassifier	15	97.53	58.59
AdaBoostClassifier	16	97.41	70.31
GradientBoostingClassifier	17	97.43	60.16

3. Random Oversampler

Table 4.6 Model performance with random oversampler and feature selection

Model	# Features	Validation (%)	Test (%)
XGBClassifier	1	98.17	75.00
RandomForestClassifier	2	99.21	78.52
LGBMClassifier	3	99.29	73.05
LGBMClassifier	4	99.29	73.05
LGBMClassifier	5	99.29	75.00
LGBMClassifier	6	99.29	75.00
LGBMClassifier	7	99.12	79.30
RandomForestClassifier	8	99.21	76.95
LGBMClassifier	9	99.12	78.52
LGBMClassifier	10	99.12	81.64
RandomForestClassifier	11	99.21	77.34
RandomForestClassifier	12	99.35	82.42
RandomForestClassifier	13	99.65	75.00
XGBClassifier	14	99.03	80.47
RandomForestClassifier	15	98.98	80.86
RandomForestClassifier	16	99.83	72.66
RandomForestClassifier	17	99.65	78.12

4. Borderline SMOTE

Table 4.7 Model performance with borderline SMOTE and feature selection

Model	# Features	Validation (%)	Test (%)
LogisticRegression	1	94.43	72.66
RandomForestClassifier	2	96.54	74.22
XGBClassifier	3	96.22	80.86
XGBClassifier	4	96.22	80.86
AdaBoostClassifier	5	96.26	73.83
AdaBoostClassifier	6	96.26	73.83
AdaBoostClassifier	7	95.83	70.31
GradientBoostingClassifier	8	96.09	71.48
LGBMClassifier	9	95.92	76.56
LGBMClassifier	10	95.75	77.34
LGBMClassifier	11	95.75	74.61
GradientBoostingClassifier	12	96.45	74.61
GradientBoostingClassifier	13	97.34	73.83
AdaBoostClassifier	14	97.47	71.09
AdaBoostClassifier	15	96.63	77.34
AdaBoostClassifier	16	96.63	77.34
GradientBoostingClassifier	17	96.77	71.88

4.5.2 Multi Partition Experiment

In this experiment we utilized Dataset 2, which contains multiple partition numbers: 4, 6, 8, 10, and 12. We employed 2-fold cross-validation due to the limited instances in some classes (partition numbers). Table 4.8 presents the results without using oversampling techniques.

In Table 4.9, we show the best model, the number of features selected for each oversampling algorithm, and their corresponding validation and test results. The best choice was the LGBMClassifier, using the top 8 features determined by feature importance. This model was selected based on the number of features, with both validation and test scores being close compared to other models.

Table 4.8 Feature selection and model performance using Dataset 2

Model	**#Features**	**Validation (%)**	**Test (%)**
LogisticRegression	1	82.26	84.87
LGBMClassifier	2	81.64	78.29
LGBMClassifier	3	78.44	76.43
LGBMClassifier	4	77.2	68.7
LGBMClassifier	5	77.88	70.89
LGBMClassifier	6	77.19	69.41
LGBMClassifier	7	77.37	74.61
LGBMClassifier	8	78.69	76.89
LGBMClassifier	9	78.78	74.98
LGBMClassifier	10	78.78	74.98
LGBMClassifier	11	78.78	76.02
LGBMClassifier	12	78.78	76.02
LGBMClassifier	13	78.43	75.85
LGBMClassifier	14	79.04	71.81
LGBMClassifier	15	78.43	72.02
LGBMClassifier	16	78.43	72.02
LGBMClassifier	17	78.43	72.02

Table 4.9 Oversampling techniques comparison

Method	**Model**	**#Features**	**Validation (%)**	**Test (%)**
SMOTE	RandomForestClassifier	15	98.71	65.24
ADASYN	LGBMClassifier	9	95.94	68.21
RandomOverSampler	RandomForestClassifier	6	99.04	73.37
BorderlineSMOTE	LGBMClassifier	15	99.00	63.26

4.5.3 Partition 4 and 112 Experiment

In this experiment, we used Dataset 3 from the distributed memory system. Due to the small size of the dataset, only the validation score was considered, and 4-fold cross-validation was applied. We observed that the model's performance improved consistently after selecting the top 2 features based on feature importance, with the random forest model showing the best performance. However, the performance decreased when including features from the top 13 to 16 and then increased again when all features were selected, as presented in Table 4.10. Based on these findings, we concluded that the combination of the top 2 features is the most effective for this dataset.

Table 4.10 Feature selection and model performance using Dataset 3

Model	#Features	Validation (%)
Logistic Regression	1	75
Logistic Regression	2	100
Logistic Regression	3	100
Logistic Regression	4	100
Logistic Regression	5	100
Gradient Boosting	6	100
Gradient Boosting	7	100
Logistic Regression	8	100
Gradient Boosting	9	100
Logistic Regression	10	100
Logistic Regression	11	100
Logistic Regression	12	100
Logistic Regression	13	88
Gradient Boosting	14	81
Logistic Regression	15	88
Logistic Regression	16	88
Logistic Regression	17	90

4.6 Model Selection and Result

We conducted three experiments with three different machine-learning models.

- **Experiment 1**: The model predicts two partition numbers: 4 and 12.
- **Experiment 2**: The model predicts multiple partition numbers: 4, 6, 8, 10, and 12.

- **Experiment 3**: The model predicts two partition numbers: 4 and 122.

We performed feature importance analysis for all our datasets and then conducted feature selection for each experiment. Subsequently, we selected the best models from Experiments 1 and 2 but not from Experiment 3 because Dataset 3 contains only 10 matrices, whereas Datasets 1 and 2 each contain 120 matrices. In Experiment 1, we selected the top 10 features based on feature importance from Dataset 1 and used the SVC model for these features. In Experiment 2, we selected the top 8 features and found that the LGBMClassifier was the best model for these features. We then compared the test performance of the selected models based on the accuracy, AUC, and F1 scores of these matrices. These metrics are illustrated in Table 4.11.

Table 4.11 Models evaluation

Model	Accuracy (%)	AUC (%)	F1-Score (%)
Dual Partition	91.6	89.0	89.3
Multi Partition	75.0	76.8	70.0

4.7 Performance Analysis

We normalized the actual solution times of all matrices based on the solution times for partition 4. Specifically, we divided the solution times of each matrix partition by the solution time of partition 4. The partition with the smallest value was selected as the best time ratio. Additionally, we identified the predicted partition value as the prediction ratio. We then averaged the values of each partition, the best time ratios, and the prediction ratios, presenting these results in Tables.4.12 and 4.13

Table 4.12 Performance analysis for Dataset 1

Average Best Time	Average Prediction ratio	Average P4	Average P12
0.94	0.98	1.00	6.57

Table 4.13 Performance analysis for Dataset 2

Average Best Time	Average Prediction ratio	Average P4	Average P6	Average P8	Average P10	Average P12
0.93	0.97	1.00	4.67	5.10	6.66	6.89

We have two models. One predicts two partition numbers, 4 and 12, while the other predicts multiple partition numbers, 4, 6, 8, 10, and 12. When we used these models, the

dual partition model achieved a 2% faster parallel solution time, while the multipartition model achieved a 3% improvement. Overall, these methods show faster parallel solution times with machine learning predictions. Their performances are very close to the best-case scenario, demonstrating the effectiveness of these approaches.

CHAPTER 5

DISCUSSION AND FUTURE WORK

In this thesis, we present a machine learning approach that effectively predicts the partition number for solving sparse linear systems. The current approaches like the trial and error method or choosing the maximum number of cores as the number of partitions are time-consuming and unreliable. We observed that some large sparse matrices perform best with small partition numbers, while smaller sparse systems might require a larger partition number. This variability demonstrates that each sparse matrix has unique features determining its optimal partition number. We have identified and highlighted the important features that influence the partition number.

Our goal is to save time by predicting the optimal partition number for sparse matrices instead of relying on older methods that may be ineffective and time-consuming.

For future work, the current limitation lies in the small dataset, which restricts our model's ability to predict other partition numbers. To address this, increasing the dataset will be a top priority along with adding new partition numbers.

CHAPTER 6

CONCLUSION

This thesis introduces a machine learning-based method for solving sparse linear systems by predicting the optimal partition number. This method significantly outperforms traditional trial-and-error methods in terms of computational efficiency and solution times. The approach demonstrates substantial improvements in efficiency and resource usage by selecting appropriate features for model training and employing techniques such as random forest classifiers for feature importance. It presents two models: one predicting two partition numbers with an AUC score of 89% and another predicting multiple partition numbers with an AUC score of 76%. Both models, trained on diverse matrices, leveraged existing features in a novel context, addressing the common partitioning problem in sparse linear systems, particularly with the block cimmino method. This approach eliminates the need for traditional methods such as trial and error or choosing the maximum number of cores. It highlights the potential of integrating machine learning into scientific computing to enhance performance and efficiency in solving complex problems.

REFERENCES

[1] Borgatti S. P. and Halgin D. S., "On network theory," *Organization Science*, vol. 22, pp. 1168–1181, 9 2011.

[2] Wang J. L., Chiou J. M., and Müller H. G., "Functional data analysis," pp. 257–295, 6 2016.

[3] Wilson E. L., Bathe K., Peterson F., and Dovey H., "Sap—a structural analysis program for linear systems," *Nuclear Engineering and Design*, vol. 25, no. 2, pp. 257–274, 1973.

[4] Liang Y.-Z., Fang K.-T., and Xu Q.-S., "Uniform design and its applications in chemistry and chemical engineering," pp. 43–57, 2001.

[5] Bramley R. and Sameh A., "Row projection methods for large nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 1, pp. 168–193, 1992.

[6] Drummond L. A., Duff I. S., Guivarch R. *et al.*, "Partitioning strategies for the block cimmino algorithm," *Journal of Engineering Mathematics*, vol. 93, pp. 21–39, 2015.

[7] Torun F. S., Manguoglu M., and Aykanat C., "A novel partitioning method for accelerating the block cimmino algorithm," *SIAM Journal on Scientific Computing*, vol. 40, pp. C827–C850, 2018.

[8] Torun F. S., Manguoglu M., and Aykanat C., "Enhancing block cimmino for sparse linear systems with dense columns via schur complement," *SIAM Journal on Scientific Computing*, vol. 45, no. 2, pp. C49–C72, 2023. [Online]. Available: https://doi.org/10.1137/21M1453475

[9] Duff I., Leleux P., Ruiz D., and Torun F. S., "Row replicated block cimmino," *SIAM Journal on Scientific Computing*, vol. 45, no. 4, pp. C207–C232, 2023. [Online]. Available: https://doi.org/10.1137/22M1487710

[10] Funk Y., Götz M., and Anzt H., "Prediction of optimal solvers for sparse linear systems using deep learning," in *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2022, pp. 14–24.

[11] Sood K., "Automated selection of numerical solvers," *University of Oregon, Computer and Information Sciences Department, Directed research proposal*, 2015.

[12] Nisa I., Siegel C., Rajam A. S., Vishnu A., and Sadayappan P., "Effective machine learning based format selection and performance modeling for spmv on gpus." Institute of Electrical and Electronics Engineers Inc., 8 2018, pp. 1056–1065.

[13] Dufrechou E., Ezzatti P., and Quintana-Orti E. S., "Automatic selection of sparse triangular linear system solvers on gpus through machine learning techniques," vol. 2019-October. IEEE Computer Society, 10 2019, pp. 41–47.

[14] Tang Z., Zhang H., and Chen J., "Graph neural networks for selection of preconditioners and krylov solvers," in *NeurIPS 2022 Workshop: New Frontiers in Graph Learning*, 2022.

[15] Sood K., Norris B., and Jessup E., "Comparative performance modeling of parallel preconditioned krylov methods," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2017, pp. 26–33.

[16] Wei B., Wang Y., Chang F., Gao J., and Ji W., "Predicting optimal sparse general matrix-matrix multiplication algorithm on gpus," *International Journal of High Performance Computing Applications*, vol. 38, pp. 245–259, 5 2024.

[17] Grementieri L. and Galeone P., "Towards neural sparse linear solvers," *arXiv preprint arXiv:2203.06944*, 2022.

[18] Kuefler E. and Chen T.-Y., "Lncs 5101 - on using reinforcement learning to solve sparse linear systems," 2008.

[19] Dufrechou E., Ezzatti P., Freire M., and Quintana-Ortí E. S., "Machine learning for optimal selection of sparse triangular system solvers on gpus," *Journal of Parallel and Distributed Computing*, vol. 158, pp. 47–55, 12 2021.

[20] Chen W., Zhang B., Jin S., Ai B., and Zhong Z., "Solving sparse linear inverse problems in communication systems: A deep learning approach with adaptive depth," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 1, pp. 4–17, 2020.

[21] Elgohary A., Boehm M., Haas P. J., Reiss F. R., and Reinwald B., "Compressed linear algebra for large-scale machine learning," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 960–971, 2016.

[22] Zenadi M., "The augmented block cimmino distributed solver, ABCD solver," Ph.D. dissertation, University of Toulouse, 2013.

[23] Davis T. A. and Hu Y., "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011.

[24] UHEM, "National Center for High Performance Computing," 2024. [Online]. Available: http://www.uhem.itu.edu.tr

APPENDICES

Appendix A: Dataset 1 Matrix properties and solution time

Appendix B: Dataset 1 Performance Analysis

Appendix C: Dataset 2 Matrix and solution time

Appendix D: Dataset 2 Performance Analysis

Appendix A – Dataset 1 Matrix properties and solution time

Table A.1 Dataset 1: Matrix properties and solution time

Matrix Name	Rows	NNZ	P4	P12
ASIC_320ks	321671	1316085	2.29	1.05
ASIC_320k	321821	1931828	Failed	6.33
circuit5M_dc	3523317	14865409	24.73	10.34
ss1	205282	845089	Failed	3.05
cage13	445315	7479343	Failed	457.39
cage12	130228	2032536	49.14	19.98
language	399130	1216334	182.03	163.39
majorbasis	160000	1750416	Failed	3.66
memplus	17758	99147	4.40	3.51
Zhao1	33861	166453	0.55	0.53
jan99jac040sc	13694	72734	10.22	9.00
rajat30	643994	6175244	102.50	66.72
blockqp1	60012	640033	1.61	2.07
crashbasis	160000	1750416	5.34	5.42
lung2	109460	492564	1.49	1.54
torso2	115967	1033473	1.77	2.05
trans4	116835	749800	14.10	17.26
poli4	33833	73249	1.30	1.41
ASIC_100ks	99190	578890	3.22	4.69
trans5	116835	749800	17.65	19.93
rajat27	20640	97353	0.70	1.13
bayer01	57735	275094	1.22	4.74
dc2	116835	766396	20.93	36.81
cz10228	10228	102876	0.23	0.44
coupled	11341	97193	0.95	1.41
cz20468	20468	206076	0.35	1.51
powersim	15838	64424	0.59	1.38

Table A.1 Dataset 1 Matrix properties and solution time (continued)

Matrix Name	Rows	NNZ	P4	P12
cz40948	40948	412148	0.91	4.47
largebasis	440020	5240084	11.76	59.21
wang4	26068	177196	1.81	3.34
ifiss_mat	96307	3599932	11.85	18.91
rajat22	39899	195429	2.30	8.85
av41092	41092	1683902	10.71	18.45
windtunnel_evap3d	40816	803978	12.51	18.87
ted_A_unscaled	10605	424587	0.72	1.95
chem_master1	40401	201201	5.23	16.63
Goodwin_030	10142	312814	1.15	1.87
TSOPF_RS_b39_c7	14098	252446	1.97	4.12
TSOPF_RS_b162_c4	20374	812749	3.68	8.07
ted_A	10605	424587	0.91	2.53
nmos3	18588	237130	1.33	6.29
bayer10	13436	71594	1.10	4.61
venkat25	62424	1717763	12.40	36.80
ACTIVSg10K	20000	135888	1.90	7.91
e40r0100	17281	553562	2.11	6.19
TSOPF_RS_b162_c3	15374	610299	2.51	6.67
hvdc1	24842	158426	3.84	12.48
Goodwin_040	17922	561677	2.22	7.53
rma10	46835	2329092	8.65	32.62
Hamrle3	1447360	5514242	725.57	1248.24
venkat50	62424	1717777	12.56	47.63
shermanACb	18510	145149	14.01	30.59
epb3	84617	463625	21.08	74.89
garon2	13535	373235	2.43	6.34
LeGresley_87936	87936	593276	20.69	86.38
jan99jac060sc	20614	111903	17.45	19.50

Table A.1 Dataset 1 Matrix properties and solution time (continued)

Matrix Name	Rows	NNZ	P4	P12
Goodwin_054	32510	1030878	6.90	27.98
rim	22560	1014951	5.18	17.87
TSOPF_RS_b39_c19	38098	684206	8.67	40.44
inlet	11730	328323	1.98	6.30
jan99jac080sc	27534	151063	19.81	39.71
TSOPF_RS_b300_c1	14538	1474325	9.48	26.23
Zhao2	33861	166453	20.12	39.77
Goodwin_071	56021	1797934	17.45	79.98
TSOPF_RS_b39_c30	60098	1079986	23.45	92.59
jan99jac120sc	41374	229385	59.57	90.74
jan99jac100sc	34454	190224	52.32	74.54
rajat15	37261	443573	4.93	84.76
Goodwin_095	100037	3226066	58.63	240.59
nxp1	414604	2655880	452.89	1501.25
Goodwin_127	178437	5778545	176.71	664.60
mark3jac040	18289	106803	27.81	91.49
mark3jac060	27449	160723	47.60	223.12
mark3jac080	36609	214643	72.07	407.20
mult_dcop_03	25187	193216	1.34	Failed
lhr10c	10672	232633	7.10	65.90
lhr10	10672	228395	17.09	64.08
mimo28x28_system	13251	48737	1.72	119.05
nopss_11k	11685	44941	1.62	86.96
ww_vref_6405	13251	48737	1.99	121.06
Zd_Jac2_db	22835	676439	5.72	267.41
cryg10000	10000	49699	2.16	59.95
std1_Jac2_db	21982	498771	5.77	235.34
Zd_Jac3_db	22835	713907	8.25	267.43
std1_Jac3_db	21982	531826	8.46	242.34

Table A.1 Dataset 1 Matrix properties and solution time (continued)

Matrix Name	Rows	NNZ	P4	P12
sme3Da	12504	874887	5.43	107.13
xingo_afonso_itaipu	13250	48735	9.38	Failed
lhr14c	14270	307858	7.81	98.79
Zd_Jac6_db	22835	663643	21.60	268.38
lhr17c	17576	381975	12.72	141.12
k3plates	11107	378927	9.75	63.05
lhr34c	35152	764014	53.05	407.26
3D_51448_3D	51448	537038	119.88	756.35
lhr11c	10964	233741	18.22	72.79
g7jac200sc	59310	717620	313.65	860.33
g7jac060sc	17730	183325	109.26	165.13
scircuit	170998	958936	1300.94	2420.41
g7jac160sc	47430	564952	349.25	690.22
g7jac140sc	41490	488633	294.81	588.24
mark3jac060sc	27449	160723	188.53	318.12
mark3jac040sc	18289	106803	154.29	174.09
mark3jac080sc	36609	214643	280.79	474.18
g7jac080sc	23670	259648	211.46	269.12
mark3jac100sc	45769	268563	394.25	620.24
mark3jac140sc	64089	376395	668.62	900.38
ASIC_100k	99340	940621	8.70	11.19
bayer02	13935	63307	0.34	0.58
circuit_3	12127	48137	3.15	14.97
dc1	116835	766396	39.63	43.24
dc3	116835	766396	20.33	26.17
epb1	14734	95053	1.26	2.04
hcircuit	105676	513072	14.21	47.66
hvdc2	189860	1339638	5.23	213.39
Ill_Stokes	20896	191368	2.24	3.34

Table A.1 Dataset 1 Matrix properties and solution time (continued)

Matrix Name	**Rows**	**NNZ**	**P4**	**P12**
rajat20	86916	604299	11.05	23.16
rajat28	87190	606489	3.43	5.88
TSOPF_RS_b300_c2	28338	2943887	16.23	43.02
TSOPF_RS_b300_c3	42138	4413449	36.41	66.69
venkat01	62424	1717792	1.80	2.84
viscoplastic2	32769	381326	5.31	9.51

Appendix B - Dataset 1 Performance Analysis

Table B.1 Dataset 1 performance analysis

Matrix Name	P	Pred.P	P4	P12	Best Time	Pred. Ratio
ASIC_320ks	12	4	1.00	0.46	0.46	1.00
ASIC_320k	12	4	1.00	0.10	0.10	1.00
circuit5M_dc	12	12	1.00	0.42	0.42	0.42
ss1	12	4	1.00	0.10	0.10	1.00
cage13	12	12	1.00	0.10	0.10	0.10
cage12	12	4	1.00	0.41	0.41	1.00
language	12	4	1.00	0.90	0.90	1.00
majorbasis	12	4	1.00	0.10	0.10	1.00
memplus	12	4	1.00	0.80	0.80	1.00
Zhao1	12	4	1.00	0.96	0.96	1.00
jan99jac040sc	12	4	1.00	0.88	0.88	1.00
rajat30	12	4	1.00	0.65	0.65	1.00
blockqp1	4	4	1.00	1.29	1.00	1.00
crashbasis	4	4	1.00	1.01	1.00	1.00
lung2	4	4	1.00	1.03	1.00	1.00
torso2	4	4	1.00	1.16	1.00	1.00
trans4	4	4	1.00	1.22	1.00	1.00
poli4	4	4	1.00	1.08	1.00	1.00
ASIC_100ks	4	4	1.00	1.46	1.00	1.00
trans4	4	4	1.00	1.22	1.00	1.00
rajat27	4	4	1.00	1.61	1.00	1.00
bayer01	4	4	1.00	3.89	1.00	1.00
dc2	4	4	1.00	1.76	1.00	1.00
cz10228	4	4	1.00	1.91	1.00	1.00
coupled	4	4	1.00	1.48	1.00	1.00
cz20468	4	4	1.00	4.31	1.00	1.00
powersim	4	4	1.00	2.34	1.00	1.00

Table B.1 Dataset 1 performance analysis (continued)

Matrix Name	P	Pred.P	P4	P12	Best Time	Pred. Ratio
cz40948	4	4	1.00	4.91	1.00	1.00
largebasis	4	4	1.00	5.03	1.00	1.00
wang4	4	4	1.00	1.85	1.00	1.00
ifiss_mat	4	4	1.00	1.60	1.00	1.00
rajat22	4	4	1.00	3.85	1.00	1.00
av41092	4	4	1.00	1.72	1.00	1.00
windtunnel_evap3d	4	4	1.00	1.51	1.00	1.00
ted_A_unscaled	4	4	1.00	2.71	1.00	1.00
chem_master1	4	4	1.00	3.18	1.00	1.00
Goodwin_030	4	4	1.00	1.63	1.00	1.00
TSOPF_RS_b39_c7	4	4	1.00	2.09	1.00	1.00
TSOPF_RS_b162_c4	4	4	1.00	2.19	1.00	1.00
ted_A_unscaled	4	4	1.00	2.71	1.00	1.00
nmos3	4	4	1.00	4.73	1.00	1.00
bayer10	4	4	1.00	4.19	1.00	1.00
venkat25	4	4	1.00	2.97	1.00	1.00
ACTIVSg10K	4	4	1.00	4.16	1.00	1.00
e40r0100	4	4	1.00	2.93	1.00	1.00
TSOPF_RS_b162_c3	4	4	1.00	2.66	1.00	1.00
hvdc1	4	4	1.00	3.25	1.00	1.00
Goodwin_040	4	4	1.00	3.39	1.00	1.00
rma10	4	4	1.00	3.77	1.00	1.00
Hamrle3	4	4	1.00	1.72	1.00	1.00
venkat50	4	4	1.00	3.79	1.00	1.00
shermanACb	4	4	1.00	2.18	1.00	1.00
epb3	4	4	1.00	3.55	1.00	1.00
garon2	4	4	1.00	2.61	1.00	1.00
LeGresley_87936	4	4	1.00	4.17	1.00	1.00
jan99jac060sc	4	4	1.00	1.12	1.00	1.00

Table B.1 Dataset 1 performance analysis (continued)

Matrix Name	P	Pred.P	P4	P12	Best Time	Pred. Ratio
Goodwin_054	4	4	1.00	4.06	1.00	1.00
rim	4	4	1.00	3.45	1.00	1.00
TSOPF_RS_b39_c19	4	4	1.00	4.66	1.00	1.00
inlet	4	4	1.00	3.18	1.00	1.00
jan99jac080sc	4	4	1.00	2.00	1.00	1.00
TSOPF_RS_b300_c1	4	4	1.00	2.77	1.00	1.00
Zhao2	4	4	1.00	1.98	1.00	1.00
Goodwin_071	4	4	1.00	4.58	1.00	1.00
TSOPF_RS_b39_c30	4	4	1.00	3.95	1.00	1.00
jan99jac120sc	4	4	1.00	1.52	1.00	1.00
jan99jac100sc	4	4	1.00	1.42	1.00	1.00
rajat15	4	4	1.00	17.19	1.00	1.00
Goodwin_095	4	4	1.00	4.10	1.00	1.00
nxp1	4	4	1.00	3.31	1.00	1.00
Goodwin_127	4	4	1.00	3.76	1.00	1.00
mark3jac040	4	4	1.00	3.29	1.00	1.00
mark3jac060	4	4	1.00	4.69	1.00	1.00
mark3jac080	4	4	1.00	5.65	1.00	1.00
mult_dcop_03	4	4	1.00	10.00	1.00	1.00
lhr10c	4	4	1.00	9.28	1.00	1.00
lhr10	4	4	1.00	3.75	1.00	1.00
mimo28x28_system	4	4	1.00	69.22	1.00	1.00
nopss_11k	4	4	1.00	53.68	1.00	1.00
mimo28x28_system	4	4	1.00	69.22	1.00	1.00
Zd_Jac2_db	4	4	1.00	46.75	1.00	1.00
cryg10000	4	4	1.00	27.75	1.00	1.00
std1_Jac2_db	4	4	1.00	40.79	1.00	1.00
Zd_Jac3_db	4	4	1.00	32.42	1.00	1.00
std1_Jac3_db	4	4	1.00	28.65	1.00	1.00

Table B.1 Dataset 1 performance analysis (continued)

Matrix Name	P	Pred.P	P4	P12	Best Time	Pred. Ratio
sme3Da	4	4	1.00	19.73	1.00	1.00
xingo_afonso_itaipu	4	4	1.00	10.00	1.00	1.00
lhr14c	4	4	1.00	12.65	1.00	1.00
Zd_Jac6_db	4	4	1.00	12.42	1.00	1.00
lhr17c	4	4	1.00	11.09	1.00	1.00
k3plates	4	4	1.00	6.47	1.00	1.00
lhr34c	4	4	1.00	7.68	1.00	1.00
3D_51448_3D	4	4	1.00	6.31	1.00	1.00
lhr11c	4	4	1.00	4.00	1.00	1.00
g7jac200sc	4	4	1.00	2.74	1.00	1.00
g7jac060sc	4	4	1.00	1.51	1.00	1.00
scircuit	4	4	1.00	1.86	1.00	1.00
g7jac160sc	4	4	1.00	1.98	1.00	1.00
g7jac140sc	4	4	1.00	2.00	1.00	1.00
mark3jac060	4	4	1.00	4.69	1.00	1.00
mark3jac040	4	4	1.00	3.29	1.00	1.00
mark3jac080	4	4	1.00	5.65	1.00	1.00
g7jac080sc	4	4	1.00	1.27	1.00	1.00
mark3jac100sc	4	4	1.00	1.57	1.00	1.00
mark3jac140sc	4	4	1.00	1.35	1.00	1.00
ASIC_100k	4	4	1.00	1.29	1.00	1.00
bayer02	4	4	1.00	1.71	1.00	1.00
circuit_3	4	4	1.00	4.75	1.00	1.00
dc2	4	4	1.00	1.76	1.00	1.00
dc2	4	4	1.00	1.76	1.00	1.00
epb1	4	4	1.00	1.62	1.00	1.00
hcircuit	4	4	1.00	3.35	1.00	1.00
hvdc2	4	4	1.00	40.80	1.00	1.00
Ill_Stokes	4	4	1.00	1.49	1.00	1.00

Table B.1 Dataset 1 performance analysis (continued)

Matrix Name	P	Pred.P	P4	P12	Best Time	Pred. Ratio
rajat20	4	4	1.00	2.10	1.00	1.00
rajat28	4	4	1.00	1.71	1.00	1.00
TSOPF_RS_b300_c2	4	4	1.00	2.65	1.00	1.00
TSOPF_RS_b300_c3	4	4	1.00	1.83	1.00	1.00
venkat01	4	4	1.00	1.58	1.00	1.00
viscoplastic2	4	4	1.00	1.79	1.00	1.00

Appendix C - Dataset 2 Matrix and Solution time

Table C.1 Dataset 2 matrix and solution time

Matrix Name	P	P4	P6	P8	P10	P12
LeGresley_87936	4	19.99	33.68	50.14	71.72	85.97
TSOPF_RS_b300_c1	4	9.33	13.37	13.23	24.76	25.64
nxp1	4	442.74	Failed	1021.79	1251.56	1431.29
powersim	6	0.64	0.49	0.79	1.15	1.34
rajat20	4	8.31	13.69	20.76	28.50	23.76
cage13	10	Failed	Failed	525.10	451.41	452.49
venkat01	4	1.70	1.84	2.13	2.47	2.76
venkat50	4	12.27	18.23	28.13	36.86	47.72
dc1	6	39.44	36.33	38.79	36.68	46.24
mark3jac060sc	4	176.58	218.31	235.26	268.19	318.13
nopss_11k	4	1.47	118.08	108.11	90.16	88.54
std1_Jac3_db	4	7.56	216.74	192.64	36.51	238.35
TSOPF_RS_b162_c4	4	3.66	5.34	5.36	7.51	8.07
cz20468	4	0.32	0.53	0.63	0.95	1.53
g7jac160sc	4	344.31	503.58	532.42	600.37	684.24
g7jac200sc	4	321.63	657.93	699.63	789.45	857.38
hcircuit	8	17.86	17.35	16.80	44.49	42.90
jan99jac060sc	6	17.34	13.66	17.14	16.23	21.18
largebasis	4	11.33	19.82	30.36	51.97	58.67
g7jac080sc	4	110.44	116.30	199.24	239.18	271.11
windtunnel_evap3d	4	12.35	13.56	13.47	14.65	18.80
TSOPF_RS_b39_c30	4	23.02	37.48	59.30	77.66	91.80
Zd_Jac2_db	4	5.62	219.05	235.68	231.56	267.40
ACTIVSg10K	4	1.69	2.50	4.21	6.08	7.66
g7jac140sc	4	300.82	444.57	450.41	514.33	580.22
Ill_Stokes	6	2.19	2.06	2.19	2.68	3.23
coupled	4	0.91	1.70	1.02	1.45	1.33

Table C.1 Dataset 2 Matrix properties and solution time (continued)

Matrix Name	P	P4	P6	P8	P10	P12
blockqp1	4	1.53	2.09	1.65	1.86	2.04
epb3	4	20.65	29.56	40.03	55.32	74.61
TSOPF_RS_b300_c3	4	35.00	48.42	80.34	70.00	59.06
Goodwin_071	4	17.59	32.46	42.91	58.79	80.67
bayer02	4	0.34	0.37	0.53	0.51	0.67
Goodwin_095	4	58.82	96.27	140.63	189.93	238.59
TSOPF_RS_b39_c19	4	8.89	19.57	19.30	27.25	40.33
circuit_3	4	2.44	4.12	11.07	11.88	13.98
ifiss_mat	4	11.92	12.35	14.93	17.05	18.77
rim	4	5.54	5.74	7.73	11.83	17.95
dc3	6	20.92	20.82	28.56	21.23	23.65
lhr14c	4	7.36	94.58	90.05	39.93	98.01
torso2	4	1.81	2.01	2.26	2.46	2.48
rajat28	6	3.27	3.26	4.55	5.14	5.11
inlet	4	1.95	2.90	4.11	4.64	6.36
sme3Da	4	5.39	114.21	99.48	100.04	107.13
trans4	4	13.08	13.48	18.20	16.49	15.82
Goodwin_127	4	176.73	301.01	412.75	551.26	664.61
Zd_Jac3_db	4	7.30	136.00	224.58	226.56	262.41
jan99jac080sc	4	19.73	27.32	26.67	31.73	38.80
mimo28x28_system	4	1.66	120.07	114.06	113.08	120.10
ASIC_100k	4	6.91	7.44	8.02	8.01	8.24
3D_51448_3D	4	113.96	551.84	592.48	Failed	755.37
language	10	169.50	134.72	136.21	126.58	157.82
rajat30	6	101.84	47.88	Failed	60.72	66.24
cryg10000	4	1.98	74.36	73.56	63.36	62.58
cz40948	4	0.86	1.43	2.01	3.04	4.51
Zhao2	4	20.61	21.25	29.67	31.94	40.55
ted_A	4	0.76	0.96	1.43	2.20	2.52

Table C.1 Dataset 2 Matrix properties and solution time (continued)

Matrix Name	P	P4	P6	P8	P10	P12
mark3jac060	4	47.41	74.40	120.27	157.19	228.14
bayer10	4	1.22	2.31	1.79	92.80	4.28
bayer01	4	1.11	2.16	2.67	3.51	4.69
rajat22	4	2.21	4.53	6.96	5.99	8.85
xingo_afonso_itaipu	4	5.71	121.07	112.08	105.07	Failed
trans5	8	17.13	20.51	16.97	17.94	20.14
lhr10	4	18.66	64.74	19.63	59.22	65.69
lung2	8	1.63	2.01	1.51	1.59	1.55
e40r0100	4	2.13	2.79	3.56	4.65	6.24
mult_dcop_03	4	1.34	1.35	1.40	2.17	Failed
hvdc1	4	3.49	4.26	5.92	8.95	13.07
mark3jac040sc	6	157.36	143.18	145.14	171.15	172.10
viscoplastic2	4	5.32	6.77	7.10	8.27	9.04
Goodwin_030	4	1.05	1.23	1.46	1.75	1.91
TSOPF_RS_b300_c2	4	17.26	23.90	33.83	33.25	37.86
nmos3	4	1.05	1.73	2.68	4.73	6.38
poli4	8	1.28	1.36	1.23	1.35	1.40
chem_master1	4	5.41	7.43	10.27	13.03	16.40
k3plates	4	7.30	60.59	55.39	58.17	63.57
ww_vref_6405	4	1.81	125.08	113.07	113.09	119.06
jan99jac040sc	8	9.43	6.93	5.63	7.02	9.00
venkat25	4	12.91	18.26	23.89	30.60	36.77
cz10228	4	0.20	0.24	0.36	0.36	0.45
epb1	4	1.20	1.43	1.31	1.56	2.02
shermanACb	4	13.91	16.40	15.46	26.32	30.47
dc2	4	21.38	25.06	27.29	23.86	37.20
majorbasis	10	Failed	Failed	4.30	4.20	4.45
memplus	8	3.26	4.32	2.69	3.99	3.26
lhr34c	4	48.60	166.49	314.42	358.30	403.25

Table C.1 Dataset 2 Matrix properties and solution time (continued)

Matrix Name	P	P4	P6	P8	P10	P12
jan99jac120sc	4	62.00	62.79	73.03	87.23	91.84
garon2	4	2.35	3.01	3.67	4.78	6.15
hvdc2	4	2.41	30.07	83.94	126.46	264.34
mark3jac080	4	66.78	134.43	179.34	275.27	411.18
wang4	4	1.74	2.21	2.43	3.16	3.29
Zd_Jac6_db	4	9.87	239.08	200.71	33.90	262.51
lhr17c	4	11.92	45.87	118.24	120.18	141.14
mark3jac140sc	4	628.59	684.04	739.70	805.51	893.35
jan99jac100sc	6	50.31	48.06	63.28	62.98	74.32
Zhao1	6	0.49	0.45	0.51	0.49	0.57
TSOPF_RS_b162_c3	4	2.45	3.73	3.92	5.24	6.62
av41092	6	10.90	9.32	11.59	14.83	18.50
ASIC_320ks	12	2.29	1.74	1.43	1.38	1.32
lhr11c	4	16.88	74.86	69.64	36.72	73.09
scircuit	4	1241.04	2000.62	2260.59	2300.50	2500.40
mark3jac080sc	4	262.76	326.55	352.33	411.27	462.21
Hamrle3	4	738.38	1002.65	931.65	1078.84	1238.19
mark3jac040	4	26.74	41.40	55.48	71.85	93.50
crashbasis	6	5.19	4.66	4.89	5.03	5.28
TSOPF_RS_b39_c7	4	1.87	2.67	2.58	3.63	3.88
rajat27	6	0.57	0.52	0.83	0.99	1.08
std1_Jac2_db	4	5.79	107.82	177.50	208.52	233.36
lhr10c	4	6.93	69.72	15.92	65.10	64.51
ASIC_100ks	6	3.31	3.16	3.62	3.70	4.55
ted_A_unscaled	4	0.60	0.84	1.15	1.41	2.14
Goodwin_054	4	6.54	9.88	15.94	20.11	27.37
ASIC_320k	6	Failed	5.96	10.20	Failed	6.09
mark3jac100sc	4	393.00	448.64	482.41	541.37	614.30
cage12	12	48.63	37.65	31.74	23.60	19.97

Table C.1 Dataset 2 Matrix properties and solution time (continued)

Matrix Name	P	P4	P6	P8	P10	P12
rajat15	4	4.77	25.15	40.10	500.21	81.99
rma10	4	8.36	11.01	15.82	23.88	32.21
Goodwin_040	4	2.57	2.94	4.20	4.93	7.56
circuit5M_dc	12	24.66	Failed	13.52	11.48	10.29
ss1	12	Failed	Failed	4.94	Failed	3.01
g7jac060sc	4	100.28	153.23	141.17	149.12	166.10

Appendix D – Dataset 2 Performance Analysis

Table D.1 Performance analysis of Dataset 2

Matrix Name	P	Pred. P	P4	P6	P8	P10	P12	Best T	Pred. R
3D_51448_3D	4	4	1.00	4.84	5.20	10.00	6.63	1.00	1.00
ACTIVSg10K	4	4	1.00	1.48	2.49	3.60	4.54	1.00	1.00
ASIC_100k	4	4	1.00	1.08	1.16	1.16	1.19	1.00	1.00
ASIC_100ks	6	4	1.00	0.95	1.09	1.12	1.37	0.95	1.00
ASIC_320k	6	4	1.00	0.10	0.17	1.00	0.10	0.10	1.00
ASIC_320ks	12	12	1.00	0.76	0.63	0.61	0.58	0.58	0.58
Goodwin_030	4	4	1.00	1.17	1.38	1.66	1.81	1.00	1.00
Goodwin_040	4	4	1.00	1.14	1.63	1.92	2.94	1.00	1.00
Goodwin_054	4	4	1.00	1.51	2.43	3.07	4.18	1.00	1.00
Goodwin_071	4	4	1.00	1.85	2.44	3.34	4.59	1.00	1.00
Goodwin_095	4	4	1.00	1.64	2.39	3.23	4.06	1.00	1.00
Goodwin_127	4	4	1.00	1.70	2.34	3.12	3.76	1.00	1.00
Hamrle3	4	4	1.00	1.36	1.26	1.46	1.68	1.00	1.00
Ill_Stokes	6	6	1.00	0.94	1.00	1.23	1.48	0.94	0.94
LeGresley_87936	4	4	1.00	1.68	2.51	3.59	4.30	1.00	1.00
TSOPF_RS_b162_c3	4	4	1.00	1.52	1.60	2.14	2.70	1.00	1.00
TSOPF_RS_b162_c4	4	4	1.00	1.46	1.46	2.05	2.20	1.00	1.00
TSOPF_RS_b300_c1	4	4	1.00	1.43	1.42	2.65	2.75	1.00	1.00
TSOPF_RS_b300_c2	4	4	1.00	1.39	1.96	1.93	2.19	1.00	1.00
TSOPF_RS_b300_c3	4	4	1.00	1.38	2.30	2.00	1.69	1.00	1.00
TSOPF_RS_b39_c19	4	4	1.00	2.20	2.17	3.06	4.53	1.00	1.00
TSOPF_RS_b39_c30	4	4	1.00	1.63	2.58	3.37	3.99	1.00	1.00
TSOPF_RS_b39_c7	4	4	1.00	1.43	1.38	1.94	2.07	1.00	1.00
Zd_Jac2_db	4	4	1.00	38.99	41.96	41.22	47.60	1.00	1.00
Zd_Jac3_db	4	4	1.00	18.62	30.75	31.03	35.94	1.00	1.00
Zd_Jac6_db	4	4	1.00	24.22	20.34	3.43	26.60	1.00	1.00
Zhao1	6	4	1.00	0.91	1.04	1.00	1.15	0.91	1.00
Zhao2	4	4	1.00	1.03	1.44	1.55	1.97	1.00	1.00
av41092	6	6	1.00	0.85	1.06	1.36	1.70	0.85	0.85
bayer01	4	4	1.00	1.95	2.40	3.15	4.22	1.00	1.00
bayer02	4	4	1.00	1.09	1.55	1.49	1.96	1.00	1.00
bayer10	4	4	1.00	1.90	1.47	76.16	3.51	1.00	1.00
blockqp1	4	4	1.00	1.37	1.08	1.22	1.34	1.00	1.00
cage12	12	12	1.00	0.77	0.65	0.49	0.41	0.41	0.41
cage13	10	10	1.00	1.00	0.12	0.10	0.10	0.10	0.10
chem_master1	4	4	1.00	1.37	1.90	2.41	3.03	1.00	1.00
circuit5M_dc	12	12	1.00	4.17	0.55	0.47	0.42	0.42	0.42
circuit_3	4	4	1.00	1.69	4.54	4.87	5.73	1.00	1.00
coupled	4	4	1.00	1.87	1.12	1.60	1.46	1.00	1.00
crashbasis	6	6	1.00	0.90	0.94	0.97	1.02	0.90	0.90
cryg10000	4	4	1.00	37.61	37.20	32.05	31.65	1.00	1.00
cz10228	4	4	1.00	1.18	1.76	1.79	2.22	1.00	1.00
cz20468	4	4	1.00	1.66	1.98	2.98	4.79	1.00	1.00
cz40948	4	4	1.00	1.66	2.33	3.54	5.25	1.00	1.00
dc1	6	4	1.00	0.92	0.98	0.93	1.17	0.92	1.00
dc2	4	4	1.00	1.17	1.28	1.12	1.74	1.00	1.00
e40r0100	4	4	1.00	1.31	1.68	2.19	2.94	1.00	1.00
epb1	4	4	1.00	1.19	1.09	1.30	1.68	1.00	1.00
epb3	4	4	1.00	1.43	1.94	2.68	3.61	1.00	1.00
g7jac060sc	4	4	1.00	1.53	1.41	1.49	1.66	1.00	1.00

Table D.2 Performance analysis of Dataset 2 (continued)

Matrix Name	P	Pred. P	P4	P6	P8	P10	P12	Best T	Pred. R
g7jac080sc	4	4	1.00	1.05	1.80	2.17	2.45	1.00	1.00
g7jac140sc	4	4	1.00	1.48	1.50	1.71	1.93	1.00	1.00
g7jac160sc	4	4	1.00	1.46	1.55	1.74	1.99	1.00	1.00
g7jac200sc	4	4	1.00	2.05	2.18	2.45	2.67	1.00	1.00
garon2	4	4	1.00	1.28	1.56	2.03	2.61	1.00	1.00
hcircuit	8	8	1.00	0.97	0.94	2.49	2.40	0.94	0.94
hvdc1	4	4	1.00	1.22	1.70	2.56	3.75	1.00	1.00
hvdc2	4	4	1.00	12.48	34.82	52.46	109.66	1.00	1.00
ifiss_mat	4	4	1.00	1.04	1.25	1.43	1.58	1.00	1.00
inlet	4	4	1.00	1.49	2.11	2.39	3.27	1.00	1.00
jan99jac040sc	8	4	1.00	0.73	0.60	0.74	0.95	0.60	1.00
jan99jac060sc	6	6	1.00	0.79	0.99	0.94	1.22	0.79	0.79
jan99jac080sc	4	6	1.00	1.38	1.35	1.61	1.97	1.00	1.38
jan99jac100sc	6	6	1.00	0.96	1.26	1.25	1.48	0.96	0.96
jan99jac120sc	4	4	1.00	1.01	1.18	1.41	1.48	1.00	1.00
k3plates	4	4	1.00	8.29	7.58	7.96	8.70	1.00	1.00
language	10	4	1.00	0.79	0.80	0.75	0.93	0.75	1.00
largebasis	4	4	1.00	1.75	2.68	4.59	5.18	1.00	1.00
lhr10	4	4	1.00	3.47	1.05	3.17	3.52	1.00	1.00
lhr10c	4	4	1.00	10.06	2.30	9.39	9.31	1.00	1.00
lhr11c	4	4	1.00	4.44	4.13	2.18	4.33	1.00	1.00
lhr14c	4	4	1.00	12.85	12.23	5.42	13.31	1.00	1.00
lhr17c	4	4	1.00	3.85	9.92	10.08	11.84	1.00	1.00
lhr34c	4	4	1.00	3.43	6.47	7.37	8.30	1.00	1.00
lung2	8	8	1.00	1.23	0.93	0.98	0.95	0.93	0.93
majorbasis	10	10	1.00	1.00	0.10	0.10	0.11	0.10	0.10
mark3jac040	4	4	1.00	1.55	2.08	2.69	3.50	1.00	1.00
mark3jac040sc	6	6	1.00	0.91	0.92	1.09	1.09	0.91	0.91
mark3jac060	4	4	1.00	1.57	2.54	3.32	4.81	1.00	1.00
mark3jac060	4	4	1.00	1.57	2.54	3.32	4.81	1.00	1.00
mark3jac080	4	4	1.00	2.01	2.69	4.12	6.16	1.00	1.00
mark3jac080	4	4	1.00	2.01	2.69	4.12	6.16	1.00	1.00
mark3jac100sc	4	4	1.00	1.14	1.23	1.38	1.56	1.00	1.00
mark3jac140sc	4	4	1.00	1.09	1.18	1.28	1.42	1.00	1.00
memplus	8	8	1.00	1.32	0.82	1.22	1.00	0.82	0.82
mimo28x28_system	4	4	1.00	72.14	68.54	67.95	72.16	1.00	1.00
mult_dcop_03	4	4	1.00	1.01	1.04	1.62	10.00	1.00	1.00
nmos3	4	4	1.00	1.64	2.55	4.49	6.05	1.00	1.00
nopss_11k	4	4	1.00	80.54	73.75	61.50	60.40	1.00	1.00
nxp1	4	4	1.00	10.00	2.31	2.83	3.23	1.00	1.00
poli4	8	8	1.00	1.06	0.96	1.06	1.09	0.96	0.96
powersim	6	4	1.00	0.76	1.24	1.80	2.10	0.76	1.00
rajat15	4	4	1.00	5.27	8.40	104.82	17.18	1.00	1.00
rajat20	4	6	1.00	1.65	2.50	3.43	2.86	1.00	1.65
rajat22	4	4	1.00	2.05	3.16	2.72	4.01	1.00	1.00
rajat27	6	6	1.00	0.91	1.44	1.72	1.88	0.91	0.91
rajat28	6	6	1.00	1.00	1.39	1.57	1.56	1.00	1.00
rajat30	6	4	1.00	0.47	4.70	0.60	0.65	0.47	1.00
rim	4	4	1.00	1.04	1.40	2.14	3.24	1.00	1.00
rma10	4	4	1.00	1.32	1.89	2.86	3.85	1.00	1.00
scircuit	4	4	1.00	1.61	1.82	1.85	2.01	1.00	1.00
shermanACb	4	4	1.00	1.18	1.11	1.89	2.19	1.00	1.00

Table D.3 Performance analysis of Dataset 2

Matrix Name	P	Pred. P	P4	P6	P8	P10	P12	Best T	Pred. R
sme3Da	4	4	1.00	21.17	18.44	18.55	19.86	1.00	1.00
ss1	12	4	1.00	1.00	0.16	1.00	0.10	0.10	1.00
std1_Jac2_db	4	4	1.00	18.62	30.66	36.01	40.30	1.00	1.00
std1_Jac3_db	4	4	1.00	28.69	25.50	4.83	31.55	1.00	1.00
ted_A	4	4	1.00	1.28	1.90	2.92	3.34	1.00	1.00
ted_A	4	4	1.00	1.28	1.90	2.92	3.34	1.00	1.00
torso2	4	4	1.00	1.11	1.25	1.36	1.38	1.00	1.00
trans4	4	8	1.00	1.03	1.39	1.26	1.21	1.00	1.39
trans5	8	8	1.00	1.20	0.99	1.05	1.18	0.99	0.99
venkat01	4	4	1.00	1.08	1.25	1.45	1.62	1.00	1.00
venkat25	4	4	1.00	1.41	1.85	2.37	2.85	1.00	1.00
venkat50	4	4	1.00	1.49	2.29	3.00	3.89	1.00	1.00
viscoplastic2	4	4	1.00	1.27	1.33	1.55	1.70	1.00	1.00
wang4	4	4	1.00	1.27	1.40	1.81	1.89	1.00	1.00
windtunnel_evap3d	4	4	1.00	1.10	1.09	1.19	1.52	1.00	1.00
mimo28x28_system	4	4	1.00	72.14	68.54	67.95	72.16	1.00	1.00
xingo_afonso_itaipu	4	4	1.00	21.20	19.62	18.39	10.00	1.00	1.00