ITERATIVE SOLVER SELECTION TECHNIQUES FOR SPARSE LINEAR SYSTEMS

by

KANIKA SOOD

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2019

DISSERTATION APPROVAL PAGE

Student: Kanika Sood

Title: Iterative Solver Selection Techniques for Sparse Linear Systems

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Boyana Norris	Chair
Dejing Dou	Core Member
Lei Jiao	Core Member
Elizabeth Jessup	External Member
Elizabeth Bohls	Institutional Representative

and

Janet Woodruff-Borden	Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2019

© 2019 Kanika Sood

DISSERTATION ABSTRACT

Kanika Sood

Doctor of Philosophy

Department of Computer and Information Science

June 2019

Title: Iterative Solver Selection Techniques for Sparse Linear Systems

Scientific and engineering applications are dominated by linear algebra and depend on scalable solutions of sparse linear systems. For large problems, preconditioned iterative methods are a popular choice. High-performance numerical libraries offer a variety of preconditioned Newton-Krylov methods for solving sparse problems. However, the selection of a well-performing Krylov method remains to be the user's responsibility. This research presents the technique for choosing well-performing parallel sparse linear solver methods, based on the problem characteristics and the amount of communication involved in the Krylov methods.

This dissertation includes previously published (unpublished) co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Kanika Sood

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene
Mody Institute of Technology & Science, Rajasthan

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2019, University of Oregon
Master of Science, Computer and Information Science, 2014, University of Oregon
Bachelor of Technology, Computer Science, 2011, Mody Institute of Technology & Science

AREAS OF SPECIAL INTEREST:

Machine Learning
Databases
Artificial Intelligence

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, University of Oregon, 2014 – 2019

Subcontract Intern, Argonne National Laboratory- University of Oregon, Summer 2018

Givens Associate, Argonne National Laboratory, Summer 2017

REMS Intern, Schlumberger, 2016

Instructor, University of Oregon, Summer 2014

Graduate Teaching Fellow, University of Oregon, 2012 – 2014

Summer Intern, Indian Institute of Technology, Delhi, Summer 2013

Systems Engineer, IBM 2011 – 2012

GRANTS, AWARDS AND HONORS:

Outstanding Lightning Talk of the day Award, Supercomputing (SC), 2018

Best paper finalist: K. Sood, B. Norris, and E. Jessup. Comparative performance modeling of parallel preconditioned Krylov methods, Proceedings the 18th IEEE International Conference on High Performance Computing and Communications (HPCC), 2017

Erwin and Gertrude Juilfs Scholarship, University of Oregon, 2016

Intern Video Competition Winner, Schlumberger, 2016

UO Work-Study Award, University of Oregon, 2013

PUBLICATIONS:

Sood, K. & Norris, B. & Neill, B. & Srinivasan, S. & Jessup, E. (2019). Speeding up Parallel Scientific Computations through Low-Cost Machine Learning Models. [In preparation]

Grannan, A. & **Sood, K.** & Norris, B. & Dubey, A. (2019). Understanding the Landscape of Scientific Software Used on High-Performance Computing Platforms *The International Journal of High Performance Computing Applications.* [Submitted]

Sood, K. & Norris, B. & Jessup, E. (2017). Comparative Performance Modeling of Parallel Preconditioned Krylov Methods *IEEE International Conference on High Performance Computing and Communications.*

Jessup, E. & Motter, P. & Norris, B. & **Sood, K.** (2016) Performance-Based Numerical Solver Selection in the Lighthouse Framework. *SIAM Journal on Scientific Computing.*

Motter, P. & **Sood, K.** & Jessup, E. & Norris, B. (2015). Lighthouse : An Automated Solver Selection Tool. *Software Engineering for High Performance Computing in Computational Science and Engineering* .

Sood, K. & Norris, B. & Jessup, E. (2015). Lighthouse: A taxonomy-based solver selection tool *Proceedings of the Second Workshop on Software Engineering for Parallel Systems* .

ACKNOWLEDGEMENTS

Professional

Most of all, I would like to thank my advisor, Dr. Boyana Norris, for her infinite guidance, unbounded support and immense encouragement throughout my Ph.D. journey. Boyana taught me how to conduct research, write readable technical papers, target a milestone rather than a deadline, and keep going. I am most grateful to you for giving me the opportunity to present the research work in numerous conferences and workshops, and introducing me to qualified researchers in the field and open up collaboration opportunities and make connections. I wish to thank you for making this research extremely interesting, for sharing ideas to explore new research opportunities for our existing work and for resolving the challenges that came along the way. I want to thank you for your contributions in all aspects of the research, for the numerous discussions we had, for your supervision and guidance without which this work would not have been possible. I would like to thank you for closely reading all the papers, posters and thesis and providing constructive feedback while teaching me technical writing. Above all, you taught me how to prepare for goals: with an open mind, and a broad smile. Thank you for being more than an advisor and for being the person that you are, I could not have asked for a better advisor and mentor.

I would like to thank Dr. Elizabeth Jessup, who served as my co-advisor, for her guidance and advice, without which this work would not have been possible. I am most grateful to Liz for collaborating on all our papers and her valuable feedback for this research. Liz taught me how to write readable research papers better in a way that makes that conveys the idea in the simplest way. I sincerely

thank you for the numerous edits for all our writings, your constructive feedback and suggestions for thesis writing and answering my math queries throughout my Ph.D. years. Your timely feedback and suggestions have greatly improved this research.

I would like to thank my committee members, Dr. Dejing Dou, Dr. Lei Jiao, and Dr. Elizabeth Bohls for their suggestions and support for this dissertation. I would like to thank my other collaborators Pate Motter, Ben O'Neill and Samuel Pollard for being the best of collaborators. Thank you Sam for proofreading my reports, for the technical discussions and the constructive feedback. Thank you Ellen Klowden for proofreading my technical reports. I am grateful to all my HPCL colleagues for attending my rehearsal talks and suggestions for improving the same and for making this journey more exciting. Also, I would like to thank all my friends in the Computer Science department and at University of Oregon for making my graduate school journey more interesting.

Personal

I dedicate my thesis to my father and my husband, who have been my support system throughout the journey. My father has been my inspiration and motivation all my life. Thank you for giving me the freedom to follow my dreams, even when it meant sending me away to another country. Thank you for teaching me the importance of hard work and to never give up even when it gets tough. Thank you for making our education your first priority and for always choosing nothing less than the best for us. I cannot thank you enough for your endless support, countless blessings, unconditional love and all your hard work. You gave true meaning to my life. I am so lucky to have the best father a daughter can have. This achievement would not have been possible without you.

I would also like to thank my husband for his immense support, understanding and unconditional love during graduate school. The way you supported me and stood up for me at each and every phase of this journey is commendable. Thank you for accompanying me throughout my Ph.D., and giving a push whenever I needed it. Thank you for your selfless support and motivating me whenever I fell short. I could not have asked for a more supportive husband. This dream would not have come true without you.

I thank my mom, sisters, and Saurangshu Pandey for encouraging me to attend graduate school and being extremely supportive throughout. I would like to thank my Mohit Aggarwal for his encouragement and supportive words.

I wish to thank my uncles and aunts who were a support system for my parents when I moved to the States for graduate school. Thank you Suri Uncle, and Tripta Aunty who supported me in all possible ways, during my undergraduate studies in India. I also thank my uncle, Pavan and aunt, Sushma, who have always been there for our family throughout and motivated me in all phases of my life. This achievement would not have been possible without all of you. Last but not the least, I would like to thank all my friends, relatives and others who motivated me and supported me in my educational journey.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1. Motivation 3

1.2. Research Goals and Approaches 3

1.3. Co-Authored Material 5

1.4. Dissertation Outline 5

1.4.1. Convergence model 6

1.4.2. Communication Model 7

1.4.3. Matrix-free Feature Computation 8

1.4.4. Domain-Specific Use Case 9

1.5. Summary 10

II. LINEAR SYSTEM SOLUTION SCHEMES 12

2.1. Motivation 12

2.2. Direct Solvers 14

2.3. Iterative Solvers 15

2.4. Preconditioning 16

2.5. Single-Method Solver Systems 18

2.6. Multi-method Solver Systems 20

2.6.1. Composite Solvers 20

2.6.2. Adaptive Solvers 22

2.6.3. Poly-iterative Solvers 24

Chapter Page

2.6.4. Self Adapting Solvers Large-Scale Solver Architecture (SALSA) Solvers 27

2.6.5. Linear System Analyzer Solvers 27

2.6.6. Finite Element Tearing and Interconnecting and its parallel solution algorithm(FETI) 31

2.7. Accuracy of Solutions 32

2.8. Summary 33

III. CONVERGENCE MODEL FOR SPARSE LINEAR SOLVER CLASSIFICATION 35

3.1. Motivation 36

3.2. Dataset 37

3.3. Supervised Learning 37

3.4. Machine learning classification techniques 38

3.4.1. BayeNet 39

3.4.2. SVM 39

3.4.3. k-nearest neighbor 39

3.4.4. ADT 39

3.4.5. Random Forest 40

3.4.6. J48 41

3.5. Solver Selection as a Classification Problem 41

3.6. Feature Computation 42

3.7. Solving the linear systems 45

3.8. Solver classification 46

3.8.1. PETSc Solvers 47

3.8.2. PETSc Preconditioners 49

Chapter Page

3.9. Cost Reduction 50

3.10. Feature Set 52

3.11. Performance Evaluation 57

3.12. Experimental Results 59

3.13. Summary 61

IV. COMMUNICATION MODEL FOR SPARSE LINEAR SOLVER SELECTION 63

4.1. Motivation 64

4.2. Communication Cost of Preconditioned Krylov Methods 64

4.3. Building the Analytical Communication Model 66

4.3.1. Identify operations with communication 67

4.3.2. Matrix-vector product 67

4.3.3. Reduction operations 69

4.3.4. Application of Preconditioners 70

4.3.5. Assign cost to operations with communication 70

4.4. Compare Communication Cost across Krylov Method Implementations 71

4.4.1. GMRES and Conjugate Gradient: 72

4.4.2. Flexible GMRES (FGMRES) and Conjugate Gradient: . 72

4.4.3. TQFMR and Conjugate Gradient: 73

4.4.4. BiCG and Conjugate Gradient: 73

4.4.5. BCGS and Conjugate Gradient, BCGS and BiCG: . . . 73

4.4.6. BCGS and TFQMR: 74

4.4.7. BCGS and GMRES: 74

4.4.8. BCGS and iBCGS: 75

Chapter Page

4.4.9. iBCGS and GMRES: 76

4.4.10. BiCG and GMRES: 76

4.4.11. GMRES and TFQMR 76

4.5. Compare Communication Cost for Preconditioner Implementations 77

4.6. Generate Communication-based Ranking 78

4.7. Summary 79

V. COMBINING THE CONVERGENCE AND COMMUNICATION MODELS 80

5.1. Motivation 80

5.2. Modeling Computation and Communication for Krylov Methods 81

5.3. Empirical Evaluation 84

5.3.1. Results for a $1,000 \times 1,000$ mesh with a Grashof 100 84

5.3.2. Results for a $1,000 \times 1,000$ mesh with a Grashof $1,000$ 85

5.3.3. Findings 91

5.4. Summary 97

VI. MATRIX-FREE FEATURE COMPUTATION 98

6.1. Motivation 98

6.2. Multiphysics Simulations 99

6.3. Feature Computation 99

6.3.1. Reduced Feature Set 100

Chapter Page

6.3.2. Sample Based Feature Extraction - Efficient and Practical 101

6.4. Machine Learning Classification Performance 107

6.4.1. Classification evaluation on SuiteSparse dataset 109

6.4.2. Classification evaluation on MOOSE dataset 110

6.4.3. Observations based on classification evaluations 110

6.4.4. Classification evaluation with C5.0 algorithm 112

6.4.5. Solver ranking and validation 113

6.5. Testing in Run Time Applications 115

6.5.1. Driven cavity problem 100×100 grid 117

6.5.2. Driven cavity problem 128×128 grid 117

6.6. Summary 118

VII. SOLVER SELECTION IN FINITE-ELEMENT MULTIPHYSICS SIMULATIONS 119

7.1. Motivation 119

7.2. MOOSE framework 120

7.3. Classification Model 125

7.3.1. Feature Selection 125

7.3.2. Feature Reduction 125

7.3.3. Solvers and Preconditioners 126

7.3.4. Solver Classification 127

7.4. Experiments 127

7.4.1. Data Collection 129

7.5. Results 133

7.5.1. Construction timing of each classifier 133

Chapter Page

7.5.2. Prediction accuracy of each classifier 135

7.6. Summary 137

VIII. CONCLUSION AND FUTURE WORK 139

8.1. Conclusion 139

8.2. Future Work 143

.3. Appendix 144

REFERENCES CITED 148

LIST OF FIGURES

Figure Page

1. Solver hierarchy showing the different solving strategies. 13

2. Flow control for a PETSc application. [Source: PETSc tutorial https://www.mcs.anl.gov/petsc/documentation/tutorials] . . . 16

3. Software architecture for multi-method scheme. 22

4. Sequence of operations in the poly-iterative scheme with three solvers: CGS, BiCGStab and QMR. 26

5. Sample LSA Session. 29

6. Linear System Analyzer Solver Scheme Architecture. 29

7. Domain decomposition and splitting of multipliers and cluster formation in FETI. 31

8. Comparison of various solve schemes 34

9. Weka workflow for various classifiers for full feature set and reduced feature sets. 42

10. The overall workflow of the linear system solver selection process with the convergence (ML) model. 59

11. Machine learning classifier accuracy comparison for full feature set and reduced feature sets. 60

12. Time taken (seconds) to construct the classifier for machine learning classification. 60

13. Matrix and Vector operations in PETSc. 68

14. Function plot for communication cost comparison for (1) BCGS and GMRES and (2) TFQMR and GMRES. 75

Figure | Page

15. Speedup w.r.t. default solver-preconditioner for $12,288$, $6,144$ and $1,536$ MPI processor counts respectively for the driven cavity problem on a $1,000 \times 1,000$ grid with Grashof$= 100$. 88

16. Ratio w.r.t. average time/solve for $12,288$, $6,144$ and $1,536$ MPI processor counts respectively sorted by average time per solve for the driven cavity problem on a 1000×1000 grid with Grashof$= 100$. 89

17. Ratio w.r.t. average time/solve for $12,288$, $6,144$ and $1,536$ MPI processor counts respectively sorted by average time per solve for the driven cavity problem on a 1000×1000 grid with Grashof$= 1,000$. 92

18. Speedup w.r.t. default solver-preconditioner for $12,288$, $6,144$ and $1,536$ MPI processor counts respectively for the driven cavity problem on a $1,000 \times 1,000$ grid with Grashof$= 1,000$. 93

19. MOOSE Architecture 121

20. Terzaghi's problem of consolidation. 123

LIST OF TABLES

Table Page

1. Subset of PETSc solvers and preconditioners. 18
2. Full feature set comprising of 68 features computed using Anamod [32]. 43
3. PETSc Krylov iterative solvers and preconditioners. 45
4. Reduced feature sets (RS1 and RS2) used for classification 61
5. Number of calls of communication-relevant functions per iteration of various Krylov methods. 65
6. Matrix-vector operations with communication 66
7. Operations with communication in ASM. 77
8. Operations for analytical measurements 78
9. Communication-based ranking of solvers for $p > 258$. 78
10. Combining ML-based predictions with the analytical model ranking for systems arising in the driven cavity application (1000×1000 grid) for a Grashof 100. 85
11. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$ for $1,536$ MPI tasks. 86
12. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$ for $6,144$ MPI tasks. 87
13. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$ for $12,288$ MPI tasks. 90

Table | Page

14. Combining ML-based predictions with the analytical model ranking for systems arising in the driven cavity application (1000×1000 grid) for a Grashof 1,000. 91
15. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$ for $1,536$ MPI tasks. 94
16. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$ for $6,144$ MPI tasks. 95
17. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$ for $12,288$ MPI tasks. 96
18. Reduced Feature for MOOSE and SuiteSparse Datasets 101
19. Convergence model accuracy and build time for 10 -fold cross-validation with RS1 set for SuiteSparse dataset. 109
20. Convergence model accuracy and build time for $66 - 34\%$ train-test split with RS1 set for SuiteSparse dataset. 109
21. Convergence model accuracy and build time for 10-fold cross-validation with RS1 set for MOOSE dataset. 111
22. Convergence model accuracy and build time for $66 - 34\%$ train-test split with RS1 set for MOOSE dataset. 111
23. Classification model "good" accuracy comparison for C5.0 and J48 for SuiteSparse and MOOSE dataset 112
24. Top 5 solvers that were labeled as "good" as a percentage of all the "good" solvers for MOOSE dataset. 113
25. Top 5 solvers that were labeled as "good" as a percentage of all the "good" solvers for SuiteSparse dataset. 113

Table Page

26. Speedup w.r.t default solver time and number of MatMult operations saved. 118

27. Dimensions for original MOOSE matrices. 131

28. Reduced feature set for MOOSE dataset. 134

29. Top 10 good solvers for PETSc with their occurrence percentage in the dataset. 134

30. Time taken (in seconds) for constructing each classifier method using all features T_{All} and two different reduced feature sets T_{RS1} and T_{RS2}. 135

31. Prediction accuracy of each classifier method using full feature set 135

32. Prediction accuracy of each classifier method using Reduced Set 1 features 136

33. Prediction accuracy of each classifier method using Reduced Set 2 features 136

34. Validation with 10-fold cross-validation train-test data split for the dataset with the best classifier J48. 137

35. Validation with 66%-34% train-test data split for the dataset with the best classifier J48. 138

CHAPTER I

INTRODUCTION

Linear systems of equations are commonly used to represent problems from a vast variety of domains, including, but not limited to tomography, computational fluid dynamics, thermodynamics, statistics, electric circuits, quantum mechanics, astrophysics and fossil fuels. These linear systems can be symbolized as $Ax = b$, where A is the coefficient matrix, x is the unknown vector, b is the known right-hand side solution vector. Linear systems are widespread across different areas of scientific research [91, 90], therefore providing accurate and efficient solution methods is an important capability for scientists in these domains. In particular, large sparse linear systems arise in many computational problems in science and engineering.

Over the last several decades, applied mathematicians and computer scientists have developed multiple approaches for solving such linear systems. The traditional approach involves using a single solver, possibly combined with a preconditioner to get the solution. Preconditioning is a technique to modify the existing linear system into a similar system with the same solution yet easier to be solved by the numerical solver methods. The numerical solver can be chosen from a number of available options which fall into two main categories: (1) Direct solvers [30] tend to provide a solution in finite number of steps and (2) Iterative solvers [77] start with an initial guess and tend to improve the solution iteratively and generate successive approximations to the solution. Direct solvers can be computationally more expensive than iterative solvers. Therefore, iterative solvers are a preferable choice for sparse linear systems. Direct solvers and iterative solvers are described in detail in the next chapter.

Narrowing down the solver categories and selecting iterative solvers for sparse systems are not the only decisions to make. Now, the challenge is to identify which solver to use among the numerous iterative solvers available. In addition, the preconditioner to be used in combination with the solver technique is yet another choice that the user has to make. Lastly, the preconditioners have various parameter configurations that can be tuned to change their behavior. As a result, the number of possible options available for solvers and preconditioners that can be used to solve the linear system are further increased.

Selecting a quasi-optimal[1] solver and preconditioner is nontrivial even for experts. Despite the background knowledge, domain expertise, programming skills, grasp of documentation and information about the linear system properties, selecting a quasi-optimal solver-preconditioner pair for any given linear system may not be possible. The main reason is that the best solution is not consistent for a variety of problems occurring in different domains or even different problems from the same domain. In addition, with the development of new numerical libraries, solver techniques and preconditioning schemes, the pool of available solvers and preconditioners is further expanding.

Designing a model that can select a well-performing solver-preconditioner choice for a given linear system can dramatically improve the time to solve the system. A challenge in using a model that generates a single solver suggestion is the reliability of getting a solution from the sole recommendation. For example, consider the situation where the chosen solver technique fails to provide a solution, which defeats the purpose of using a model to ensure better performance. These

[1]quasi-optimal refers to “almost optimal". In Italian quasi means “almost".

problems motivate a different methodology of solving the given problem, namely, an approach that generates multiple suggestions for solving the given linear system.

1.1 Motivation

This research focuses on linear systems of the form $Ax = b$, where $A = [a_{ij}]$ is an $n \times n$ matrix and b is a given right-hand-side vector. Solutions for these systems are the one of the most expensive (time-consuming) part of the computation and obtaining an efficient solution method is even more challenging. This becomes more crucial when the problem size becomes enormous and the solution requires parallel computation resources. Despite the domain knowledge, programming skills, numerical methods background the scientists (library users) may have, it can be difficult or impossible to choose an optimal solution method for a given problem.

New numerical libraries are continuously being developed, adding to the existing pool of linear solver algorithms and implementations. With the advancement and expansion of high-performance computing libraries, they are becoming better in handling more complex problems than their predecessors. However, selecting an appropriate library and using it efficiently are non-trivial tasks. The decision cannot be made based on the algorithms' complexity analysis alone as many Krylov methods may have the same complexity but perform very differently for the same problem. Hence the need to provide better support for the selection of linear system solutions is consistently increasing.

1.2 Research Goals and Approaches

This work proposes techniques to recommend optimal solver methods and preconditioners with their parameter configurations for the users. Since one of the most expensive stages in scientific computations is obtaining the linear system

solution, a good technique would be to skip solving the system altogether and make a decision based on prior learning. The first phase of this dissertation suggests such a technique to select solution methods effectively, without the need for solving a new incoming system for relatively small-scale problems. The approach involves training a machine learning (ML) model on small to medium-sized problems and make suggestions based on problem characteristics alone. This work demonstrates that performance of various solver techniques can be modeled using a small set of structural and numerical properties of the linear systems.

One limitation of the ML-based approach is its applicability to relatively large-scale problems. The ML model requires training data collection for different processor counts. However, a collection of separate training data sets for different processor counts can get extremely expensive or infeasible in most cases. The second phase of this dissertation focuses on identifying an accurate and efficient solution method by modeling the convergence behavior and the communication overhead for parallel Krylov methods for large scale problems. The communication overhead is captured by an analytical parallel scalability model which compares the communication overhead of parallel preconditioned Krylov methods. For sufficiently large linear systems that require high levels of parallelism, the communication-based analytical model gives a scalability ranking of the solvers. The consolidation of the ML model and the communication model enables solver recommendations at different scales of parallelism.

With these approaches, we enable a user to choose a solver-preconditioner pair that will perform well for the given problem, which is otherwise a very challenging task. This dissertation focuses on applying these techniques for suggesting optimal Krylov methods for sparse linear systems arising from

different domains. This work uses the PETSc [4] toolkit which offers a variety of scalable solver methods and preconditioners that can be used for solving scientific applications modeled by partial differential equations.

1.3 Co-Authored Material

This dissertation includes work from previously published co-authored material. This section lists the chapters with the publications and their authors. The beginning of each chapter provides the details on individual contributions.

– Chapter III is based on collaboration work [51, 86] between Elizabeth Jessup (CU-Boulder), Pate Motter (CU-Boulder), Boyana Norris (UO), and myself.

– Chapter IV, V are based on a collaboration [87] between Elizabeth Jessup (CU-Boulder), Boyana Norris (UO), and myself.

– Chapter VI is based on work-in-progress collaboration work between Boyana Norris (UO), Ben O'Neill (RNET Technologies Inc.), Elizabeth Jessup (CU-Boulder) and myself.

1.4 Dissertation Outline

This document is organized as follows. The next chapter presents background useful for understanding the rest of this dissertation. The next chapter describes various techniques for solving linear systems. Chapter III describes the convergence model that uses machine learning techniques to classify different Krylov methods based on their computation time. Chapter IV discusses the communication model we develop for modeling the performance of parallel preconditioned Krylov methods. In Chapter V results obtained by using the convergence model in conjunction with the scalability model are presented. Chapter VI showcases the approaches used for feature computations using Anamod,

PETSc and finally the matrix-free approach. Chapter VII demonstrates our workflow for classifying arbitrary sparse linear systems using different-sized feature sets for a completely new multi-physics application domain– MOOSE. The last chapter delivers the conclusions and future work of this research.

The rest of this chapter briefly describes the main contributions of this research: (1) Convergence model (2) Communication model (3) Matrix-free feature computation (4) Domain-specific use case - MOOSE application.

1.4.1 Convergence model.

The performance of different Krylov methods can be modeled using an ML model based on the convergence behavior of the solver methods. We use matrices from the SuiteSparse Matrix Collection [24], formerly known as the University of Florida Sparse Matrix Collection, for training the convergence model. The linear systems obtained from this collection belong to different domains. These systems are solved with multiple solver-preconditioner combinations and are used to train the ML model. The next step involves computing various properties of these systems. These features are the structural, numerical and spectral properties of the input linear system, such as the number of rows, number of non-zeros per row and row variance. All these features together constitute the full feature set.

Feature computation is the most expensive stage in the entire process. To reduce the overall cost of the process, a feature reduction technique is applied where a small number of features, referred to as the reduced feature sets are selected from the full feature set. Once the features are computed, the linear systems are solved and their convergence time is recorded. Based on the convergence time of all the solver-preconditioner pairs, for each linear system, a binary labeling scheme is used to identify “good” and “bad” solvers. Next, the data

set is prepared for performing classification by combining the feature values, class label and a unique solver id, which together serve as the input for the ML-based convergence model. The convergence model is described in detail in ChapterIII.

1.4.2 Communication Model. An analytical communication-based approach is proposed for problems that require high levels of parallelism (>1000 cores). The communication model is useful for capturing the performance of parallel preconditioned Krylov methods. The performance modeling is achieved by modeling the parallel overhead based on analytical communication estimates for various Krylov methods. To rank these methods based on scalability alone, the communication-based approach analyzes the differences in the amount and type of communications in each of the methods, when solving the same problem on the same number of processors. The model generates a scalability ranking of the solver techniques and preconditioners by identifying the matrix-vector operations that are communication-intensive. These operations are analyzed for identifying their communication cost and counting the number of times these operations have been performed in solver and preconditioner implementation.

The number of matrix-vector operations, together with each operation's communication cost gives the total cost of communication for each solver-preconditioner pair, which can then be compared with every other solver-preconditioner pair to generate a comparative scalability ranking. The communication model does not capture the convergence behavior and only models the communication behavior of the solver techniques. The communication model is described in detail in Chapter IV. The convergence behavior can be modeled by combining this analytical scaling technique with the supervised machine learning approach. The two models when used together can enable solver recommendations

at different scales of parallelism. The combined model approach is described in further detail in Chapter V.

1.4.3 Matrix-free Feature Computation. The machine learning approach uses the linear system properties in addition to the binary label, and a unique solver id as input. These properties are referred to as matrix features in the rest of this document. There are multiple approaches to compute matrix features. In the first phase of this research, the features were computed using an open source software package, Anamod [32]. In the second phase, feature computation was achieved with PETSc in C, to eliminate the use of the external library Anamod. In the final phase, the features are computed using a matrix-free approach.

Sparse systems have an advantage over the dense systems because of their nature, i.e. most of the elements are zero in a sparse system. This property is exploited to store only the non-zero elements of the sparse systems. A conventional way of storing these systems is in the form of a 2D structure, where each non-zero matrix element is represented by an element in the 2D structure. The elements are accessed by their row and column indices. When all the elements of the matrix stored and available at any given time, various properties of the matrix can be computed, such as matrix norm, diagonal of the matrix inverse, trace, and others.

For an extremely large matrix, storing it and performing matrix operations can be very expensive due to memory cost and computation time respectively. When the coefficient matrix is not available, a contemporary approach involves accessing the matrix by computing matrix-vector products. In this technique, the coefficient matrix is not assembled explicitly. For the past few years, this approach has been used for computing the properties such as trace, diagonal of the matrix inverse and norm. However, to the best of our knowledge, properties such as row

and column variance, infinity norm, column variability ($\max_j \log_{10} |\frac{\max_i |a_{ij}|}{\min_i |a_{ij}|}$ where i is the row and j is the column index), have not been computed in the past. Therefore, an approach that does not require explicit storage of the coefficient matrix, would support even larger problems, which otherwise have high storage costs and cause challenges for matrix operations.

1.4.4 Domain-Specific Use Case. The ML-based approach for small-scale is tested on a set of use cases based on a single framework, the Multi-Physics Object-Oriented Simulation Environment (MOOSE) [46], which is a finite-element, multi-physics framework that leverages other toolkits, notably PETSc. MOOSE aims to make predictive modeling accessible and scalable. MOOSE simulations include problems from computational fluid dynamics, high energy physics, computational biology, and computational finance.

The focus is primarily on iterative Krylov methods and preconditioners to solve the sparse linear systems obtained from the MOOSE matrices. The dataset consists of problems from a single domain and is generated using the sample applications in MOOSE. We instrumented the MOOSE code to save the matrices (linear systems) that are being solved with KSP solvers in PETSc. For expanding the data set, we varied the size of the MOOSE example problems and auto-generated bigger problems. We enlarged the mesh and run in parallel to produce more realistic use cases. As a result, we have up to three- dimensional meshes, 80,802 rows and 11,826,432 number of non-zeros.

In this work, we provide a new set of features of the linear systems which are comparatively less expensive and compute them using the PETSc toolkit. We use PETSc to compute features, which removes the dependence on an external library for feature computation. This work also includes automated input scaling

and generation from the MOOSE test suite. To our knowledge, this work is the first attempt at automated solver selection in this domain.

For feature selection, in applications with dynamic mesh adaptation, such as the Finite Element Multiphysics domain, we consider the structure of the mesh changes at run time and therefore the best solver method could depend on physical and geometric properties of the mesh. Such properties become the basis of the kind of features we compute for these problems. The full feature set contains features which belong to different categories namely, simple or norm-like quantities, variability and structural. As using all the features of a linear system makes it expensive to solve an incoming linear system, we perform feature reduction as mentioned earlier in this document.

The solver selection capabilities of our approach are particularly useful to the target MOOSE users, who are scientists who do not have in-depth knowledge of computer science and would like to develop an application by leveraging the "plug and play" component organization of the MOOSE simulation platform. This work provides a demonstration of an accurate, generalizable, machine learning-based workflow for classifying arbitrary sparse linear systems using different-sized feature sets for a completely new application domain. The classification approach is applied to a set of examples in the MOOSE framework, achieving high accuracy when targeting problems in the more limited domain of finite element multi-physics applications.

1.5 Summary

This research enables iterative solver recommendations for sparse linear systems by modeling the convergence behavior and the parallel overhead for parallel preconditioned Krylov methods. This document describes our ML-based model

to capture the computation aspect and the analytical approach that captures the parallel overhead of various Krylov methods. The ML-based model generates a set of "good" solvers for a linear system and the communication-based model generates a scalability-based ranking for different Krylov methods. We suggest the recommendations made by the ML model for cases where computation aspect is enough to make the decision. For cases where modeling both computation and communication is useful, we combine the ML suggestions by finding the top-ranked methods within that set of solvers. With this approach, both aspects of parallel Krylov method can be modeled: convergence behavior and parallel overhead. The communication-based ranking is validated by comparison with empirical results on a numerical simulation of driven fluid flow in a cavity. The suggested ML-based approach when combined with the comparative performance modeling approach, improves the quality of the recommendations, resulting in improved performance at different scales.

CHAPTER II

LINEAR SYSTEM SOLUTION SCHEMES

Linear systems can be categorized as either sparse or dense systems. The linear systems with majority of the matrix elements as zero are known as sparse linear systems. The systems with most of the elements as non-zero are referred to as dense systems. A common approach of storing sparse matrices involves storing only the non-zero elements, along with their row and column indexes. For dense systems, all elements need to be stored, including the zero elements. Problems from various application evolve continuously in time and can be well represented by large sparse linear systems. Therefore, in this dissertation, the main focus is on sparse linear systems.

This chapter describes the two main categories of solvers: direct and iterative and presents some popular solver techniques which include (1) single-method solver schemes, where only one solver is applied for solving the system, (2) multi-method solver schemes, where more than one solvers are used during different solve stages.

2.1 Motivation

The solution of large sparse linear systems of the form $Ax = b$, where $A = [a_{ij}]$ is an $n \times n$ matrix and b is a given right-hand-side vector, is an elementary problem in scientific computing. Advancements in domains such as multi-physics, aerodynamics, and others, where the problems can be formulated as partial differential equations, rely heavily on the efficient solution of the linear systems. Frequently, the total time in such formulations is predominated by the time taken to solve the linear systems.

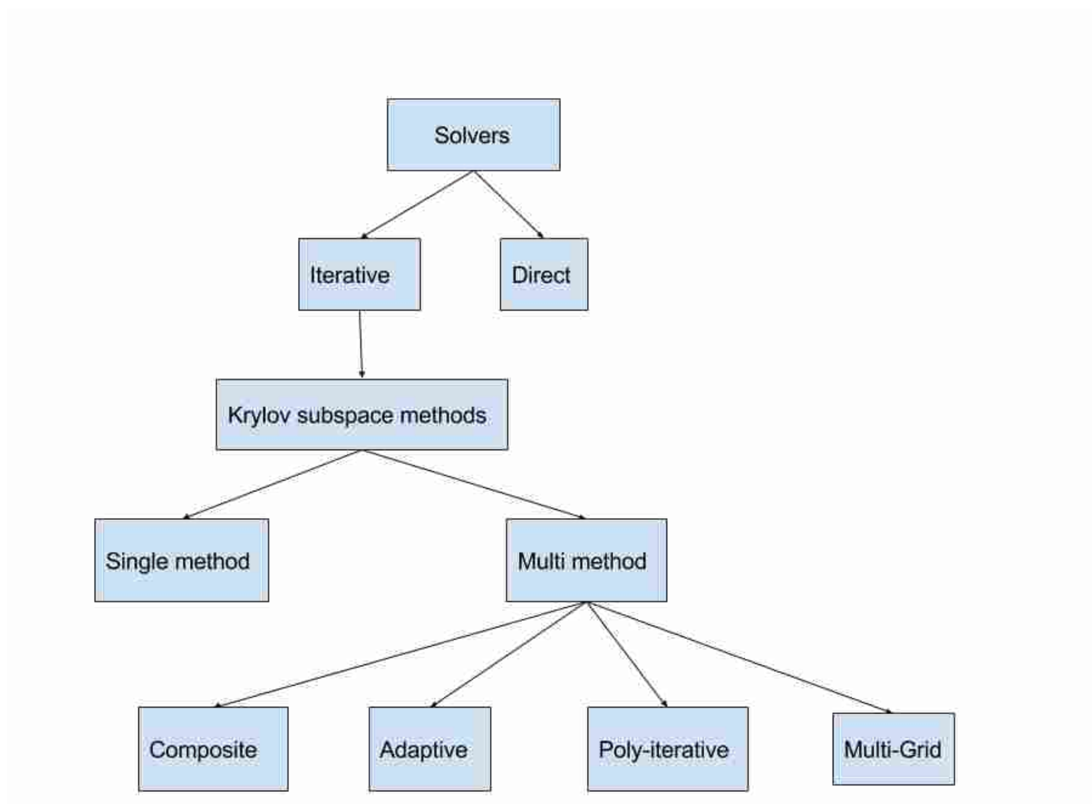


Figure 1. Solver hierarchy showing the different solving strategies.

The general strategy for solving a sparse linear system of the form $Ax = b$, involves transforming the system, A into a similar system which has the same solution, x and that is easier to solve. Such a transformation is applied by using various preconditioners. Preconditioners are discussed in more detail in Section 2.4. There are two popular techniques for solving the systems, namely, direct and iterative solving strategies. Direct solvers have high numerical accuracy and work even for sparse matrices with irregular patterns. Iterative solvers use an initial guess to get an approximation of the solution. For a given approximation solution x_{k-1}, the solution, x_k, at the next iteration is expected to be better. Iterative solvers keep updating the solution until it gets close enough to the actual solution. Figure 1 shows the hierarchy of the sparse linear solvers, which are discussed in detail in the later sections.

2.2 Direct Solvers

For linear systems $Ax = b$, where A is the coefficient matrix, x is the unknown vector and b is the right-hand side vector, direct solvers [30, 23] provide an exact solution, $x = A^{-1}b$ for the linear system and are more robust than the second category of solvers– iterative solvers. Direct solvers are preferable for small matrices and in cases where an exact solution is possible and preferred. However, they are less desirable for very large matrices because of their high costs, because the memory requirement for direct solvers can be huge, as direct solvers need the entire matrix to be in memory. For the work presented throughout this dissertation, we focus on sparse matrices. For sparse matrices, most of the elements are zero, therefore storing the entire matrix can be avoided and is rather expensive.

2.3 Iterative Solvers

The second class of solvers is known as iterative solvers. Iterative solvers provide an approximation of the solution. An iterative solver approach starts with an initial guess and generates successive approximations to the solution. In cases of large linear systems, iterative methods are often preferable for two reasons. First, an exact solution for the systems may be too expensive and second, recurrently, a solution approximation is acceptable. The traditional approach of solving large sparse linear systems involves using a solver combined with a preconditioner. There are many solver techniques that have been in existence for solving large sparse linear systems of the form $Ax = b$ where A is the sparse matrix, x is the solution vector and b is the right-hand side vector (known vector). The residual vector r, can be given as $r = b - Ax$. The aim of iterative solvers is to reduce the residual vector as much as possible.

One of the popular class of iterative numerical solvers is the Krylov subspace methods. Krylov subspace methods start with an initial guess and generate a sequence of approximate solutions, which tend to improve with the progression of iterations. Krylov methods form a sequence, called the Krylov sequence shown below:

$$K_k(A, b_0) = span\{b_0, Ab_0, A^2b_0, \ldots, A^{k-1}b_0\}$$

Here A is a $n \times n$ matrix, b is a vector of dimension n, k is the order of the subspace, b_0 is an initial vector of successive matrix power times the initial residual (the Krylov sequence). The subspace is the successive powers of the matrix A starting from 0 to $k - 1$ applied to the residual form. The approximations to the solution are then formed by minimizing the residual over the subspace formed.

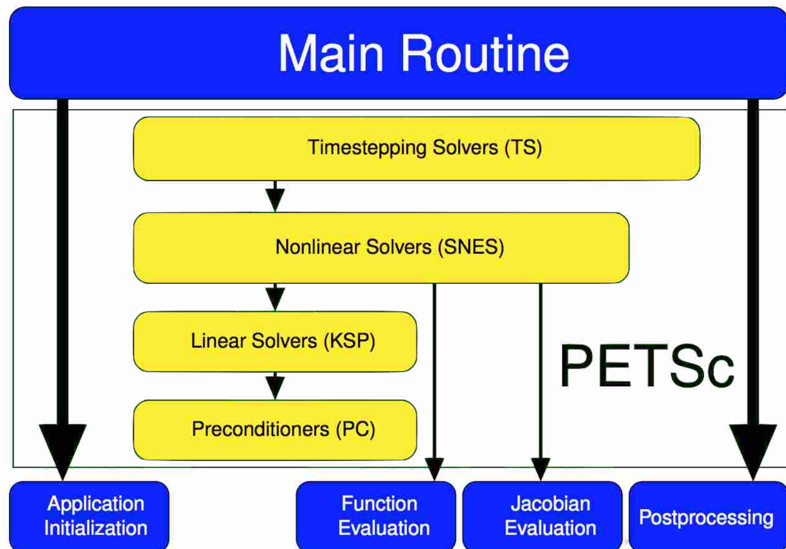


Figure 2. Flow control for a PETSc application. [Source: PETSc tutorial https://www.mcs.anl.gov/petsc/documentation/tutorials]

They are considered to be desirable for solving linear and non-linear systems because of their efficiency and reliability.

2.4 Preconditioning

The general strategy of solving a linear system involves transforming the system into another system which is easier to be solved and has the same solution x as the original system. The transformation process is known as preconditioning [6]. Preconditioning is applied by combining a solver method with a preconditioner. One such transformation is pre-multiplying a linear system with a non-singular matrix. In other words, multiplying the left-hand side and the right-hand side with a non-singular matrix P. The multiplication process leaves the system unaffected. The system transformation is shown below:

$$Ax = b$$

$$PAx = Pb$$

$$x = (PA)^{-1}Pb = A^{-1}P^{-1}Pb = A^{-1}b$$

For a given linear system of the form $Ax = b$, the rate of convergence of a Krylov subspace method depends on the condition number of the matrix A. The matrix P has a smaller conditioner number than the original matrix A, therefore, it is expected that the solver method will converge faster for the preconditioned system. Preconditioning in iterative solution of linear systems can be applied in the form of left preconditioning or right preconditioning. In the case of left preconditioning, the preconditioned system is shown below:

$$P^{-1}Ax = P^{-1}b$$

In case of right preconditioning, the system can be shown as:

$$AP^{-1}u = b, \text{where } x = P^{-1}u$$

As shown above, the system of equation changes from $Ax = b$, to $P^{-1}Ax = P^{-1}b$. The transformed systems can be more easily solved because of the change in the condition number. The original matrix A, had a higher condition number than the transformed system $P^{-1}A$. A system with a higher condition number is more ill-conditioned than a system with a lower condition number. The convergence rate of iterative solvers increases with a decrease in condition number. Therefore, selecting a suitable preconditioner is equally important as selecting a suitable solver.

This dissertation focuses on using solvers and preconditioners offered by PETSc [4]. Portable Extensible Toolkit for Scientific Computing (PETSc) is a widely used toolkit for linear systems, developed at Argonne National Laboratory.

Table 1. Subset of PETSc solvers and preconditioners.

KSP Methods	Preconditioners
GMRES [75]	Cholesky [78]
Flexible GMRES(FGMRES) [74]	ASM (0,1,2,3) [18]
PipeFGMRES [80]	GASM (0,1,2,3) [29]
LGMRES [3]	SVD [55]
DGMRES (Deflated GMRES) [36]	Jacobi [92]
Conjugate Gradient [50]	Block Jacobi [84]
Flexible Conjugate Gradient (FCG) [62]	SOR [48]
PipeFCG [81]	LU [78]
LCD (Left Conjugate Direction) [99]	ILU (0,1,2,3) [31]
Hypre [38]	ICC [19]
GCR [35]	
TFQMR [43]	
Richardson [72]	
Chebyshev [47]	

PETSc is a collection of data structures and functions for the scalable (parallel) solution of scientific applications and offers solver techniques and preconditioners for linear and non-linear equations. PETSc can run on different architectures, various operating systems and is portable to any parallel system that supports MPI. PETSc is widely used for modeling small-scale and large-scale applications and is considered to be a highly efficient toolkit. PETSc offers scalable solutions for scientific applications ranging from brain surgery [40], cancer treatment [45], earthquakes [95], ocean dynamics [53], among many others. Figure 2 shows the flow control for a PETSc application. Table 1 enumerates the subset of solvers and preconditioners offered by PETSc, that are used in the work presented throughout this dissertation.

2.5 Single-Method Solver Systems

In this approach, only one method [30, 65, 64, 50, 75, 93, 47, 74, 60, 94] is used to solve the given linear system. If the applied method fails to solve the

system, there is no other solving technique used to solve the system. Depending on the problem, either direct or iterative methods can be used in the Single-Method solver scheme. Iterative solvers can be used accordingly for small and large problems. As the problems tend to grow, iterative solvers become a preferable choice. However, in some large-scale applications, direct solvers are used because of non-familiarity with the iterative solvers.

A single-method solver scheme is useful in situations where the solver technique to be used is pre-established or decided based on prior information or experience. One of the disadvantages of using a single solver is that the numerical properties of a system can change during the course of the nonlinear iterations and the single-method solver scheme does not take that into consideration. On the other hand, a multi-method solver (explained in more detail in Section 2.6), uses multiple solvers instead of using a single solver for solving the system. The multi-method approach, makes it complex as the number of decisions to be made are more, for instance, which set of solvers should be used as base methods, when should a new solver be applied, which solver should be applied next, when can a solver be eliminated from the list of base methods.

In a single solver scheme, the choice of the solver method is made by experts or resources available for a selection. However, in many cases, there is often no single solver that is consistently better, even for problems from a single application domain. There is also no guarantee that the solver technique used to solve the system will eventually converge. These challenges generated the idea of a solving strategy that involved more than one solver algorithm. The earliest work [71, 69] which suggested that the efficiency of a system is expected to improve with polyalgorithm solvers, used three basic solvers. The work presented in [69] provides

the design details of a polyalgorithm for automated solution of the equation $F(x) = 0$.

2.6 Multi-method Solver Systems

The second approach for solving a given system involves using multiple solvers (a composite of suitable solvers) [9, 88, 79, 33], instead of a single solver. If two or more solvers are used instead of one, the chances of getting to a solution increase. Further, there are different techniques of using multiple solvers for solving sparse linear systems: composite solvers, adaptive solvers and poly-iterative solvers. These techniques are discussed in detail later in this section. In this section, we talk about these different types of multi-method solvers approaches namely composite solvers, poly-iterative solvers and adaptive solvers. Multi-method linear systems include a variety of techniques like composite solvers, iterative solvers and adaptive solvers.

2.6.1 Composite Solvers. In a composite solver approach, basic solver methods are sequenced in an ordered fashion. The first choice for solving the system is the first solver method in the sequence. If the method fails, then the next method in the sequence is invoked. Switching to the next solver in the sequence continues until the linear system is solved successfully. Since the composite algorithms [8, 11, 12, 13, 1] use multiple solver methods for obtaining the solution, the probability of solving the systems increases, thereby making the approach more efficient and robust. The research presented in [8] uses multiple preconditioned iterative methods in sequence to provide a solution. The solution obtained by this strategy is believed to be reliable and have a good performance in parallel.

The work presented in [1, 8], mention that the reliability of a solver scheme can be given as r_i = 1 - f_i, where f_i refers to the failure rate of the solver. The run time of the composite scheme depends on the sequence of the solver methods. The worst case time scenario (T_π) for the composite scheme occurs when all the solver techniques in the given sequence have to attempt to solve the system. With the base methods $S_1, S_2, S_3 \ldots S_n$, the total time required in this scenario can be given as follows:

$T_\pi = t_{\pi(1)} + f_{\pi(1)}.t_{\pi(2)} + f_{\pi(2)}.t_{\pi(3)} + ...(f_{\pi(1)}...f_{\pi(n-1)})t_{\pi(n)}.$

Here, $f_{\pi(1)}, f_{\pi(2)}, \ldots f_{\pi(n)}$ are given as To have minimum worst-case running time among all the possible combinations possible, the base methods are arranged in the sequence in the increasing order of their utility ratio, u_i which is given by the ratio t_i/r_i. For computing this ratio, r_i is substituted as shown below and using estimates of t_i with some sampling technique: r_i = 1 - f_i. The composite solver technique uses the knowledge obtained in the past, which enables using domain-specific knowledge for the selection of solvers. The system maintains the past performance history and allows monitoring system performance. The solvers are then arranged in the increasing order of their utility ratio, u_i. They use a simple sampling technique for the optimal composite by computing this ratio from running all the solver methods in the sequence on a small dataset and obtaining the mean of the time taken per iteration by the solvers and the failure rates.

The software architecture that supports this strategy is shown in Figure 3. The architecture diagram has the following components: solver proxy, non-linear solvers, linear solvers, ordering agent, and application driver. The proxy linear solver method acts as an intermediate between the non-linear solver algorithm and linear algorithm. The proxy, linear solvers and non-linear solvers have the same

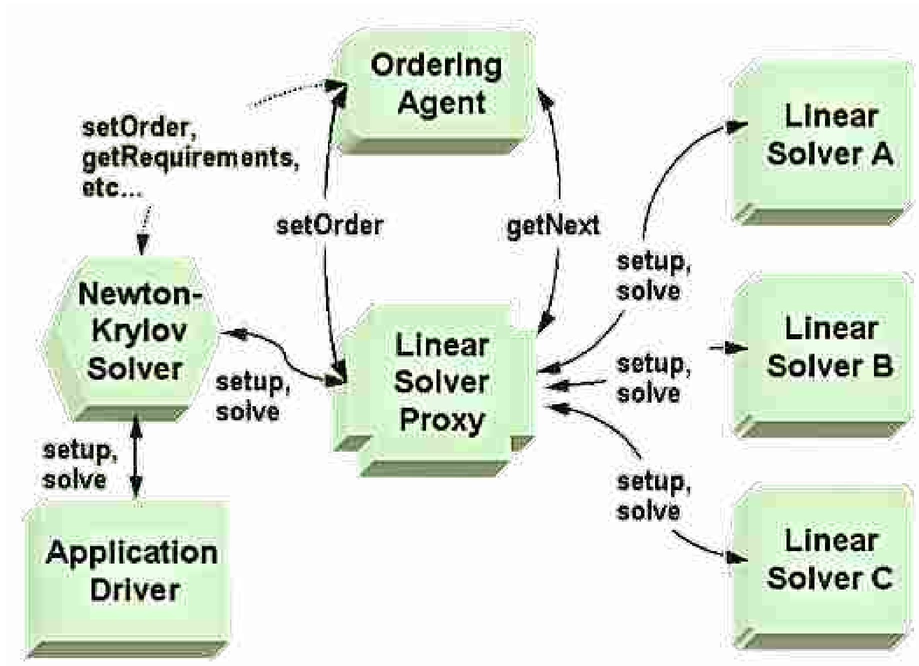


Figure 3. Software architecture for multi-method scheme.

solver interface to make it easy to use multiple solvers. The proxy interacts with the ordering agent to choose the linear solvers based on the ordering strategy. The proxy is used with Newton-Krylov solver. The following four set of base solution methods are used: (1) GMRES(30), restricted additive Schwarz method (RASM)(1) with Jacobi subdomain solver (2) GMRES(30), RASM(1), with SOR subdomain solver (3) TFQMR, RASM(3) with no-fill ILU subdomain solver and (4) TFQMR, RASM(4) with no- fill ILU subdomain solver. The numbers in brackets denote the degree of overlap.

2.6.2 Adaptive Solvers. In this approach [58, 25, 26, 10, 34] only one solver is used by selecting the most suitable solver dynamically, based on the match of the solver with the characteristics of the linear system under consideration. The technique adapts the solver method during a simulation, based on the changing attributes of the problem. The advantage this approach has over

the composite solve approach is that it uses only one base solver for each linear system.

Numerical properties for a system change at each iteration of solving simulation and so does the choice of the solver and the selection criteria. In [58], linear solvers are selected at each iteration based on the characteristics of the problem emerging at each level and given the nature of the problem at that stage, the decision of the best-suited solver is taken at each stage. The solver strategy presented in [58] applies a different preconditioner during different simulation stages while maintaining a low overall time for finding the solution. With this approach, the chances of getting a solution are increased, as there are robust methods involved and also the total time for the solution is acceptable as well. The authors use GMRES(10) with a point-block ILU(1) preconditioner in their work.

The work described in [10] is an extension of the previous work, to solve a more complex parallel application to demonstrate that the adaptive poly-algorithmic approach is parallelizable and scalable. The four linear solvers used are as follows:

- GMRES with a block Jacobi preconditioner and SOR as a subdomain solver, called GMRES-SOR.
- Bi-Conjugate Gradient-Squared (BCGS) with a Block Jacobi preconditioner and no-fill incomplete factorization (ILU(0)) as a subdomain solver, called BCGS-ILU(0).
- Flexible GMRES (FGMRES) with a Block Jacobi preconditioner with ILU(0) as a subdomain solver, designated as FGMRES-ILU(0).

– FGMRES with a Block Jacobi preconditioner that uses ILU(1) as a subdomain solver, called FGMRES-ILU(1).

The transition of solvers is made based on the following two indicators:

– The nonlinear residual norm is calculated and assigned to the following four categories: (a) $||f(u)|| \geq 10^{-2}$, (b) $10^{-4} \leq ||f(u)|| < 10^{-2}$, (c) $10^{-10} \leq ||f(u)|| < 10^{-4}$, and (d) $||f(u)|| < 10^{-10}$. A new solver is chosen when the simulation moves from one category to another and at that point, the solver method is moved up or down accordingly.

– Average time per nonlinear iteration: The base solver methods are arranged in increasing order of their corresponding average time per nonlinear iteration values.

On the other hand, the research [25] has a different approach. It uses statistical data modeling for making the solver choice automatically. They combine different solver techniques with different preconditioners and different parameters as well.

2.6.3 Poly-iterative Solvers.

Poly-iterative solver approach uses multiple solvers applied simultaneously to the system so that the chances of getting a solution increase. If one solver fails, one of the other solvers from the system can provide a solution. One of the earliest suggestions made in [71] for poly-iterative solver strategy was made in the late 1960's. The Poly-iterative solver strategy is based on selecting the solver based on the problem size and the user's specifications about the problem and the accuracy level expected from the system. If no information is specified by the user, then the size is used to pick the solver. If it is a small matrix, with less than 15 rows and 15 columns, the solver chosen is LU decomposition. If the LU decomposition method fails, then the solution obtained

just before the failure is taken as the initial guess for SOR. If the SOR method also fails, the user is provided with the summary and prompted for further instructions to solve. The user may lower the accuracy level to accept somewhat less acceptable solution or allow longer computations for solving the systems.

For problems of large size (considered large in that decade), more than 80 rows and columns, SOR is applied. If SOR fails, SOR is applied again but this time on the product of matrix transpose and the original matrix. If this strategy fails then the user is asked for further instructions similar to the small matrix scheme. For problems with intermediate size, properties namely bandedness, diagonal nature is investigated and if either of these properties is valid for the problem in consideration, SOR is applied. If SOR fails, then LU decomposition is tried. If LU fails as well, then the system relies on the user feedback for accepting lower accuracy level or allowing longer computations.

In the work presented in [5], the authors mention the advantages of using a poly-iterative approach [71, 70, 37] in parallel. Firstly, the approach has an increased probability of finding a solution. Secondly, an increased performance resulting from an efficient matrix-vector product can be obtained. In addition, once any one of the solver methods has converged, the process can be terminated. Their algorithm uses three solver techniques, namely QMR, CGS, and BiCGSTAB. These methods start computing the inner product, then perform the vector updates and finally a preconditioner solve. All these methods are applied simultaneously and as soon as one of them converges, the iteration is stopped for all other methods. The cost per iteration can be given as the sum of the cost of the three methods. In case if a method fails, it is removed from the iterative scheme. Although the poly-iterative strategy takes more time than the best method, the strategy is

preferred as it has a higher probability of finding the solution. Another situation in which this method incurs higher cost is when one of the methods is comparatively more expensive than the others and it is not the first method to converge nor it fails. However, this strategy is more beneficial in parallel implementation as this approach aligns the mathematical operations of the solver methods and combines the communication stages to make it more efficient. Figure 4 shows the sequence of operations and how communication is combined.

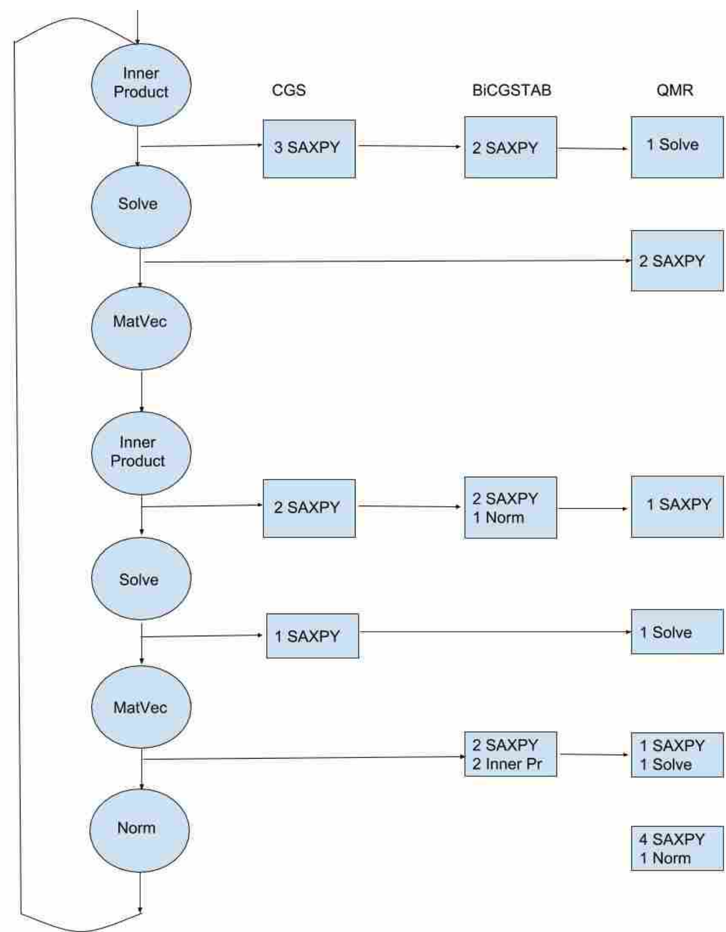


Figure 4. Sequence of operations in the poly-iterative scheme with three solvers: CGS, BiCGStab and QMR.

2.6.4 Self Adapting Solvers Large-Scale Solver Architecture (SALSA) Solvers.

SALSA [27] is a self-adapting solver technique which has several levels on which the computational choices for the application scientist is automated. The choice of solver technique can be made based on the nature of data and on the efficiency of the available kernels on the architecture under consideration to facilitate tuned high-performance kernels. One of the advantages of the scheme is that it is expected to increase its intelligence gradually. SALSA remembers the results of the runs and learns over time. There are three levels of adaptivity:

1. Kernel level: It can be done in a one-time installation and is independent of the data given by the user.
2. Network level: Some level of interaction with user data.
3. Algorithm level: At this level, analysis is done dynamically based on the user data.

2.6.5 Linear System Analyzer Solvers.

The Linear System Analyzer(LSA) [15] is a component-based problem-solving environment for large sparse linear systems. The components LSA provides are broadly categorized into four categories: IO, Filter, Solver, and Information. IO is for feeding the problem into the system and getting the solution out of the system. The user also feeds various parameters and settings for solving, such as the relaxation for solving and which solver to be used. Although this system takes a lot of input from the user apart from the problem to be fed as input, it provides settings to choose default parameters for the various solving techniques and other settings shown on the interface. Figure 5 shows a sample LSA session. The 'filter' is for providing filtering

of data or system manipulation in other words, such as scaling, eliminating entries based on the size. The 'solver' is for actually getting the system solved. LSA offers for choices to the users to use for solving. They are as follows:

1. Banded: A matrix has a banded structure if its rows and columns can be permuted such that the non-zero entries form a diagonal band, more like exhibiting a staircase pattern of overlapping rows. It converts the system to banded structure and solves the system, with the new data structure and uses LINPACK [28] routines for solving. LINPACK, LINear algebra PACKage is a Fortran package developed in the 1970's. It uses BLAS as the underlying routine.

2. Dense: It converts the system into a dense 2D array data structure and then solves it using Lapack [2] routines. system to a dense 2D array data structure.

3. SuperLU: SuperLU [57] is a solver library for getting direct solutions for large, sparse, non-symmetric systems of linear equations.

4. SPLIB: They use preconditioned iterative solvers offered by SPLIB library [16]. The library had 13 solvers and 7 preconditioners when this research was performed.

Figure 6 shows the LSA architecture with the four components namely, user control, manager, communication subsystem, and information subsystem. This approach provides parallelism between components, which supports solving large problems by simultaneously using the computational resources of multiple machines. The system allows comparisons of different solver methods and support to facilitate practical solution strategies.

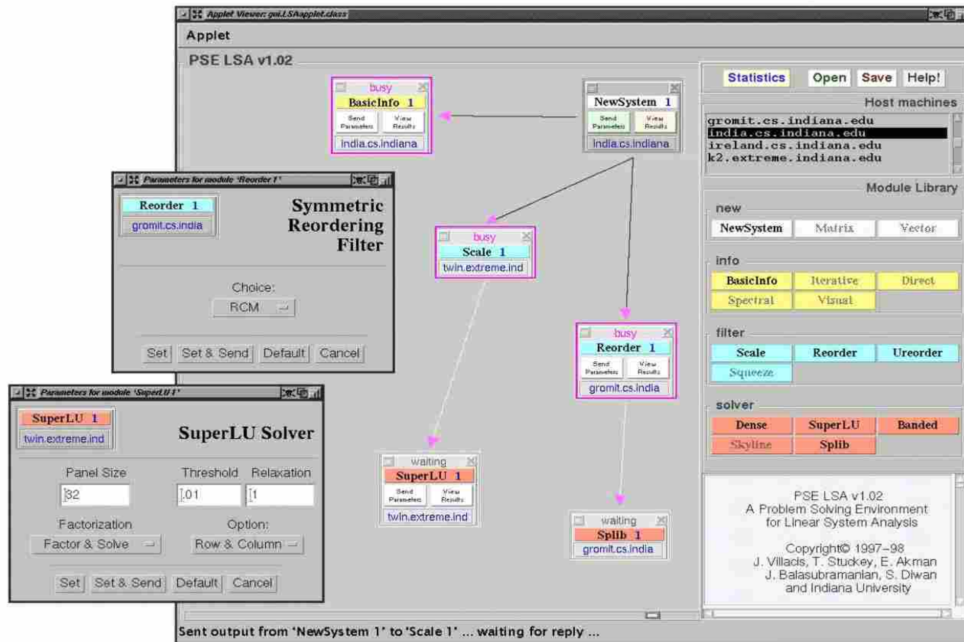


Figure 5. Sample LSA Session.

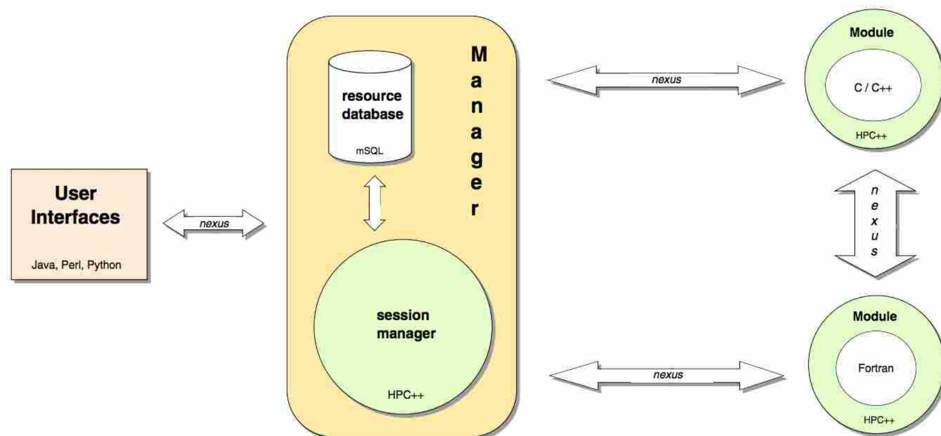


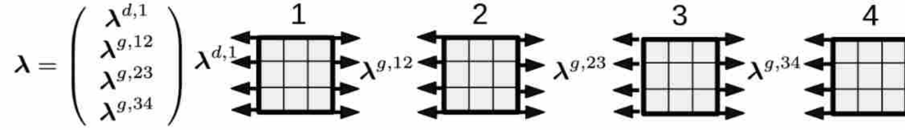
Figure 6. Linear System Analyzer Solver Scheme Architecture.

A new system is fed as input in the first component in the module and in the meanwhile, the matrix is scaled and reordered, both simultaneously on different machines. The input feeding and matrix operations are performed through the user control module, which also has options for choosing parameters for all the modules in the interface. The scaled problem is sent to SuperLU and the reordered version is sent to SPLIB where SuperLU and SPLIB are the two component subinterfaces. LSA has the option of running multiple solvers on a single system in order to compare these techniques and use them for research purposes.

The LSA manager collaborates control and resource management. It establishes a component network to facilitate multiple user control systems with a single LSA session. It also assigns unique identifiers and maintains the database of various machines and components. The next module is the communication subsystem, Nexus [42], which is a cross-platform system for facilitating parallel applications and distributed computing. Nexus provides the bridge between the different languages used in LSA. LSA uses a bunch of libraries for solvers and preconditioners that are written in different programming languages and therefore there is a need for a module that handles this and makes it robust as a mixed language system.

The Information subsystem module provides any information that the user may want about the solving process except the undesirable information. The results are shown in the form of a summary with the performance metrics for that scenario. There is a small description provided, along with the details whether the event was successful or failure or there was a warning. The user is also redirected to more information, in case if she wants more details.

a) Domain Decomposition



b) Hybrid FETI - splitting of Lagrang. multipliers

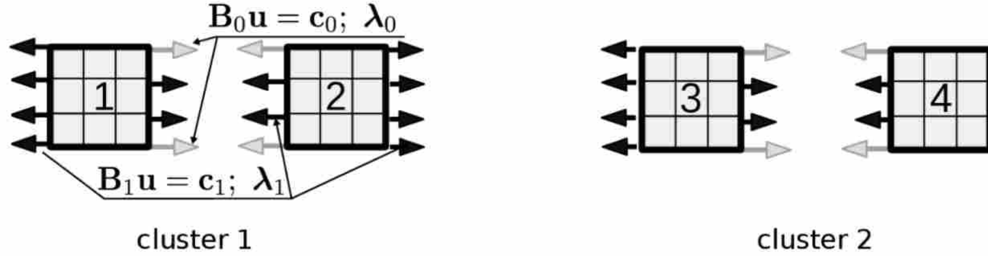


Figure 7. Domain decomposition and splitting of multipliers and cluster formation in FETI.

2.6.6 Finite Element Tearing and Interconnecting and its parallel solution algorithm(FETI). Direct solvers are more suitable for small problems and iterative solvers become preferable in bigger problems. FETI uses a hybrid approach of using iterative solver and then breaks the problem into sub-problems and applies direct solvers on them. The algorithm [39, 73] uses a domain decomposition approach for solving the given linear system for the finite element solution in a parallel fashion. The main problem domain is partitioned into non-overlapping sub-domains. These sub-domains are fully-independent, which makes FETI suitable for parallel computing. Each one of these sub-domains is assigned to a separate processor. These sub-domains are connected later on by using Lagrange multipliers on neighboring sub-domains. Each of the sub-domain is solved by applying a direct solver to solve the unknowns present in that domain. The solution of the sub-domain problems is then parallelized. Such an approach improves the chances of convergence for a given overall. In Figure 7, the first stage

shows the decomposition into four sub-domains and the second stage shows the splitting of Lagrange multipliers and forming clusters.

2.7 Accuracy of Solutions

Either direct solvers of iterative solver techniques can be used to get a solution or an approximation of the solution. The solution of a given system can be verified using different metrics. In this section, we discuss two of the popular metrics used for solution validation:

1. **Residual of a solution**: In order to check the validity of a solution, the easiest way is to plug it in the equation and compare how close the left and right sides of the equation are to each other. The residual vector of a computed solution x' for the linear system $Ax = b$ can be given as:

   $$r = b - Ax^{'}$$

   A large residual implies a large error in the solution. For direct solutions, we want the error E to be equal to zero, which is given by the equation shown below.

   $$E = ||x' - x|| = 0$$

   For iterative solutions, the computed solution is an approximation of the actual solution, therefore we want the error to be as close to zero as possible.

2. **Estimation with the condition number:** Conditioning is a characteristic of a system given by the formula $cond(A) = |A|.|A^{-1}|$. The condition number can determine the possible relative change in the solution for relative changes in the entries of the matrix. Therefore, it can give an estimate of the error

in the computed solution. In other words, changes in the input, i.e., A and b of the equation $Ax = b$, get multiplied by the condition number to produce changes in the output, i.e. x in $Ax = b$. This means small errors in the input operations can cause large errors in the solution of the system. Hence a very large value for condition number for a matrix denotes that the matrix is ill-conditioned. On the other hand, a smaller value implies a well-conditioned matrix.

2.8 Summary

This chapter presents the two categories of solvers, direct and iterative, that can be used for solving large sparse linear systems. Figure 8 shows the various solver techniques discussed in this chapter. Single-method solver technique uses a single solver (direct) throughout the process. This chapter also describes the various multi-method solver techniques. In an adaptive solver scheme, many solver methods are used, although at a time only one solver is applied. The solver scheme changes the solver based on the switching criteria. It runs one solver and then applies the switching check which involves some calculations, such as convergence rate and increase in the number of iterations. The solver switching is applied multiple times, each time the system decides either to use the same solver or switch to a different solving technique. In a poly-iterative approach multiple solvers are applied simultaneously and whichever converges the fastest, terminates the solving process. In the composite solver scheme, the solvers are sequenced in order and everything is preassembled. If the first solver fails, the system switches to the second solver in the order.

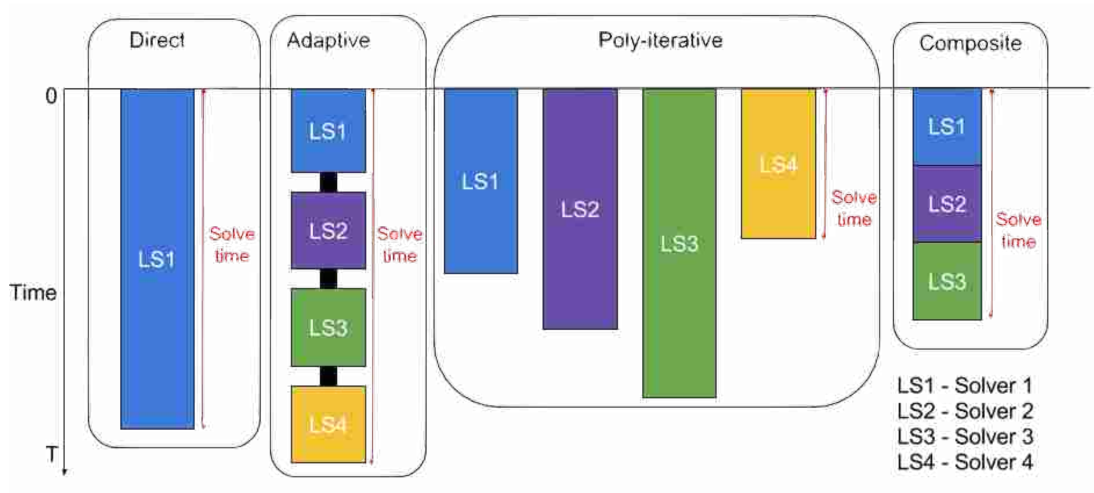


Figure 8. Comparison of various solve schemes

CHAPTER III

CONVERGENCE MODEL FOR SPARSE LINEAR SOLVER CLASSIFICATION

This chapter is based on collaboration [51, 86] among Elizabeth Jessup (CU Boulder), Pate Motter (CU Boulder), Boyana Norris (UO), and myself. For the collaboration [51], Boyana Norris and Elizabeth Jessup provided the strategy to evaluate the performance of the solvers for PETSc and Trilinos. For the collaboration: [51], I created and prepared the dataset for solver classification and extracted the features for PETSc. Elizabeth Jessup and Boyana Norris provided thorough edits for the approach and experimental results sections. I analyzed the features to form the reduced feature sets used for solver classification, performed the solver classification and executed the performance evaluation of the solver classification for PETSc. For the collaboration [86], Elizabeth Jessup conducted the evaluation for the Lighthouse project at the University of Colorado Boulder. I collected the dataset, prepared it for classification, computed the features, performed the classification and validated the performance of the machine-learning models.

This chapter describes our machine learning-based classification technique to select solvers offered by PETSc for sparse linear systems. In this chapter, we present the application of popular machine-learning classification techniques for generating a solver classification model and perform a comparative analysis of the solver classification results for a multi-domain set of linear problems. We investigate feature selection and provide a technique to recommend sparse solvers based on the convergence time of solvers. The technique creates a comprehensive machine-learning based workflow for automated classification of sparse solvers.

3.1 Motivation

With the advancement of time, the volume of data is multiplying manifold. New data is generated by humans, computers, and almost every device that uses technology. Traditionally, humans analyzed data and adapted to the changes in data patterns by writing new rules to the system manually. As the volume of data starts to grow, updating the system becomes excessively difficult. With the increase in the volume of data, there came the need for a technique where the system can learn automatically from the data and adapt accordingly. Machine learning is a class of algorithms that offers the tools and technology that can be used to predict outcomes by using statistical analysis. Typical machine-learning algorithms receive input data, learn from the data and generally improve over time as new data patterns are observed.

Current high-performance linear algebra software is based on years of research and expertise. Using machine learning for automated algorithm selection has been an intensive research topic in many application domains [41, 7, 96, 97]. Solver selection is a non-trivial process and hence requires techniques that can enable users to make a quasi-optimal selection. For automating the solver selection process for Krylov methods, we employ supervised machine learning techniques to classify solvers based on their performance. Machine learning algorithms are trained by providing input characteristics to the model and then used for making predictions. Supervised learning uses pre-classified data for training and needs class labels, unlike unsupervised learning. This chapter presents the machine learning techniques used throughout the rest of this dissertation.

3.2 Dataset

The dataset consists of thousands of linear systems of the form $Ax = b$ arising from different applications. These systems are chosen from the SuiteSparse matrix collection [24]. The collection contains a variety of sparse matrices that are formed in real applications. It is a widely acceptable collection for testing the performance of solver techniques and offers matrices in MATLAB, Matrix-market and Rutherford Boeing format. We use the matrix-market format, which is converted to the PETSc format for convenience. A set of $1,015$ input matrices from the SuiteSparse collection is used and each of them is solved with 154 combinations of solver-preconditioner configurations offered by PETSc. The dataset is split into training and testing sets with a split of $66\%-34\%$ training-testing set. The training set is used to build the classifier and then tested on the test set which has not been seen by the classifier before to ensure fair classification performance. We use supervised learning for building the ML model, which is explained in detail in the next section.

3.3 Supervised Learning

Machine learning techniques can be categorized into two categories, namely supervised and unsupervised learning. Supervised learning approach analyzes the data to determine the mapping function, which establishes a relationship between the input variables and the target variable y. Supervised learning techniques can be further categorized as classification and regression techniques. Regression techniques are used when the dependent output variable has continuous values, for example, stock prices or temperature. When the dependent output variable has categorical values, with limited possible values, for example, days of the week or the blood type of a person. Unsupervised learning technique does not require the label

for the output variable y for training, and rather explores the structure of the data to derive inferences based on the independent variables and dependent variable.

Since supervised learning relies on the pre-classified data for training, we provide the class label for attribute y by solving each of the systems with multiple solver-preconditioner combinations. We use machine learning classification techniques to classify solver techniques as "good" or "bad" for any given system. Machine learning techniques analyze input data to learn and train the model to make predictions in the future. The data comprises one or more independent variables $x_1, x_2, \ldots x_n$, referred to as the input attributes and one dependent output variable y called the labeled class attribute or the target variable. For this work, we primarily use supervised learning techniques by applying different classification based machine learning techniques, described in detail in the next section.

3.4 Machine learning classification techniques

There are many machine learning techniques available for classification problems. In general, a classification technique has a class variable y, and attribute variables $x_1, x_2, x_3 \ldots x_n$. The attribute variables are the independent variables and the class variable is the dependent variable. A classifier $c : x- > y$ is a mapping function that maps an instance of the attribute variables to a value of the class variable, based on the training performed on a dataset.

We use binary and tertiary classification for the solver selection process. With binary labeling, we label solvers as "good" or "bad" based on the convergence time. With tertiary labeling, we assign labels "good", "fair" and "bad". We use some of the popular supervised machine learning techniques for classification, which are described below:

3.4.1 BayeNet.

Bayesian networks are a probabilistic model that computes probability using Bayesian inference. The edges in Bayesian networks represent conditional dependencies in a directed graph. The nodes are the attribute variables and have a probability distribution given their parents in the directed graph. The goal of the BayesNet classifier is to find a mapping function for associating the attribute variables with the class variable. The BayesNet classifier offered by Weka uses the K2 hill climbing algorithm, [20], for searching network structures.

3.4.2 SVM.

SVM is a machine learning algorithm which performs classification by generating a decision boundary, commonly referred to as the hyperplane. To find the optimal hyperplane for classification, the objective function identifies the plane that has the maximum distance between the distinct classes. The dimension of the hyperplane is decided by the number of attributes in the data.

3.4.3 k-nearest neighbor.

k-nearest neighbor is a classification algorithm where k denotes the number of neighbors to be used for classification. The technique assigns the class based on a voting system. The neighbors of the data point are assigned using Euclidean distance estimation. Euclidean distance between any two points x and y is computed as the $|y - x|$. Each neighbor votes for the class for the new test instance and based on the class that gets the majority of the votes is assigned to the test instance.

3.4.4 ADT.

Decision Trees are a popular supervised machine learning algorithm method for classification and regression problems. A decision tree is a flowchart-based algorithm, where the root node is and each internal node represents an attribute or feature. There are two kinds of nodes: decision nodes

and prediction nodes. The leaf nodes represent the class label and each has a numeric value associated with it. The value of the leaf node is the likelihood of that class, provided the values of the attribute variables. The input variable values are represented by the path from the root node to the leaf node. Decision trees implicitly apply feature selection for performing classification by analyzing each attribute and making the best possible inference. The algorithm uses these attributes to split the data into subsets. For a new test point, the algorithm identifies the subset to which the test point belongs, based on its attribute values and the subsets in the decision tree. It assigns the dominant class of that subset to the new test instance.

An Alternate decision tree is a variant of decision tree and is used mainly for classification problems. ADT has alternate layers of prediction node and decision nodes in the tree structure. The root node and the leaf nodes of the tree structure are prediction nodes. The main difference between ADT and Decision tree is the approach to compute the class of the new test instance. In ADT the classification of a new test instance is obtained by including all the paths for which all the decision nodes are true and adding all those predictions along the path that holds true.

3.4.5 Random Forest. Random Forest is a popular classification technique which uses sampling for classifying the data. This technique develops multiple decision trees based on the random selection of data and attribute variables. The class of the dependent variable y is decided by forming multiple trees using a random subset of data and attribute variables. Random forest is a collection of multiple random trees, hence the name Random forest. Each of these trees votes for the most popular class for the input and the class with the majority

of the votes is assigned as the class for the new dependent variable. With the use of multiple decision trees, the probability of correct classification usually improves over other classification techniques.

3.4.6 J48. J48 often referred to as the C4.5 algorithm, is an extension of the ID3 algorithm, used for classification. C4.5 and ID3 techniques using information entropy for generating a decision tree. For each node in the tree, the algorithm picks the attribute variable with the most information gain to make the decision for the subtree. The subtree is formed by using the value that the attribute can have as a descendant node and splitting further down in the same way. The data is sorted based on their values for the attribute. The main difference between ID3 and C4.5 is that the latter uses gain ratio instead of information gain. Gain ratio is the ratio of information gain and split information value. Split information value can be computed as follows:

$$SplitInfo_A(D) = -\sum_{j=1} \frac{|D_j|}{|D|} \times \log_2(\frac{|D_j|}{|D|}) \quad (3.1)$$

Using information gain ratio over information gain helps in reducing the bias towards attributes with large number of values.

3.5 Solver Selection as a Classification Problem

Solver selection can be represented as a binary/tertiary classification problem. Consider a set of linear systems, represented by the matrix A and the right-hand vector b. For finding the solution for the set of linear systems, say M, there are N possible solvers. Ideally, solving each linear system with each solver from the set S would represent an exhaustive dataset for a classification system. Given M input matrices, and N possible solvers, the dataset size would be $M \times N$ data points, which can be prohibitively large, therefore we construct the training set by computing a smaller number of randomly selected points, $P_{i,j}, j \epsilon \{1, M\}$.

For analyzing the classification performance of various ML algorithms, we use Weka [49] to compare the performance accuracy for predicting "good" solvers. We use all the machine learning techniques described in Section 3.4 as we wanted to observe the behavior of each of these algorithms on our dataset. Figure 9 shows the Weka knowledge flow components we defined and used to generate the results presented in this chapter.

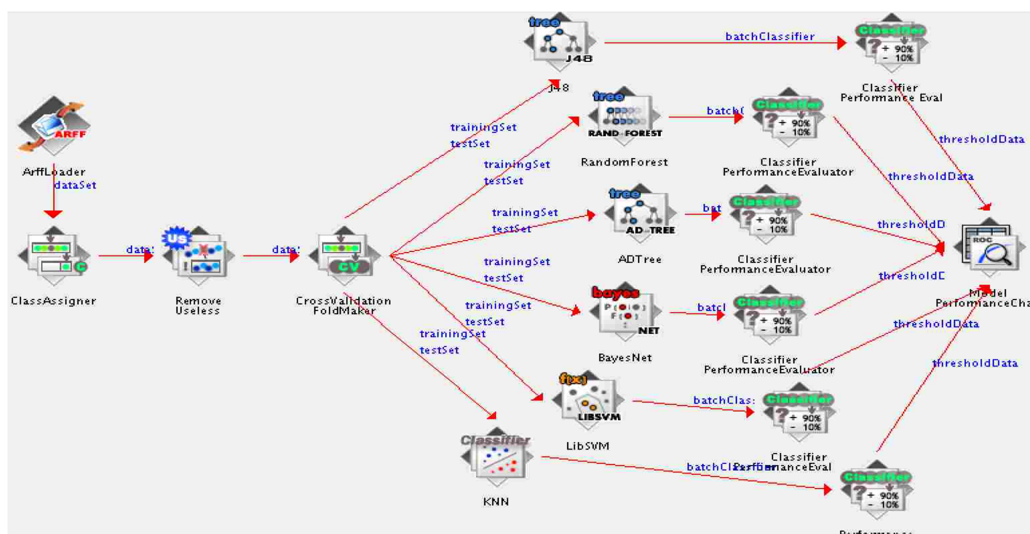


Figure 9. Weka workflow for various classifiers for full feature set and reduced feature sets.

3.6 Feature Computation

Once the linear systems and solvers are chosen from the matrix collection and the numerical library (PETSc in this case) respectively, the next step involves computing various properties of these systems. For the first phase of this research, Anamod [32] was used to compute properties of the linear systems, referred to as the features. Anamod is a library of modules that use PETSc functions for computing 68 matrix features. These features include several categories, including simple (norm-like quantities), variance (heuristics estimating how different matrix

Feature names	
avgnnzprow	right-bandwidth
avgdistfromdiag	symmetry
n-dummy-rows	blocksize
max-nnzeros-per-row	diag-definite
lambda-max-by-magnitude-im	lambda-max-by-magnitude-re
ellipse-cy	nnzup
ruhe75-bound	avg-diag-dist
nnz	left-bandwidth
lambda-min-by-magnitude-im	lambda-min-by-magnitude-re
norm1	sigma-min
upband	n-struct-unsymm
colours	diagonal-average
diagonal-dominance	dummy-rows
ritz-values-r	symmetry-snorm
symmetry-fanorm	symmetry-fsnorm
lambda-max-by-real-part-im	lambda-max-by-real-part-re
lambda-max-by-im-part-re	lambda-max-by-im-part-im
col-variability	trace-abs
ritz-values-c	nnzeros
diag-zerostart	loband
positive-fraction	trace
min-nnzeros-per-row	diagonal-sign
row-variability	nrows
colour-offsets	n-colours
relsymm	diagonal-variance
departure	nnzlow
n-nonzero-diags	sigma-max
dummy-rows-kind	kappa
n-ritz-values	colour-set-sizes
sigma-diag-dist	symmetry-anorm
ellipse-ax	ellipse-ay
ellipse-cx	lee95-bound
normInf	normF
nnzdia	trace-asquared

Table 2. Full feature set comprising of 68 features computed using Anamod [32].

elements are), normality (estimates of the departure from normality), structure (nonzero structure properties), and spectrum (eigenvalue and singular value estimates produced using SLEPc). Figure 2 shows the complete list of all the features extracted using Anamod.

- Simple (norm-like) properties: This category includes properties that provide estimates of departure from normality. The computation time depends on the number of nonzeros of the matrix. All norms of the matrix belong to this category, for instance, 1-norm, infinity-norm, frobenius norm among others.

- Structural properties: This class of property involves properties that provide the nonzero structure of the matrix. Since these properties only describe the sparsity structure, they will most likely remain the same during a nonlinear solve. Some examples of these properties include the average number of nonzeros per row, the number of nonzeros in the diagonal and many more.

- Spectral properties: Spectral properties are the various estimates of the coefficient matrix spectrum, i.e. eigenvalues and singular values. These properties are believed to be very informative and also most expensive to compute as they may take up to hours or more for computation. Since they are very hard to compute, an estimation of these properties is mostly acceptable. Therefore, in this research, estimates of spectral properties are used.

- Variability properties: This class of properties includes measurements of matrix element variance. These properties describe how different are various elements in the matrix. Some examples of, variability properties are row variability, column variability and diagonal average.

- JPL properties: These properties describe the Jones-Plassman multi-colouring structure of a matrix. Examples of JPL properties include the number of colors computed, zero-based locations of the color sets in the colors array.

The next section, Section 3.10 describes the features that will be used in the reduced feature sets throughout the thesis.

3.7 Solving the linear systems

Table 3. PETSc Krylov iterative solvers and preconditioners.

Capability	**Algorithms**
Preconditioners	
Diagonal, block	Jacobi*, point block Jacobi, block Jacobi*, additive Schwarz*
Incomplete factorization	ILU*, ICC*
Matrix-free	infrastructure
Multigrid	infrastructure, geometric (DMDA for structured grid), geometric/algebraic, structured geometric, classical algebraic (BoomerAMG/hypre), classical algebraic (ML/Trilinos), unstructured geometric and smoothed aggregation
Physics-based splitting	relaxation and Schur-complement, least squares commutator
Substructuring	balancing Neumann-Neumann, BDDC
Krylov methods	Richardson, Chebyshev*, conjugate gradient*, GMRES*, transpose-free QMR*, TCQMR*, conjugate residual, conjugate gradient squared, bi-conjugate gradient (BiCG), BiCG-stab*, improved BiCG-stab*, MINRES, flexible GMRES*, LSQR*, SYMMLQ, LGMRES*, GCR, conjugate gradient on the normal equations

Once the features are computed, the next stage involves solving the linear systems with multiple solver-preconditioner combinations to generate the training and test datasets. In this work, the focus is on Krylov methods offered by PETSc.

We choose a subset of solvers and preconditioners available with PETSc, as mentioned in the previous chapter. The preconditioner-solver configurations used in this work are shown in Table 3, marked with an asterisk, with a unit right-hand-side vector. Each linear system is solved and its convergence time is captured, which is used to label the solver-preconditioner combination as "good" or "bad" based on the time.

The solver performance information was collected with PETSc version 3.5.3 on two supercomputers: Blue Gene/Q at Argonne National Laboratory and the ACISS cluster at the University of Oregon consisting of nodes containing two hex-core 2.66GHz Intel Westmere (X5650) processors and 72 GB of RAM. A single node was used for all the matrices as the University of Florida matrices are not very large. PETSc offers a collection of direct methods, iterative methods and preconditioner that can be used for application codes in C, C++, Python and Fortran. For the entirety of this thesis work, we use a set of the iterative methods, and preconditioners provided by PETSc. There are more than 300 valid options for solvers, preconditioners and their parameter configurations available in PETSc with all right-hand side elements set to one. We use a subset of these options and include 154 pairs of solvers and preconditioners. Below is a list of solvers and preconditioners used throughout this work. All these Krylov methods and preconditioners are described in more detail in Section II.

3.8 Solver classification

For classifying the solver-preconditioner pairs, supervised learning is used to train the machine learning model. For preparing the training dataset, each solver-preconditioner pair is assigned a binary/tertiary label ("good", "fair", "bad") for all the linear systems. For binary labeling, each datapoint $P_{i,j}$ is labeled as

"good" or "bad" or for tertiary labeling as "good", "fair" or "bad", based on the performance of the solver S_i, in terms of convergence time on matrix M_j. The solver-preconditioner with the least solve time was used as a threshold for the rest of the solver-preconditioner combinations that solved the system. If a combination failed to converge, it was timed out and labeled as "bad". For assigning the "good" label, a threshold parameter b was chosen in the range $0, 1$ based on how close a solver S_i's performance is to the known best-performing method. For instance, for binary labeling, when $b = 25$, solvers whose performance for a given problem is within 25% of the best timing were labeled as "good", while all other solvers were labeled as "bad". For tertiary labeling, a threshold parameter r in the range $b, 1$ was considered in addition to the parameter b for labeling solvers as "fair".

The rest of the section describes the PETSc Krylov methods and preconditioners used for the solver classification with the Suite Sparse [24] dataset.

3.8.1 PETSc Solvers. Using PETSc solvers is fairly easy compared to other linear algebra libraries. PETSc can be downloaded and installed in three ways: (1) by using GitHub (2) by installing the PETSc Debian package or (3) by using the PETSc web download link and instructions. Krylov methods have parameter configurations that can be varied. For this research, only default parameters were considered. After a successful installation for PETSc by any other three methods mentioned, the next step is to configure and build it. PETSc offers very easy to understand, step-by-step instructions via online documentation and tutorials. PETSc comes in with its prerequisites with automatic download, configure, build and installation with no additional work for the user. There are many PETSc examples available online along with the command line options to be

used to enable using different solvers offered by PETSc. Below is a brief description of the subset of solvers used in the research presented in this chapter.

1. **Generalized minimal residual method (GMRES)**: This method approximates the solution by the vector in a Krylov subspace with minimal residual. The Arnoldi iteration is used to find this vector.
2. **The Flexible Generalized minimal residual method (FGMRES)**: It is a generalization of GMRES that allows larger flexibility in the choice of solution subspace than GMRES.
3. **LGMRES**: It augments the standard GMRES approximation space with approximations to the error from previous restart cycles.
4. **Conjugate gradient method (CG)**: This method starts with an initial guess of the solution, with an initial residual and with an initial search direction.
5. **Biconjugate Gradient Method (BICG)**: Implements the Biconjugate gradient method, similar to running the conjugate gradient on the normal equations.
6. **Biconjugate gradient stabilized method (BCGStab)**: It is a stabilized version of BiConjugate Gradient Squared method.
7. **Improved Stabilized version of BiConjugate Gradient Squared (IBCGS)**: It is an improved stabilized version of BiConjugate Gradient Squared method.

8. **Transpose-Free Quasi-Minimal Residual Method (TFQMR)**: It is a quasi-minimal residual version of CGS. It retains the desirable convergence features of CGS and corrects its erratic behavior.

9. **TCQMR**: It is a variant of quasi-minimal residual provided by Tony Chan.

10. **LSQR**: This is an algorithm for sparse linear equations and sparse least squares.

11. **Chebyshev**: This method requires enough knowledge about the spectrum of the matrix, which is an upper estimate for the upper eigenvalue and lower estimate for the lower eigenvalue. Chebyshev iteration method avoids the computation of inner products as is necessary for the other methods.

3.8.2 PETSc Preconditioners. Preconditioning refers to the process of applying a transformation on the original problem and brings it into a form that is more suitable for the solving methods. The main idea behind applying a preconditioner is that, instead of solving $Ax = b$, solve $M^{-1}Ax = M^{-1}b$ using a nonsingular $m \times m$ preconditioner M, which has the same solution x.

Similar to using the PETSc solvers, different preconditioners can be used combined with the PETSc solvers using command line options. The PETSc documentation and tutorials also describe the different parameters that can be configured for the preconditioners. This part of the section lists the various preconditioners that were considered and the subset of parameters used for them. The preconditioners are as follows:

1. **Incomplete factorization preconditioners (ILU)**: ILU is an approximation of the LU (Lower Upper) factorization. LU factorization factors a matrix as the product of the lower and the upper triangular matrix.

Parameters: Factor levels which are the number of levels of fill for ILU.

Parameter values: 0, 1, 2 and 3.

2. **Additive Schwarz method (ASM)**: Solves an equation approximately by splitting it into boundary value problems and adds the results.

   Parameter considered: The amount of overlap between sub-domains.

   Parameter values: 0, 1, 2 and 3.

3. **Jacobi or diagonal**: One of the simplest forms of preconditioning in which the preconditioner is the diagonal of the matrix as shown below.

   $M = diag(A)$ for $M^{-1}Ax = M^{-1}$ b.

4. **Block Jacobi**: It is similar to Jacobi, except that in this case, instead of the diagonal, the block-diagonal is chosen as the preconditioner (M).

5. **Incomplete Cholesky factorization (ICC)**: It is a sparse approximation of the Cholesky factorization. The Cholesky factorization A is $A = LL^*$ where L is a lower triangular matrix. An incomplete Cholesky factorization is given by a sparse lower triangular matrix K that is very close to L. The corresponding preconditioner is KK^*. **Parameter considered**: Factor levels which are the number of levels of fill for ICC. **Parameter values**: 0, 1, 2 and 3.

3.9 Cost Reduction

One of the goals when building a classification model is to reduce the overall cost of solving the system. Cost reduction can be achieved in two stages: (1) at the time of training the model and (2) while making predictions. If the number of features to be used for building the model can be reduced, substantial cost

reduction can be accomplished. Feature reduction is performed by selecting features that are significant and contribute the most towards the classification process. Random selection of features would not be suitable as there may be significant features that get removed by this selection process.

The computation time of each feature varies depending on the category it belongs to as some features are more expensive to compute than the others. The reduced feature set contains features that are cheaper to compute than some of the other properties in the full feature set. In other words, the top significant features to be chosen is a good strategy.

Selecting the significant features was achieved in two ways. First, the feature set is reduced to eliminate features that do not contribute significantly to the classification process. Feature elimination involves removing those features which have either more than 99% or nearly 0% variance. Second, we apply multiple attribute evaluators with different search methods. The evaluator applies a strategy to assign a weight to each feature. The search method determines the search technique would be performed. The evaluators rank the features, which helps in identifying the features that do not contribute much to the classification.

Feature elimination and relevant feature selection generate a subset of the full feature set, which includes only spectral and structural properties. The dataset comprises these features combined with the solver-preconditioner pair ids and the class label. Each solver-preconditioner pair is assigned a unique id, which is one of the attributes of the data set. The class label for the training set is assigned based on the computation time of the Krylov methods. Each linear system is solved with multiple Krylov solvers and preconditioners. The best performing solver (the fastest solver) timing serves as the threshold for the other solvers that solve the same

linear system. The solvers with solve time within the threshold value are labeled as “good” and all others are labeled as “bad”. This combination of attributes serves as the input for the ML model.

3.10 Feature Set

The full feature set comprises 68 features that are computed using Anamod. These features belong to the five categories as described in Section 3.6. This section presents the candidates for the various reduced feature sets that are used throughout the thesis. Table 4 shows the list of features in the reduced sets used for the work presented in this chapter. Below is a brief description of the features that are used as reduced features in the research presented in the rest of the thesis.

- **Dimension**: This represents the number of rows and columns. For any square matrix, the number of rows and number of columns is equal.
- **Nonzeros**: This value represents the total number of non-zeros in the matrix.
- **Maximum, minimum, and average nonzeros per row**: These features represent the row-based nonzero value statistics, i.e. maximum, minimum and average nonzeros per row.
- **Dummy rows**: This feature counts the number of rows that have only one nonzero element.
- **Dummy rows kind**: There are three possible outputs for the feature, which are based on the value of the dummy row elements. Either every dummy row of the matrix contains a 1 along the diagonal of the matrix or every dummy row has a nonzero entry or at least one dummy rowâĂŹs entry is on a non-diagonal position.

- **Absolute non-zero sum**: Sum of the absolute values of all the nonzero elements.

- **Numeric Value SymmetryV1**: This checks for the numerical symmetry of the matrix. If the matrix is symmetric then the property of this value is one, else zero. The symmetry of a matrix can be checked by $A = A^T$ where A^T is the transpose of the matrix A.

- **Non-Zero Pattern SymmetryV1**: This property checks the nonzero pattern symmetry of the matrix, if it is symmetric then the value of this property is 1 else 0. The matrix can be checked for being symmetric by finding out if A and A^T have the same nonzero pattern. In other words, if for every nonzero entry $a_{i,j}$ of A, A^T has a nonzero entry $a_{i,j}$, then the value of this property is 1.

- **Numeric Value SymmetryV2**: This property checks the numerical symmetry of the matrix. The value of this property is a percentage, computed by $v = 1-(\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}|s_{i,j}|/\sum_{i=1}^{m}\sum_{j=1}^{m}|a_{i,j}|)$, where $S = (s_{i,j})$ is $\frac{1}{2}(A-A^T)$, the antisymmetric part of A.

- **Nonzero Pattern Symmetry V2**: The nonzero pattern symmetry of the matrix, which can be given as the ratio between nonzeros $a_{i,j}$ in A for which no entry $a_{j,i}$ in A exists and the total number of nonzeros in A.

- **Trace**: This feature computes the sum of the diagonal elements of the matrix. Mathematically it can be represented as follows:

$$\frac{c}{m}\sum_{\mathbf{a}_i \in S_s}(\mathbf{a}_i)_i$$

- **Absolute Trace**: The sum of the absolute values of all the diagonal entries of the matrix.
- **One Norm**: The feature provides the maximum absolute column sum of the matrix, which can be shown as follows: $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{m} a_{i,j}^2}$.
- **Infinity Norm**: Infinity norm is the maximum absolute row sum of the matrix. It can be given as follows: $\max\limits_{1\leq i\leq m} (\sum_{j=1}^{m} |a_{i,j}|)$
- **Frobenius Norm**: Frobenius norm is computed as the square root of the sum of all elements squared, which can be written as: $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{m} a_{i,j}^2}$.
- **Symmetric Infinity Norm**: The infinity norm of the symmetric part of the matrix is computed as follows:

$$||A||_\infty \approx \frac{c}{m} \max_{i\in[0,r]} \left[\sum_{j\in[0,m]} |(\mathbf{a}_j)_i)| \right]$$

- **Symmetric Frobenius Norm**: The Frobenius norm of the symmetric part of the matrix.
- **Anti-Symmetric Infinity Norm**: The infinity norm of the antisymmetric part of the matrix.
- **Anti-Symmetric Frobenius Norm**: The Frobenius norm of the antisymmetric part of the matrix.
- **Row Diagonal Dominance**: For any row of the matrix, if the absolute value of the diagonal entry in a row is smaller than the sum of the absolute values of the non-diagonal entries then the value for this property is 0. That is, $|a_{i,i}| < \sum_{j\neq i} |a_{i,j}|$ for all j. The value for this property is 1, if $|a_{i,i}| =$

$\sum_{j \neq i} |a_{i,j}|$ for all j. The value for this property is 2, if $|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$ for all j.

- **Column Diagonal Dominance**: For any column of the matrix, if the absolute value of the diagonal entry in a column is smaller than the sum of the absolute values of the non-diagonal entries then the value for this property is 0. That is, $|a_{j,j}| < \sum_{i \neq j} |a_{i,j}|$ for all i. The value for this property is 1 if $|a_{j,j}| = \sum_{i \neq j} |a_{i,j}|$ for all i. The value for this property is 2 if $|a_{j,j}| > \sum_{i \neq j} |a_{i,j}|$ for all i.

- **Row Variability**: The row variance of the matrix can be given as the maximum ratio between a rowâĂŹs minimum and maximum entries. Row variance of any row, say, i is computed by $\frac{1}{m} \sum_{j=1}^{m} (a_{i,j} - \mu)^2$, where $\mu = \frac{1}{m} \sum_{j=1}^{m} a_{i,j}$.

$$\text{Row variability} : max_i \log \frac{max_j |a_{ij}|}{min_j |a_{ij}|} \tag{3.2}$$

- **Column Variability**: The maximum column variance of the matrix can be given as the maximum ratio between a columnâĂŹs minimum and maximum entries. Column variance of any column, say, j is computed by j is $\frac{1}{m} \sum_{i=1}^{m} (a_{i,j} - \mu)^2$, where $\mu = \frac{1}{m} \sum_{i=1}^{m} a_{i,j}$. Column variability are computed as follows:

$$\text{Column variability} : max_j \log \frac{max_i |a_{ij}|}{min_i |a_{ij}|} \tag{3.3}$$

- **Diagonal Average**: The arithmetic mean of the absolute values of the diagonal entries of the matrix. It cane be computed by $\frac{1}{m} \sum_{i=1}^{m} |a_{i,i}|$.

- **Diagonal Variance**: The variance of the diagonal elements of the matrix, which can be computed by $\frac{1}{m} \sum_{i=j=1}^{m} (a_{i,j} - \mu)^2$, where $\mu = \frac{1}{m} \sum_{i=1}^{m} a_{i,i}$.

- **Diagonal Sign**: It is used to represent the pattern of the diagonal sign. There are six possible values for this feature:
  * 3, if some of the diagonal elements of the matrix are negative and some are positive and some or none of them are zeros.
  * 2, if all the diagonal elements of the matrix are positive.
  * 1, if all the diagonal elements of the matrix are either positive or zeros.
  * 0, if all the diagonal elements of the matrix are zeros.
  * -1, if all the diagonal elements of the matrix are either negative or zeros.
  * -2, if all the diagonal elements of the matrix are negative.
- **Diagonal Nonzeros**: The number of non-zero elements present in the diagonal. The number of diagonal non-zeros is estimated as

$$dnz(A) = \frac{c}{m} \sum_{a_i \in S_s} \delta((\mathbf{a}_i)_i)$$

- **Lower Bandwidth**: The smallest number k where $a_{i,j} = 0$ when $j < i + k$ and $k > 0$.
- **Upper Bandwidth**: The smallest number k where $a_{i,j} = 0$ when $j > i + k$ and $k > 0$.
- **Row Log Value Spread**: This feature represents the spread of log values in rows: $\max_i \log_{10} \frac{\max_j |a_{ij}|}{\min_j |a_{ij}|}$
- **Column Log Value Spread**: The feature computes the spread of log values in columns: $\max_j \log_{10} \frac{\max_i |a_{ij}|}{\min_i |a_{ij}|}$
- **Symmetry**: The feature holds a boolean value; if the matrix is symmetric or Hermitian, the value is true (1) and if it is non-symmetric, the value is false.

We use the full feature set and the reduced feature sets to compare the accuracy achieved by the ML models. We then select the algorithm that performs the best and classifies solver performance most effectively. The last stages involve testing the classifiers on the test data set and observe the performance. There on, we use the best performing classifier to make predictions for the good-performing preconditioner-solver pairs for incoming problems arising in different domains. We examine Bayes Net [14], k-nearest neighbor [22], Alternate Decision Trees [44], multiclass extension of Alternating Decision Trees [63], Random Forests [17], J48 [68] and Support Vector Machines [21]. We illustrate this approach on the Conjugate Gradient [83], GMRES [76], Flexible GMRES [74], TFQMR [43], BiCG [85], iBCGS [59] and BCGS [98] solvers with ASM [18], Jacobi [92] and Block Jacobi [84] preconditioners.

3.11 Performance Evaluation

To test the performance of the machine learning techniques described in Section , we use the "good" solver accuracy and the overall accuracy of the ML technique. The confusion matrix provides statistics for how many "good" solver were classified as "good" and how many "bad" solvers were described as "bad". These are referred to as the sensitivity and specificity of the models. Sensitivity is computed as follows:

$$TPR = TP/P = TP/(TP + FN), \tag{3.4}$$

where P is the actual number of "good" instances, TP are the number of "good" solvers correctly labeled as "good". FN denotes the number of "good" solvers that were misclassified as "bad" by the classifier. Specificity of a model is given as:

$$TPR = TN/N = TN/(TN + FP) \tag{3.5}$$

Here, N is the actual number of "bad" instances, TN are the number of "bad" instances correctly labeled as "bad". FN denotes the number of "bad" instances predicted as "good" by the classifier. The focus is on the "good" solver accuracy throughout this dissertation as which solvers performed "bad" is likely to be unexciting for the users. For validation, we use the following two techniques for the full-feature dataset and the reduced sets:

- Train-test split: While training a model, the goal is to feed the data to that model so that it can recognize patterns and use the information to make predictions for new data points. One challenge that may arise during training a model is overfitting. Overfitting arises when the model learns the training data and has 100% accuracy on the training data. Such models perform poorly on data other than the training data. To avoid overfitting, the data is usually split into two sets: train and test. Training data is used to train the model and test set is used to validate the model. A popular train-test split ratio is $66 - 34\%$, where two-thirds of the data is used for training and the rest is used for validating the model.

- N-fold cross-validation: N-fold cross-validation technique evaluates the model by splitting the data randomly into N sized subsamples. All the subsamples except 1, $N - 1$ subsamples are used to train the model and one of the N subsamples is used for testing the model. The subsampling process is carried out N times, with different subsamples every time. The advantage of this technique over the train-test split is that it uses 100% of the data for training and testing and is extremely useful especially in scenarios where the dataset has small (<10,000) number of datapoints.

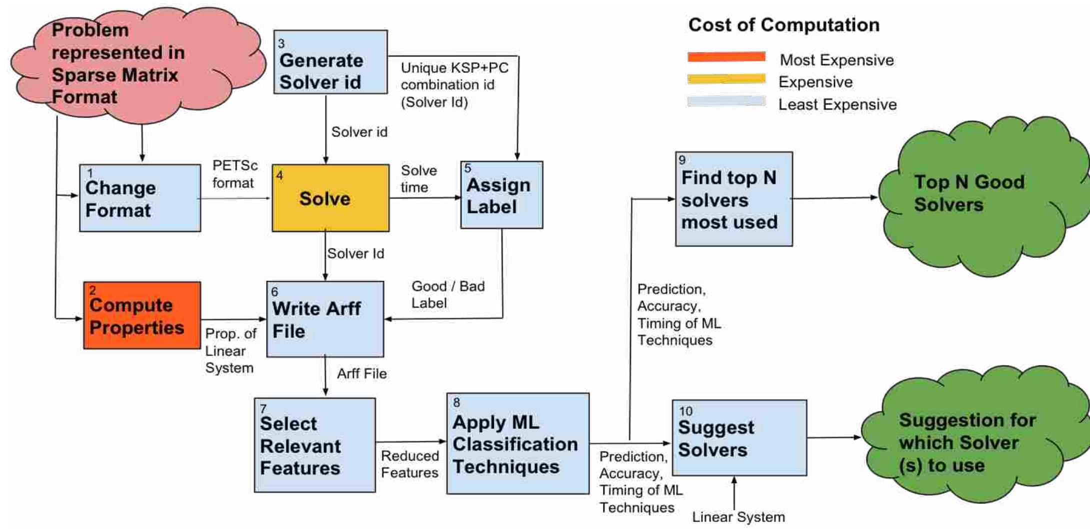


Figure 10. The overall workflow of the linear system solver selection process with the convergence (ML) model.

3.12 Experimental Results

The performance of 154 solver-preconditioner pairs is evaluated with PETSc 3.5.3 on the Blue Gene/Q supercomputer for more than 1,000 matrices from the SuiteSparse collection resulting in a total of 4,648 data-points consisting of matrix features computed using Anamod. The data-size is particularly interesting because usually, a machine learning modeling technique requires at least a sufficiently large dataset. With less than 5,000 data-points the convergence modeling produced the best accuracy of 87.6% with the full feature set that includes 68 input features from BayesNet classifier. Based on the feature reduction, as discussed in Section 3.9, we reduce the number of feature to a set, RS1, of 8 features. For the reduced set, RS1, which comprises eight features alone, an accuracy of 86.9% is achieved with BayesNet classifier. With RS2, that consists of only six features achieves an accuracy of 86.5% with the BayesNet classifier. Table 4 shows the list of features that comprise the reduced feature sets RS1 and RS2. Figure 12 shows

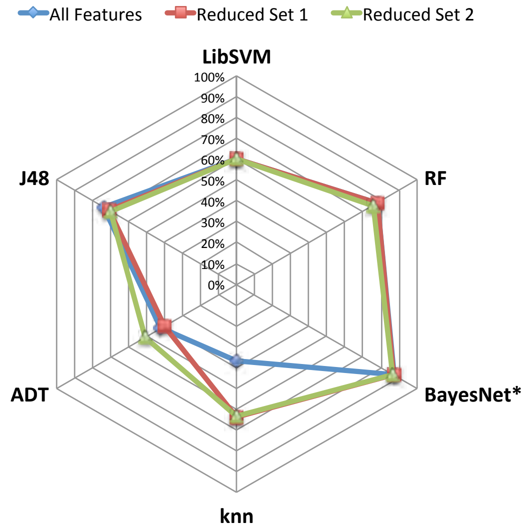


Figure 11. Machine learning classifier accuracy comparison for full feature set and reduced feature sets.

Method	T_{All}	T_{RS1}	T_{RS2}
LibSVM	6.88	1.94	1.79
RF	2.67	2.40	1.43
BayesNet	0.10	0.02	0.01
knn	0.001	0.001	0.001
ADTree	0.74	0.18	0.09
J48	0.24	0.19	0.06

Figure 12. Time taken (seconds) to construct the classifier for machine learning classification.

Table 4. Reduced feature sets (RS1 and RS2) used for classification

Feature name	Reduced Feature Set 1 (RS1)	Reduced Feature Set 2 (RS2)
avg-diag-dist	X	X
nnz	X	
norm1	X	X
col-variability	X	X
min-nnzeros-per-row	X	X
row-variability	X	X
n-nonzero-diags	X	
kappa	X	X

the classification accuracy for all the machine learning classifiers used in this work. Figure 11 shows the time taken to build each of the classifiers. A classifier with the highest accuracy and least build time is preferable. Classifier accuracy is of maximal importance in this work because the ultimate goal of this research is to make solver recommendations that are quasi-optimal. Although the cost of building the machine learning classifier is a one-time cost, classifiers with comparatively large build time were not the preferred choice for classification.

3.13 Summary

To summarize, the entire workflow of the ML model is shown in Figure 10. The first stage in the workflow involves obtaining the linear system in the right format. SuiteSparse matrix collection offers matrix market format. For convenience, the matrices are converted into PETSc format. The next stage involves extracting the features of these linear systems. A subset of PETSc solvers and preconditioners are used to solve these systems. Before performing the linear solve, a unique solver identifier is assigned to each pair of specific Krylov method and preconditioner. The next stage involves solving each of these linear systems with multiple solver-preconditioner pairs and capturing the solve time. Based on the solver time, the solver-preconditioner pairs are categorized as “good” or “bad”.

The next step is to build the data for the classification model. For the dataset, each data-point is composed of the matrix properties, unique solver id and the class label ("good" or "bad") for each linear system. Since multiple solver-preconditioner pairs solve the same linear system, only the unique solver id changes whereas the matrix properties are the same for all these data points. The class label may be "good" or "bad" depending on the time taken by the solver-preconditioner pair to solve the system. All such data points together constitute the dataset for supervised machine learning. The dataset is written in Comma-Separated Values (CSV) and Attribute-Relation File Format (Arff). The next steps involve applying various machine learning classification techniques on the full feature set and the reduced feature set. The output of this scheme is (1) Top 10 most used solvers for the entire dataset and (2) solver-preconditioner combination suggestions, in the form of an unranked list, for an unknown linear system.

The second most expensive step in the workflow is the actual solve of the linear systems. Since we need to solve the systems only once for the model training, it can be considered as a one-time cost. The most expensive step in the workflow is the property computation step because some properties are more expensive computationally than others. For instance, the spectral properties like eigenvalues are much more expensive than structural properties like the number of rows or the maximum number of nonzeros per row. To reduce the overall cost of the system, we reduce the number of features to be computed for an incoming system to less than eight features. These features mostly include structural and size-based features and therefore are comparatively cheap to compute.

CHAPTER IV

COMMUNICATION MODEL FOR SPARSE LINEAR SOLVER SELECTION

This chapter is based on the collaboration [87] between Boyana Norris (UO), Elizabeth Jessup (CU Boulder), and myself. Boyana Norris and I designed the analytical model for ranking a subset of the parallel solvers offered by PETSc. I studied the Krylov method evaluations and analyzed the inter-process communication to compare the scalability of various parallel solvers. I generated the communication-based ranking, which was meticulously verified by Boyana Norris. I performed the case by case comparison of communication costs of matrix-vector operations and Boyana Norris provided guidance on conducting the comparison in an efficient manner. Throughout this research, Elizabeth Jessup verified the various matrix-vector operation costs, our understanding of the Krylov method behaviors and provided significant feedback.

This chapter presents our communication-based performance model for comparing the scalability of several parallel preconditioned Krylov methods from PETSc without extensive empirical measurements. We generate a scalability ranking for the preconditioned solvers based on an analytical communication model. The communication model provides an extension to the convergence model presented in chapter III, to handle large scale problems and recommend solvers at different parallelism scales. The next chapter presents how the model is validated by evaluating it on a numerical simulation of driven fluid flow in a two-dimensional cavity. With the association of the convergence model with the communication model, the approach enables capturing both the convergence behavior and the parallel overhead of the Krylov methods.

4.1 Motivation

Iterative solvers are very popular for providing sparse system solutions on parallel architectures for two reasons. First, iterative solver techniques are expected to scale well as they use factorization of the coefficient matrix into invertible matrices. Second, often an exact solution is not required, for instance, the methods that solve a system of equations in which there is a new system to be solved at each iteration. In such cases, an approximation of the solution is sufficient. For solving large sparse linear systems, iterative solvers are usually paired with preconditioners to improve their robustness. There are many open-source numerical packages for the iterative solution of sparse linear systems that are derived from partial differential equation problems such as PETSc and Trilinos. Given the number of pre-existing numerical libraries and further addition of new solution methods, selecting an "well-performing" solution technique is very challenging. Therefore we propose the communication-based technique to enable solver recommendations at different parallelism scales.

4.2 Communication Cost of Preconditioned Krylov Methods

For any given solver technique, there are two types of costs affiliated: communication cost and computation cost. The computation cost is the cost of the iterative solution of a linear algebraic system $Ax = b$. The computation cost is associated with the convergence time and memory requirements. Communication cost in iterative solvers is given by the number of arithmetic operations for the individual steps in solving the system until the computation is stopped. These arithmetic operations include various operations such as global reductions, matrix-vector operations, and scatter-gather operations. For small-scale problems, the convergence model described in Chapter III is preferable for many reasons: (1) it

Table 5. Number of calls of communication-relevant functions per iteration of various Krylov methods.

Operations	Parameter	Count
Conjugate Gradient		
VecTDot	q	2
VecNorm	q	2
PCApply	c_{pc}	2
MatMult	m	2
GMRES		
VecMDot_MPI	$q * r(r+1)/2$	1
VecNorm	q	2
PCApply	c_{pc}	2
MatMult	m	2
Flexible GMRES		
VecMDot_MPI	$q * r(r+1)/2$	1
VecNorm	q	2
PCApply	c_{pc}	1
MatMult	m	2
BCGS		
VecDot	q	2
VecNorm	q	2
VecDotNorm2	q	1
PCApply	c_{pc}	3
MatMult	m	3
iBCGS		
VecDot	q	2
VecNorm	q	1
MatMultTranspose	m	1
PCApply	c_{pc}	5
MatMult	m	3
MPIU_AllReduce	w	2
TFQMR		
VecDot	q	3
VecNorm	q	2
PCApply	c_{pc}	4
MatMult	m	4
BiCG		
VecDot	q	2
VecNorm	q	2
PCApply	c_{pc}	3
PCApplyTranspose	c_{pc}	2
MatMult	m	2
MatMultTranspose	m	1

captures the convergence behavior of Krylov methods successfully (2) it involves only a handful of features to be computed for any system (3) the model building and training cost are affordable.

To compare the performance of various Krylov methods with preconditioning, this work presents an analytical communication-based model which generates a scalability ranking for the Krylov methods. A subset of Krylov methods and preconditioners were considered, with seven parallel Krylov methods and seven preconditioners from PETSc. The subset includes a non-preconditioned case for all the Krylov methods making a total of 49 cases. Each of these solver and preconditioner implementations provided in PETSc is thoroughly analyzed for the communication model. For computing the scalability ranking, we compare the communication occurring in different Krylov method implementations.

Table 6. Matrix-vector operations with communication

Operation	Description	Cost Variable
MatMult	Computes matrix-vector product: $y = Ax$	m
MatMultTranspose	Computes matrix transpose times a vector $y = A'x$	m
VecNorm	Computes norm of the vector: $r = \|\|x\|\|$	q
VecDot	Computes the dot product of the vectors x and y	q
VecMDot	Computes one or more vector dot products.	q
VecMDot_MPI	Computes vector multiple dot products and performs reductions	$q * k(k+1)/2$
VecTDot	Computes indefinite vector dot product: $y^H x$, where y^H denotes the conjugate transpose of vector y	q
VecDotNorm2	Computes the inner product of two vectors and the 2-norm squared of the second vector	q
PCApply	Performs the preconditioning on the vector	c_{pc}
PCApplyTranspose	Applies the transpose of preconditioner to a vector	c_{pc}
VecScatterBegin	Performs a scatter from one vector to another	v
MPIU_Allreduce	Determines if the call from all the MPI processes occur from the same location in the code.	w

4.3 Building the Analytical Communication Model

For analyzing each Krylov method implementation, primarily the inter-process communication is evaluated by identifying the communication-causing operations, cost of these operations and the number of times these operations have been called per iteration. The analysis is done for Krylov solver iteration alone excluding the initial setup function, I/O functions and the common operations for all the implementations. Since the total number of iterations for each solver vary,

we normalize calls per iteration so that the raw counts for the number of times operations have been called do not lead towards a bias for solver techniques with less operation count and more iterations.

4.3.1 Identify operations with communication.

While analyzing the Krylov method implementations, all matrix-vector operations were inspected for communication. Figure 13 shows the matrix and vector operations in Krylov methods offered by PETSc. Table 6 shows all the operations that have communication. Throughout the analysis of the solver implementations, the goal is to identify the communication happening within each iteration. Here, the focus is on high-level communication primitives and excludes low-level communication functions. The computation aspect is not included in the analytical modeling as on a large scale the communication cost is expected to be more dominant. The next chapter of the dissertation describes how the computation and communication aspect can be captured together with our approach of using the convergence model in combination with this analytical model. The operations of interest include global reduction operations such as the vector norms, one or more vector dot products, or the matrix multiplication and its transpose, or the matrix-vector products, nearest-neighbor scatter and gather operations.

4.3.2 Matrix-vector product.

There are two operations namely $MatMult$ and $MatMultTranspose$ that perform matrix-vector product. Let the cost of each such product be represented by variable m and the number of nonzeros per row for each processor be n and the number of processors be p. Each processor sends its nonzeros per row values to all other $p-1$ processors and receives partial sum contributions to its vector elements. The amount of communication per processor includes sending n values to $p - 1$ processors and receiving n values

Function Name	Operation
MatAXPY(Mat Y, PetscScalar a,Mat X,MatStructure);	$Y = Y + a * X$
MatMult(Mat A,Vec x, Vec y);	$y = A * x$
MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
MatMultTranspose(Mat A,Vec x, Vec y);	$y = A^T * x$
MatMultTransposeAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
MatNorm(Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
MatDiagonalScale(Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
MatScale(Mat A,PetscScalar a);	$A = a * A$
MatConvert(Mat A,MatType type,Mat *B);	$B = A$
MatCopy(Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal(Mat A,Vec x);	$x = \text{diag}(A)$
MatTranspose(Mat A,MatReuse,Mat* B);	$B = A^T$
MatZeroEntries(Mat A);	$A = 0$
MatShift(Mat Y,PetscScalar a);	$Y = Y + a * I$

Function Name	Operation
VecAXPY(Vec y,PetscScalar a,Vec x);	$y = y + a * x$
VecAYPX(Vec y,PetscScalar a,Vec x);	$y = x + a * y$
VecWAXPY(Vec w,PetscScalar a,Vec x,Vec y);	$w = a * x + y$
VecAXPBY(Vec y,PetscScalar a,PetscScalar b,Vec x);	$y = a * x + b * y$
VecScale(Vec x, PetscScalar a);	$x = a * x$
VecDot(Vec x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
VecTDot(Vec x, Vec y, PetscScalar *r);	$r = x' * y$
VecNorm(Vec x,NormType type, PetscReal *r);	$r = \|\|x\|\|_{type}$
VecSum(Vec x, PetscScalar *r);	$r = \sum x_i$
VecCopy(Vec x, Vec y);	$y = x$
VecSwap(Vec x, Vec y);	$y = x$ while $x = y$
VecPointwiseMult(Vec w,Vec x,Vec y);	$w_i = x_i * y_i$
VecPointwiseDivide(Vec w,Vec x,Vec y);	$w_i = x_i / y_i$
VecMDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
VecMTDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);	$y = y + \sum_i a_i * x[i]$
VecMax(Vec x, int *idx, PetscReal *r);	$r = \max x_i$
VecMin(Vec x, int *idx, PetscReal *r);	$r = \min x_i$
VecAbs(Vec x);	$x_i = \|x_i\|$
VecReciprocal(Vec x);	$x_i = 1/x_i$
VecShift(Vec x,PetscScalar s);	$x_i = s + x_i$
VecSet(Vec x,PetscScalar alpha);	$x_i = \alpha$

Figure 13. Matrix and Vector operations in PETSc.

from all other processors. Therefore the communication cost per processor for each matrix-vector operation can be written as $m = 2 * n * (p - 1)$.

Scatter gather vector operations: There is one operation that performs a scatter from one vector to another, namely $VecScatterBegin$. Let the cost of the nearest neighbor scatter gather vector operation be represented by variable v. Tau Performance system toolkit [82] was used to trace the parallel Krylov method implementations offered by PETSc. The cost of this operation is constant involving small amounts of data ($\leq$ 32 bytes or 4 double-precision scalar values) and does not depend on the number of processors or the size of the problem (number of nonzeros).

4.3.3 Reduction operations.

The operations that reduce an array of values into a single scalar value are known as the reduction operations. There are seven operations in this category, namely $VecDot$, $VecTDot$, $VecMDot$, $VecMDot_MPI$, $VecDotNorm2$, $VecNorm$, and $MPIU_Allreduce$. Let the cost of a reduction operation be denoted by the variable q. The dot product of two vectors, $VecDot$ or the transpose of a vector with another vector, $VecTDot$ or a combination of norm/dot product, $VecDotNorm2$ all use at least one global reduction operation. The communication cost of the reduction operation can be given by $\log p$ [89]. The operation $VecMDot$ performs one or more dot products and $VecMDot_MPI$ performs reductions in addition to the dot products. $VecMDot_MPI$ exists only in GMRES, FGMRES implementations, with k as the restart parameter value. The cost of dot products as mentioned before, is $\log p$ and for reducing x data items, where x ranges from $1, 2, \ldots k$ and the sum of the cost of all the x items can be computed by the sum of the series $1 + 2 + 3 + \cdots + k$ formula: $k(k + 1)/2$. Now, the total communication cost is $qk(k + 1)/2$. The last

operation, $MPIU_Allreduce$ combines the values from all the processors and sends back the result to all of the processors. The cost of this operations, let's denote it by w, involves a constant amount of data as it does not depend on the processor count of the number of nonzeros. The cost can be given as fixed data size: <96 bytes or 12 precision values.

4.3.4 Application of Preconditioners. There are two operations, *PCApply* and *PCApplyTranspose*, which perform the preconditioning on the vector and the transpose of a preconditioner on the vector respectively. The cost of preconditioner application is denoted by c_{pc}, where the subscript represents which preconditioner was applied, i.e. Jacobi, Block Jacobi or ASM(0) / ASM(1) / ASM(2). For ASM, the integer value in the bracket represents the overlap between a pair of subdomains for the preconditioner. In the case of no preconditioning, the cost of this operation is naturally zero.

4.3.5 Assign cost to operations with communication. Based on the description, here we describe the cost assigned to all the communication-causing operations in terms of the number of processors p, processor's average number of nonzeros per row n, restart parameter for GMRES and FGMRES k. The cost for the matrix-vector multiplication is denoted by the variable m and can be given as $2 * n * (p - 1)$. The cost of the reduction operation is denoted by q and can be given as $\log p$. The cost for the $MPIU_Allreduce$ which is represented by w, can be assigned a constant cost of $\leq$ 96 bytes or 12 double precision values. The cost for the $VecScatterBegin$ operation can be denoted by v and given as $\leq$ 32 bytes or 4 double precision values. The cost of $VecMDot_MPI$ operation is equal to $q*k(k+1)/2$. To estimate the communication cost for different Krylov methods, the communication cost for a single iteration in all implementations. The computation

cost is excluded from this analytical modeling because it can be captured with the convergence model described in the previous chapter. Table 5 shows the number of calls made to the operations with communication for each Krylov method that is analyzed.

4.4 Compare Communication Cost across Krylov Method Implementations

There are 49 combinations of parallel preconditioned Krylov methods analyzed in this work. The goal is to identify the differences in the communication cost in a single iteration by considering normalized calls per iteration. The cost of communication for each Krylov method is described with respect to the number of calls to underlying communication intensive kernels and the cost of the matrix-vector operations. Each pair of solver-preconditioner is compared with every other pair by computing the difference in their communication costs. Let us consider a pair of preconditioned Krylov methods, say S_a, S_b. There are three possible values: if the difference in the communication cost of S_a and S_b is (1) greater than zero then the communication cost of S_a is more than the communication cost of S_b (2) less than zero, then the communication cost of S_b is more than the communication cost of S_a and (3) equal to zero, then the communication cost of S_b and S_a is the same. These pairs of Krylov methods may involve different operations due to which an exact comparison of the amount of communication in each of the 49 cases is not necessary. In a few cases, basic matrix-vector operations are considered and in some cases, simply identifying which Krylov method pair has more communication is enough to perform the communication-cost comparison. Rest of the sub-section presents a case by case comparison of communication costs for various preconditioned Krylov methods.

4.4.1 GMRES and Conjugate Gradient:.

Let the communication cost of Conjugate Gradient be represented as C_{CG} and cost of GMRES be C_{GMRES}. Conjugate Gradient involves two matrix-vector products ($2m$), two vector norm computations ($2q$) and two vector dot products ($2q$). Now, the total communication cost is $2m + 2q + 2q$, i.e. $C_{CG} = 2m + 4q$. GMRES computes two vector norm computations $2q$, and two matrix-vector products ($2m$) and one vector multiple dot product $q * k(k+1)/2$, so the total communication cost is $C_{GMRES} = 2m + 2q + (q * k(k+1)/2)$. The difference in the costs for GMRES and Conjugate Gradient can be given as follows:

$$C_{GMRES} - C_{CG} = 2m + 2q + (q * k(k+1)/2) - (2m + 4q) \tag{4.1}$$

The above equation on simplification becomes:

$$C_{GMRES} - C_{CG} = (q * k(k+1)/2) - 2q = (k(k+1)/2 - 4)q > 0 \text{ for all } k > 2 \tag{4.2}$$

Therefore, the communication cost of GMRES(k) is more than the communication cost of Conjugate Gradient for all $k > 2$.

4.4.2 Flexible GMRES (FGMRES) and Conjugate Gradient:.

FGMRES includes two vector norm computations ($2q$), two matrix-vector products ($2m$) and one VecMDot_MPI operation. As shown in the previous case, $C_{CG} = 4q + 2m$. The communication cost of FGMRES and Conjugate Gradient can be shown as:

$$C_{FGMRES} - C_{CG} = 2m + 2q + q * k(k+1)/2 - (2m + 4q) \tag{4.3}$$

This can be further simplified as:

$$C_{FGMRES} - C_{CG} = (k(k+1)/2 - 4)q > 0 \text{ for all } k > 2. \tag{4.4}$$

Therefore the communication cost for FGMRES is more than Conjugate Gradients for all value of $k > 2$. The communication cost of FGMRES shown in

this case and the cost for GMRES, as shown in the previous case, are the same. Thus the comparison of Conjugate Gradient with FGMRES method is the same as that of GMRES. Therefore the communication cost of FGMRES is the same as GMRES and more than the communication cost for Conjugate Gradient.

4.4.3 TQFMR and Conjugate Gradient:. TFQMR has two vector norm computations ($2q$), three vector dot products ($3q$), four matrix-vector products ($4m$). The difference $C_{TFQMR} - C_{CG} = 5q + 4m - (4q + 2m) = q + 2m > 0$ for all $q, m > 0$. Therefore, TFQMR always has more communication than Conjugate Gradient.

4.4.4 BiCG and Conjugate Gradient:. BiCG has two vector norm computations ($2q$), two vector dot products ($2q$) and three matrix-vector products ($3m$). For comparing the communication costs of BiCG and Conjugate Gradient, their difference can be given as:

$$C_{BiCG} - C_{CG} = 4q + 3m - (4q + 2m) = m > 0 \text{ for all } m > 0. \tag{4.5}$$

This shows that BiCG will always have higher communication cost than Conjugate Gradient.

4.4.5 BCGS and Conjugate Gradient, BCGS and BiCG:. BCGS has the following communication-causing operations: two vector norm computations ($2q$), two vector dot products ($2q$), one VecDotNorm operation q and three matrix-vector products ($3m$). Therefore the total communication cost for BCGS can be given as $5q + 3m$. The difference in the communication costs of the two solvers can be given as:

$$C_{BCGS} - C_{CG} = 5q + 3m - (4q + 2m) = q + m > 0 \text{ for all } q, m > 0. \tag{4.6}$$

This shows that the communication cost for BCGS is more than that of Conjugate Gradient. Also, because BiCG's communication cost is more than that of Conjugate Gradient, BCGS has more communication than BiCG as well.

4.4.6 BCGS and TFQMR:.

The difference in the communication cost of TFQMR and BCGS can be given as follows:

$$C_{TFQMR} - C_{BCGS} = 5q + 4m - (5q + 3m) = m > 0 \text{ for all } m > 0. \tag{4.7}$$

Therefore, TFQMR has more communication than BCGS in all scenarios.

4.4.7 BCGS and GMRES:.

The communication cost for C_{BCGS} is $5q + 3m$ and the cost of C_{GMRES} is $2m + 2q + q * k(k + 1)/2$, the difference in their communication costs can be given as follows:

$$C_{BCGS} - C_{GMRES} = (5q + 3m) - (2m + 2q + q * k(k + 1)/2) \tag{4.8}$$

The above equation can be simplified as :

$$C_{BCGS} - C_{GMRES} = 3q + m - (q * k(k + 1)/2) \tag{4.9}$$

For all the cases so far, the exact communication cost for various operations was not required to compare the communication cost of the solver pairs. For this comparison, although different operations involved are known in these solver implementations, the value of m and q cost variables are required. As described earlier in this section, $m = 2 * n * (p - 1)$ and $q = \log p$. On substituting these values and given the default value of the restart parameter k in GMRES, $k = 30$ and the average number of nonzeros per row (n) for the problem are 5. On further substitution of the k and n values in the above equation gives:

$$C_{BCGS} - C_{GMRES} = -462 * \log p + 10(p - 1) > 0 \text{ for all } p > 257. \tag{4.10}$$

Refer Figure 14 for function plot. The curve below 0 shows that GMRES has more communication than BiCG. For the curve above 0 shows the cost is more for BiCG when $p >= 258$.

Therefore, the communication cost for BCGS is more than the cost for GMRES for all $p > 257$, given p is the number of processors.

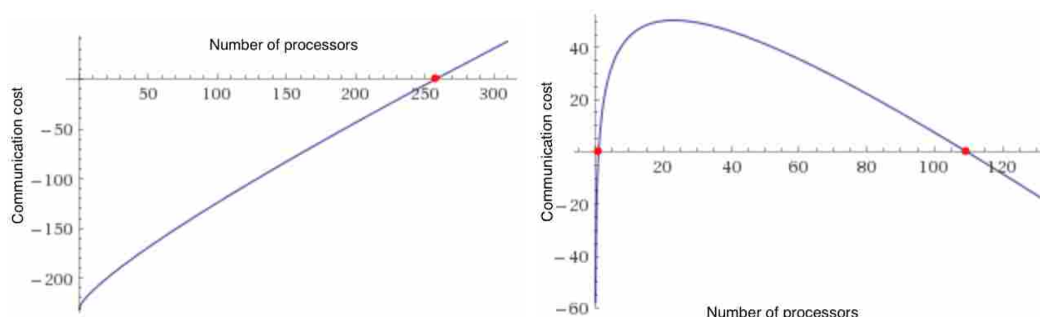


Figure 14. Function plot for communication cost comparison for (1) BCGS and GMRES and (2) TFQMR and GMRES.

4.4.8 BCGS and iBCGS:.

Communication in iBCGS involves 3 MatMult ($3q$), 2 VecDot($2q$), 1 VecNorm(q) and 1 MatMultTranspose(m) operations resulting in a total number of operations to be $3q + 4m$. As described earlier, the cost for BCGS can be given as $C_{BCGS} = 5q + 3m$. The difference in the communication cost of BCGS and iBCGS can be shown as:

$$C_{iBCGS} - C_{BCGS} = 3q + 4m - (5q + 3m) > 0 \text{ for all } q, m > 0 \tag{4.11}$$

After substituting the values of m and q, the above equation can be simplified as:

$$10 * (p - 1) - \log p > 0 \text{ for all } p > 1 \tag{4.12}$$

This shows that the cost of iBCGS is more than BCGS for all $p > 1$.

4.4.9 iBCGS and GMRES:.

iBCGS can be compared with GMRES as follows:

$$C_{iBCGS} - C_{GMRES} = (3q + 4m) - (2m + 2q + qk(k+1)/2) \tag{4.13}$$

Substituting the values, $k = 30$, $m = 2n(p-1)$, $n = 5$ and $q = \log p$, the difference in communication costs can be given as follows:

$$C_{iBCGS} - C_{GMRES} = \log p + 20(p-1) - 465 \log p > 0 \text{ for all } p > 110 \tag{4.14}$$

Therefore, iBCGS has more communication than GMRES for $p > 110$.

4.4.10 BiCG and GMRES:.

For comparing the cost of BiCG and GMRES, the comparison can be made as follows

$$C_{BiCG} - C_{GMRES} = (4q + 3m) - (2q + 2m + qk(k+1)/2) \tag{4.15}$$

On substituting the values of k, m, q and n i.e. $k = 30$, $m = 2 * n(p-1)$, $n = 5$ and $q = \log p$:

$$\log p * k(k+1)/2 - (2\log p + 2n(p-1)) = 2 * n * (p-1) - 463 \log p \tag{4.16}$$

BiCG has more communication than GMRES for $p > 258$.

4.4.11 GMRES and TFQMR.

: To compare GMRES with TFQMR the difference in communication cost is observed as follows:

$$C_{TFQMR} - C_{GMRES} = 5q + 4m - (2q + 2m + qk(k+1)/2) \tag{4.17}$$

Substituting the values for m, q and n, the above equation can be simplified as:

$$20(p-1) - 462 \log p > 0 \text{ for all } p > 109 \tag{4.18}$$

Therefore, the communication for TFQMR is more than that for GMRES for cases where $p > 109$.

Table 7. Operations with communication in ASM.

Krylov method	Operations
Conjugate Gradient	7q + 2m + 4v
GMRES	5q + 2m + qr(r+1)/2 + 4v
Flexible GMRES	4q + 2m + qr(r+1)/2 + 4v
BCGS	10q + 3m + 4v
TFQMR	9q + 3m + 4v
BiCG	7q + 3m + 4v
iBCGS	9q + 4m + 4v

4.5 Compare Communication Cost for Preconditioner Implementations

For comparing the communication cost of the preconditioners used in this work, the preconditioners were applied with the solvers discussed above. This sub-section presents the communication cost arising from the preconditioner implementations offered by PETSC. ASM, Jacobi and Block Jacobi with four variations of the ASM preconditioner are observed.

Jacobi preconditioner is a diagonal scaling preconditioner, i.e. it uses the matrix diagonal $diag(A)$ as the preconditioner. Block Jacobi preconditioner is a block-diagonal matrix, i.e. it is the block version of the Jacobi preconditioner. So the matrix is divided into blocks and each block is solved using the Jacobi method.

ASM preconditioner solves the linear system by dividing it further into smaller domains. The preconditioner has a parameter known as the overlap, which can be varied with values such as an overlap of zero, one, two, three or more. They are represented as ASM(0), ASM(1), ASM(2) and ASM(3) from here on. The overlap value refers to the data that is present between a pair of sub-domains.

ASM has an additional cost because of the communication required for the overlapping data among the sub-domains. The overlapping data has to be sent to one of the neighboring processors. Therefore the communication cost for ASM can be computed as $c_{ASM} = amount\ of\ overlap * transfer\ cost$. The transfer cost can

Table 8. Operations for analytical measurements

Operations	NoPC, Jacobi, BJacobi	ASM(0)	ASM(1)	ASM(2)	ASM(3)
CG	4q+2m	6q+2m+4v	7q+2m+4v	8q+2m+4v	9q+2m+4v
GMRES	2q+2m+qr(r+1)/2	4q+2m+qr(r+1)/2+4v	5q+2m+qr(r+1)/2+4v	6q+2m+qr(r+1)/2+4v	7q+2m+qr(r+1)/2+4v
FGMRES	2q+2m+qr(r+1)/2	3q+2m+qr(r+1)/2+4v	4q+2m+qr(r+1)/2+4v	5q+2m+qr(r+1)/2+4v	6q+2m+qr(r+1)/2+4v
BCGS	5q+3m	9q+3m+4v	10q+3m+4v	11q+3m+4v	12q+3m+4v
TFQMR	5q+4m	9q+4m+4v	10q+4m+4v	11q+4m+4v	12q+4m+4v
BiCG	4q+3m	6q+3m+4v	7q+3m+4v	8q+3m+4v	9q+3m+4v

Table 9. Communication-based ranking of solvers for $p > 258$.

Ranking	NoPC	ASM(0)	ASM(1)	ASM(2)	ASM(3)	Jacobi	BJacobi
CG	1	4	5	6	7	1	1
GMRES	8	14	17	19	21	8	8
FGMRES	8	14	16	18	19	8	8
BCGS	26	29	31	34	35	26	26
TFQMR	39	42	47	48	49	39	39
BiCG	22	25	30	31	33	22	22
iBCGS	36	43	44	45	46	36	36

be given as 0 for ASM(0), q for ASM(1), $2q$ for ASM(2) and $3q$ for ASM(3). On the other hand, Jacobi and Block Jacobi do not have any additional costs associated in the *PCApply* operations, therefore $c_{Jacobi} = 0$ and $c_{BJacobi} = 0$. This also suggests that solvers preconditioned with Jacobi and Block Jacobi will have the same cost of communication as the case of the absence of preconditioning.

Due to a transfer cost introduced in ASM due to the overlap, the communication cost for all the Krylov methods needs to be compared for the case where they are preconditioned with ASM. Table 7 shows the operations for all Krylov methods when applied with ASM. By substituting the values of the cost variables, it can be derived that Conjugate Gradient has the least communication, and Flexible GMRES has the most communication.

4.6 Generate Communication-based Ranking

With the pair-wise comparison and the preconditioner communication cost computation, the solvers can be ranked based on the number of operations and

the cost of each operation. Based on this approach, all the solver-preconditioner pairs can be ranked on the overall communication cost. There are a total of 49 such pairs, so the ranking starts from one till forty-nine. A lower rank would mean less communication for the solver-preconditioner pair, and the pair with the highest rank depicts the most communication. Pairs with the same cost of communication share the same rank. Table 9 shows the communication-based solver ranking from 1 to 49 in order of increasing communication costs. The rows in the table represent the Krylov method and the columns in the table denote the preconditioning options used along with the no preconditioning options. The next chapter describes the empirical evaluation used to test this model-based ranking with empirical results collected for numerical simulation of driven fluid flow in a cavity.

4.7 Summary

The chapter presents the modeling strategy for comparing the scalability of parallel Krylov methods for different input properties, without requiring extensive empirical measurements. We consider the PETSc implementations of Newton-Krylov methods to produce scalability rankings based on our new comparative modeling approach. The analytical modeling examines seven Krylov Method implementations and six preconditioner implementations. We provide the solver ranking for each Krylov methods, including no preconditioning by analyzing and providing a ranking for a total of 49 solver-preconditioner implementations offered by PETSc.

CHAPTER V

COMBINING THE CONVERGENCE AND COMMUNICATION MODELS

This chapter is based on the collaboration [87] among Boyana Norris (UO), Elizabeth Jessup (CU Boulder), and myself. Boyana Norris and I designed the analytical model for ranking a subset of the parallel solvers offered by PETSc. I applied the convergence model and the communication model together to enable solver recommendation techniques for large scale problems. For validating the model, Boyana Norris and I tested the approach on a driven cavity application.

This chapter presents the validation of the communication model presented in Chapter IV by evaluating the model on a numerical simulation of driven fluid flow in a two-dimensional cavity. With the association of the convergence model and the communication model, the approach enables capturing both the convergence behavior and the parallel overhead of the Krylov methods. Combining the two models allows modeling both aspects of Krylov methods: convergence and communication, which facilitates solver recommendations at different parallelism scales.

5.1 Motivation

The machine-learning based approach successfully captures the convergence behavior of the Krylov methods for small-scale problems. The convergence model depends on a training dataset that is collected by solving the linear systems with multiple combinations of solvers and preconditioners. However for problems that require larger processor counts, collecting the training dataset becomes rather expensive and therefore undesirable, as each linear system has to be solved with multiple solver-preconditioner combinations. The communication-based ranking model described in the previous chapter generates the solver ranking which can be

used for making solver predictions for large scale problems (problems that require more than $10,000$ processors.

5.2 Modeling Computation and Communication for Krylov Methods

The convergence model and the communication model, when used in conjunction can build a stronger recommendation system than either of the models as a standalone approach. Using the two models in combination enables capturing the convergence behavior and communication behavior of the parallel preconditioned Krylov methods. This chapter describes how the two models can be used in combination and also presents the evaluation performed on a cavity driven fluid flow application.

The first step involves using the convergence approach to build the machine learning model and classifying Krylov methods based on their convergence behavior. More than $1,800$ matrices from the SuiteSparse matrix collection are used to form the training dataset. The training dataset is collected over a small number of processors, 72, in this case. On applying the supervised machine learning technique, we can identify the “good” preconditioned Krylov methods for each linear system. The training dataset for supervised learning comprises the matrix features, unique solver id and binary class label (“good” or “bad”). The full feature set contains 34 features. We perform the feature reduction technique as described in Chapter III and generate a reduced set with only six features. These features are inexpensive matrix properties which can help in reducing the overall cost of the system. The reduced feature set contains six features, namely Row Variability, Lower Bandwidth, Upper Bandwidth, Diagonal Sign, Diagonal NNZ and Numeric Value Symmetry2. These features have been described in detail in Section 3.6.

A set of eleven solvers and five preconditioners from PETSc along with some parameter configurations for the preconditioners are considered. Each solver-preconditioner pair is assigned a binary class label ("good" or "bad") based on a threshold parameter. The threshold parameter value is a threshold for deciding whether a solver should be labeled as "good" or not, based on the comparison with the best solver timing, i.e. the solver that solved that system in the least time. A threshold value of 0.45 was chosen by varying the parameter value between $[0, 1]$.

Once all the data points are labeled as "good" or "bad", the next step is to perform the binary classification. The classification model is analyzed by the True Positive Rate (TPR) accuracy and the overall accuracy. TPR is the ratio of the true positive instances identified correctly by the model and the actual number of positive instances. The overall accuracy considers the model's capability to correctly identify "good" solvers as "good" and "bad" solvers as "bad". Main focus is on the TPR accuracy, as identifying the "good" solvers correctly is more interesting than correctly identifying the "bad" solvers. For 10-fold cross validation, out of all the classifiers (BayesNet, LibSVM, k-nearest neighbor, ADT, J48) tested, Random Forest classifier achieved the best overall accuracy of 98.8% and TPR accuracy of 98.4%. With a train-test split of $66-34\%$, the overall accuracy achieved by Random Forest is 98.6% and the TPR accuracy of 98.1%.

To further validate the model, the approach was applied on a driven cavity flow simulation, which includes solving a nonlinear PDE discretized on a 100×100 grid. Five different physical configurations were considered by varying Grashof numbers of 1, 10, 100, $1,000$ and $10,000$. During each simulation, multiple sparse linear systems (order $4,000,000$ with nearly $80,000,000$) are solved at each non linear iteration. The driven cavity simulation is selected because the simulation

resembles the properties of many large-scale nonlinear PDE-based applications from domains like astrophysics and aerodynamics. As known, the most expensive part of the simulation is the solution of large, sparse linear systems of equations. For modeling the scalability of Krylov methods, the analytical ranking is generated by comparing the amount of communication involved in these methods. For small-scale problems, the convergence model is sufficient to make solver recommendations. For large scale problems, which need high levels of parallelism, first the convergence model generates a list of "good" solvers. Second, the analytical solver ranking is used and the intersection of the convergence model and the analytical model is used to select the top-ranked solver-preconditioner configuration which is suggested by the convergence model and also highly ranked by the analytical scalability model.

The driven cavity model used for evaluating the model, is a combination of lid-driven flow and buoyancy-driven flow in a 2D rectangular cavity. The lid moves with a steady and spatially uniform velocity and sets a principal vortex by viscous forces. The differentially heated lateral walls of the cavity invoke a buoyant vortex flow, opposing the principal lid-driven vortex. The nonlinear system can be expressed in the form $f(u) = 0$, where $f : Rn- > Rn$. For empirical measurements, five regular grids on a uniform Cartesian mesh are solved, which generate square linear systems with up to $4,000,000$ rows and columns and approximately $80,000,000$ nonzeros. There are four unknowns per mesh point, 2D velocity, viscosity and temperature. The linear system is solved in parallel with all the preconditioned and non-preconditioned Krylov methods, using PETSc 3.8, the current version at the time of this research.

5.3 Empirical Evaluation

Using the convergence model in conjunction with the communication model recommends the Krylov methods that are suggested by the convergence model and are also highly ranked by the communication model. This section presents the empirical measurements collected on the NERSC Edison supercomputer for problems of upto 80 million nonzeros solved on up to $12,288$ processors. The problem configuration was chosen specifically because it is feasible and requires a nontrivial amount of time on larger processor counts. The initial configuration was not changed in any form during the validation.

5.3.1 Results for a $1,000 \times 1,000$ mesh with a Grashof 100.

Our first set of experiments use a $1,000 \times 1,000$ mesh with a Grashof number of 100. Table 10 shows the Krylov methods that are labeled "good" by the Random Forest classifier and their ranking from the analytical communication model for driven cavity simulation on a 1000×1000 mesh with a Grashof 100. Figure 16 shows the ratio with respect to the average time per solve for different processor counts, $12,288$, $6,144$ and $1,536$. The data is sorted by the average time per solve for all the solvers that finished the computation in time. Other solver-preconditioners either failed or were timed out as they were taking too long.

The results are shown in Tables 11, 12, and 13 for different number of MPI tasks used to solve the problem with a GrashOf of 100. The tables are sorted based on the measured average linear system solution time in increasing order. Each row shows the performance of a Krylov method and its comparison with the solver with the best execution time along with the speedup with respect to the default solver/preconditioner combination for PETSc (GMRES/Block Jacobi). The

Table 10. Combining ML-based predictions with the analytical model ranking for systems arising in the driven cavity application (1000×1000 grid) for a Grashof 100.

Grashof=100	
Rank	Krylov method
14	FGMRES/ASM(0)
21	GMRES/ASM(3)
30	BiCG/ASM(1)
31	BiCG/ASM(2)
34	BCGS/ASM(2)
44	iBCGS/ASM(1)
47	TFQMR/ASM(1)
49	TFQMR/ASM(3)

predicted solver configuration, highlighted in bold, is the highest-ranked (based on communication) Krylov method that is also suggested by the convergence model.

The information represented in the tables 11, 12 and 13 for the driven cavity problem on a $1,000 \times 1,000$ grid with Grashof $= 100$ can be graphically represented for different processor counts in terms of the average time per solver and the speedup with respect to the default PETSc solver-preconditioner pair– GMRES with Block Jacobi.

5.3.2 Results for a $1,000 \times 1,000$ mesh with a Grashof $1,000$.

The second set of experiments use a $1,000 \times 1,000$ mesh with a Grashof number of 1000. Table 14 shows the Krylov methods that are labeled “good” by the Random Forest classifier and their ranking from the analytical communication model for driven cavity simulation on a 1000×1000 mesh with a Grashof $1,000$. Figure 17 shows the ratio with respect to the average time per solve for different processor counts, $12,288$, $6,144$ and $1,536$ for a Grashof number of $1,000$. A higher Grashof number reflects a more complex problem. The data is sorted by the average time per solve for all the solvers that finished the computation in time. Other solver-preconditioners either failed or were timed out as they were taking too long.

Table 11. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$ for $1,536$ MPI tasks.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
1,536 MPI tasks:					
35	BCGS/ASM(3)	0.70	1.00	2.90	3
34	BCGS/ASM(2)	0.73	1.05	2.77	3
29	BCGS/ASM(0)	0.73	1.05	2.77	3
44	iBCGS/ASM(1)	0.73	1.05	2.77	3
42	TFQMR/ASM(0)	0.77	1.10	2.65	3
46	iBCGS/ASM(3)	0.77	1.10	2.65	3
45	iBCGS/ASM(2)	0.80	1.14	2.54	3
36	iBCGS/Block Jacobi	0.87	1.24	2.35	3
26	BCGS/Block Jacobi	0.87	1.24	2.35	3
47	TFQMR/ASM(1)	0.90	1.29	2.26	3
43	iBCGS/ASM(0)	0.90	1.29	2.26	3
26	BCGS/Jacobi	0.93	1.33	2.18	3
39	TFQMR/Block Jacobi	1.03	1.48	1.97	3
49	TFQMR/ASM(3)	1.15	1.64	1.77	4
48	TFQMR/ASM(2)	1.33	1.90	1.53	3
31	BCGS/ASM(1)	1.87	2.67	1.09	3
8	GMRES/Block Jacobi	2.03	2.90	1.00	2
14	GMRES/ASM(0)	2.03	2.90	1.00	2
21	GMRES/ASM(3)	2.03	2.90	1.00	3
8	FGMRES/Block Jacobi	2.07	2.95	0.98	2
16	FGMRES/ASM(1)	2.07	2.95	0.98	3
17	GMRES/ASM(1)	2.30	3.29	0.88	2
19	FGMRES/ASM(3)	2.40	3.43	0.85	3
18	FGMRES/ASM(2)	3.13	4.48	0.65	3
19	GMRES/ASM(2)	3.17	4.52	0.64	3
14	**FGMRES/ASM(0)**	**3.20**	**4.57**	**0.64**	**2**

Table 12. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$ for $6,144$ MPI tasks.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
6,144 MPI tasks:					
19	FGMRES/ASM(3)	0.33	1.00	3.90	2
39	TFQMR/Block Jacobi	0.55	1.65	2.36	4
35	BCGS/ASM(3)	0.60	1.80	2.17	3
26	BCGS/Block Jacobi	0.73	2.20	1.77	3
31	BCGS/ASM(1)	0.83	2.50	1.56	3
42	TFQMR/ASM(0)	0.93	2.78	1.41	4
48	TFQMR/ASM(2)	0.97	2.90	1.34	3
47	TFQMR/ASM(1)	1.00	3.00	1.30	3
26	BCGS/Jacobi	1.00	3.00	1.30	3
34	BCGS/ASM(2)	1.07	3.20	1.22	3
8	GMRES/Block Jacobi	1.30	3.90	1.00	2
29	BCGS/ASM(0)	1.40	4.20	0.93	3
8	FGMRES/Block Jacobi	1.47	4.40	0.89	2
49	TFQMR/ASM(3)	1.50	4.50	0.87	3
14	GMRES/ASM(0)	1.57	4.70	0.83	2
16	FGMRES/ASM(1)	1.63	4.90	0.80	2
21	GMRES/ASM(3)	1.73	5.20	0.75	3
18	FGMRES/ASM(2)	1.90	5.70	0.68	3
17	GMRES/ASM(1)	1.93	5.80	0.67	2
19	GMRES/ASM(2)	2.43	7.30	0.53	3
14	**FGMRES/ASM(0)**	**2.47**	**7.40**	**0.53**	**2**

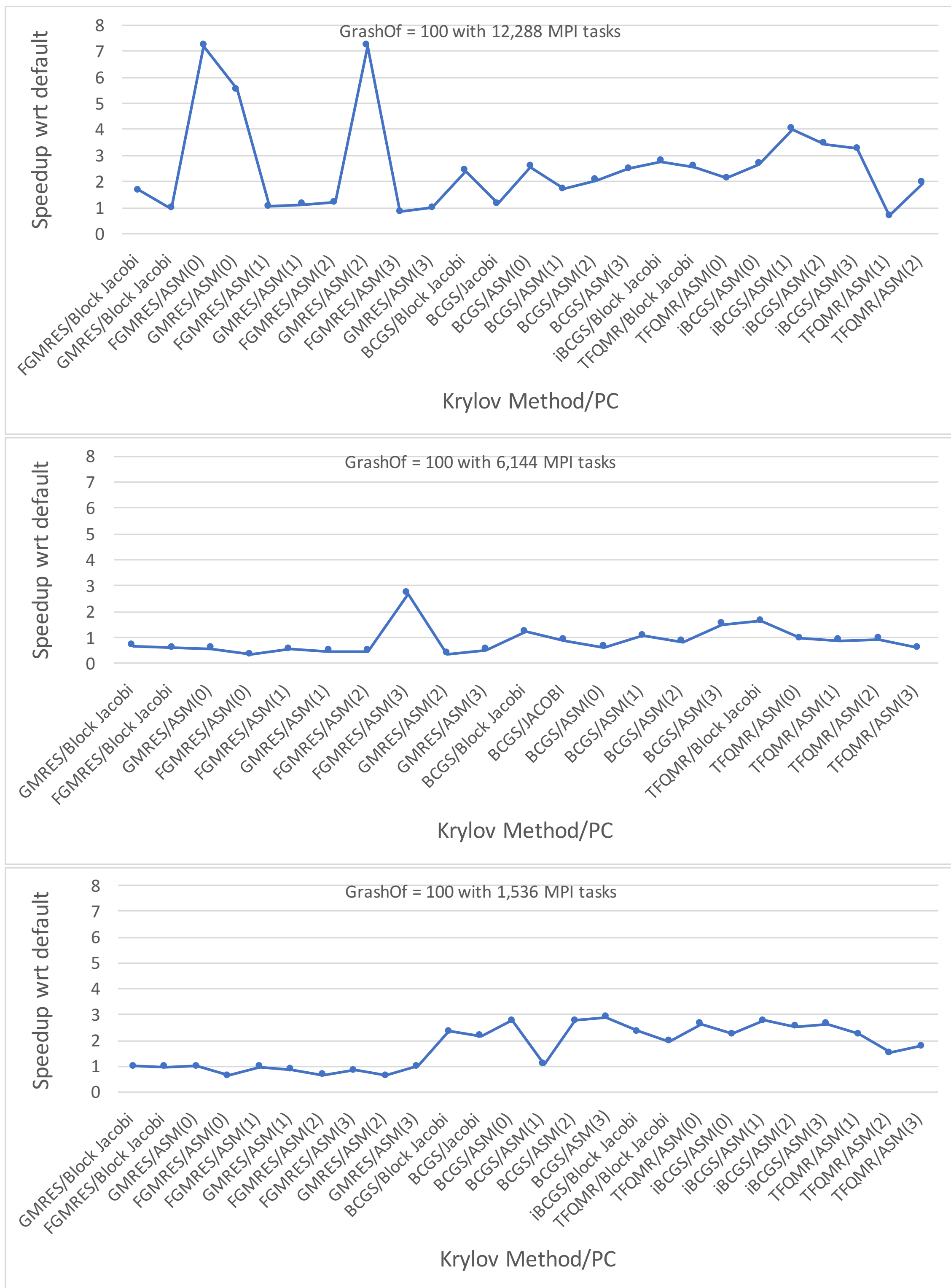
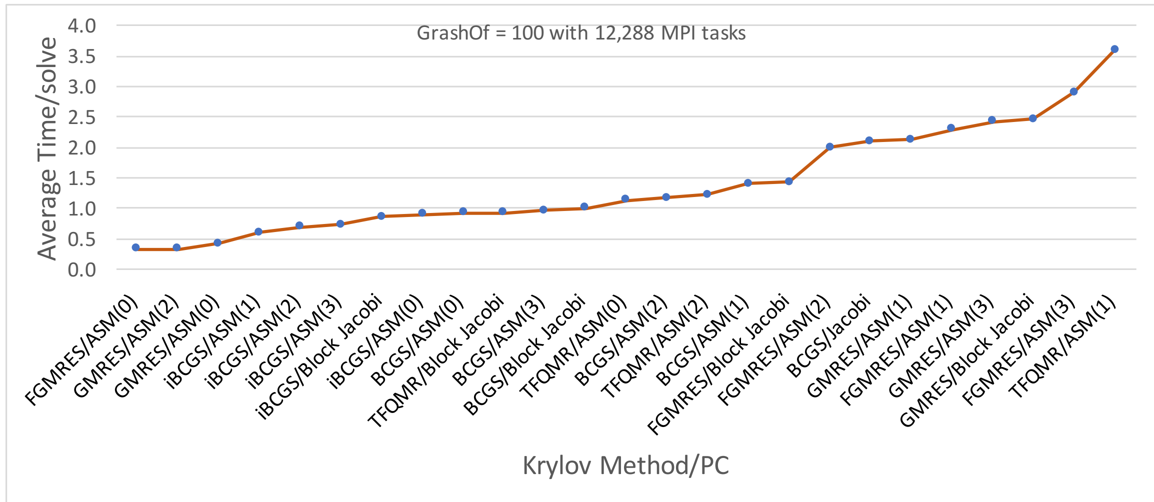
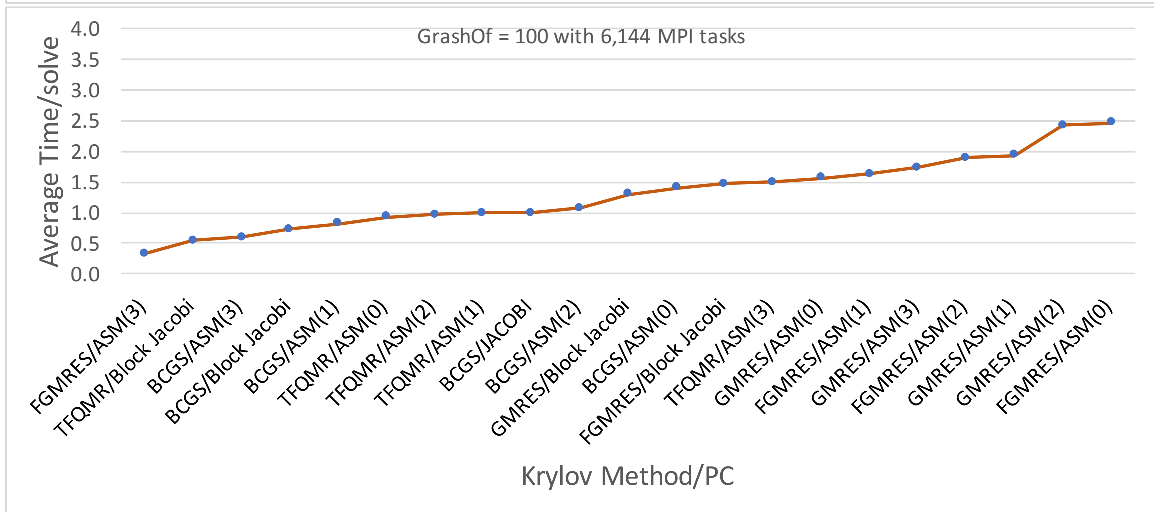


Figure 15. Speedup w.r.t. default solver-preconditioner for 12,288, 6,144 and 1,536 MPI processor counts respectively for the driven cavity problem on a $1,000 \times 1,000$ grid with Grashof$= 100$.





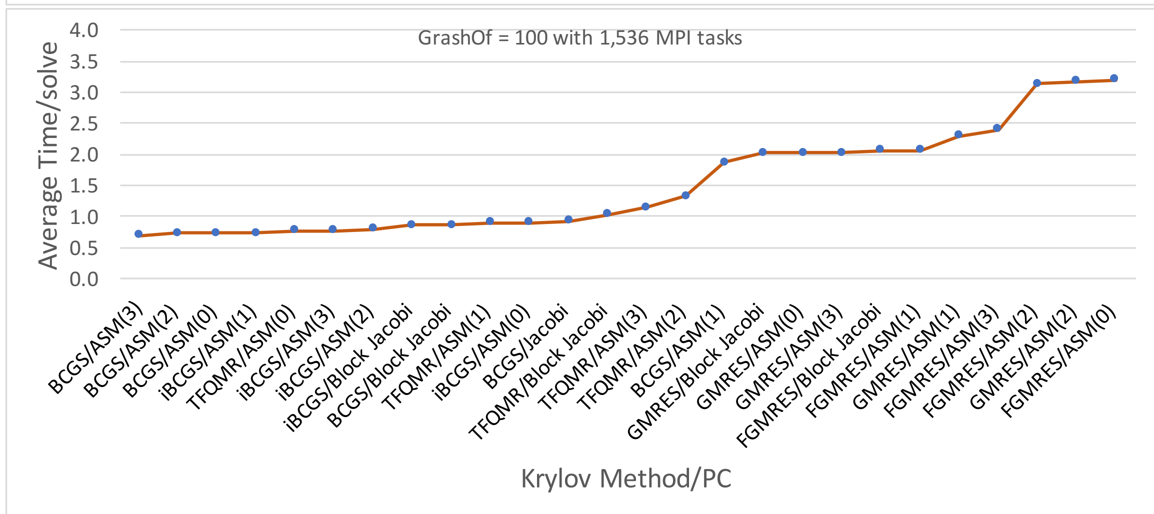


Figure 16. Ratio w.r.t. average time/solve for 12, 288, 6, 144 and 1, 536 MPI processor counts respectively sorted by average time per solve for the driven cavity problem on a 1000×1000 grid with Grashof$=100$.

Table 13. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$ for $12,288$ MPI tasks.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
12,288 MPI tasks:					
14	**FGMRES/ASM(0)**	**0.33**	**1.00**	**7.40**	**2**
19	GMRES/ASM(2)	0.33	1.00	7.40	3
14	GMRES/ASM(0)	0.43	1.30	5.69	2
44	iBCGS/ASM(1)	0.60	1.80	4.11	3
45	iBCGS/ASM(2)	0.70	2.10	3.52	3
46	iBCGS/ASM(3)	0.73	2.20	3.36	3
36	iBCGS/Block Jacobi	0.87	2.60	2.85	2
43	iBCGS/ASM(0)	0.90	2.70	2.74	3
29	BCGS/ASM(0)	0.93	2.80	2.64	2
39	TFQMR/Block Jacobi	0.93	2.80	2.64	3
35	BCGS/ASM(3)	0.97	2.90	2.55	3
26	BCGS/Block Jacobi	1.00	3.00	2.47	2
42	TFQMR/ASM(0)	1.13	3.40	2.18	3
34	BCGS/ASM(2)	1.17	3.50	2.11	3
48	TFQMR/ASM(2)	1.23	3.70	2.00	3
31	BCGS/ASM(1)	1.40	4.20	1.76	3
8	FGMRES/Block Jacobi	1.43	4.30	1.72	2
18	FGMRES/ASM(2)	2.00	6.00	1.23	2
26	BCGS/Jacobi	2.10	6.30	1.17	3
17	GMRES/ASM(1)	2.13	6.40	1.16	2
16	FGMRES/ASM(1)	2.30	6.90	1.07	2
21	GMRES/ASM(3)	2.43	7.30	1.01	3
8	GMRES/Block Jacobi	2.47	7.40	1.00	2
19	FGMRES/ASM(3)	2.90	8.70	0.85	2
47	TFQMR/ASM(1)	3.60	10.80	0.69	1

Table 14. Combining ML-based predictions with the analytical model ranking for systems arising in the driven cavity application (1000×1000 grid) for a Grashof 1,000.

Grashof=1,000	
Rank	Krylov method
14	FGMRES/ASM(0)
36	iBCGS/Block Jacobi
43	iBCGS/ASM(0)
49	TFQMR/ASM(3)

The results are shown in Tables 15, 16, and 17 for different number of MPI tasks used to solve the problem with a GrashOf of 1,000. The tables are sorted based on the measured average linear system solution time in increasing order. Each row shows the performance of a Krylov method and its comparison with the solver with the best execution time along with the speedup with respect to the default solver/preconditioner combination for PETSc(GMRES/Block Jacobi). The predicted solver configuration, highlighted in bold, is the highest-ranked (based on communication) Krylov method that is also suggested by the convergence model. The information represented in the tables 15, 16, and 17 for the driven cavity problem on a $1,000 \times 1,000$ grid with Grashof $= 1,000$ can be graphically represented for different processor counts in terms of the average time per solver and the speedup with respect to the default solver-preconditioner pair– GMRES with Block Jacobi.

5.3.3 Findings.

Figure 15 and 18 show the speedup with respect to the default solver/preconditioner pair, sorted based on their communication ranking. The communication-based ranking is the most effective at 12,288 count for the driven cavity problem with Grashof = 100. On the larger driven cavity problem, with Grashof = 1,000, this trend is not necessarily visible, because the amount of computation increases with the increase in the complexity of the

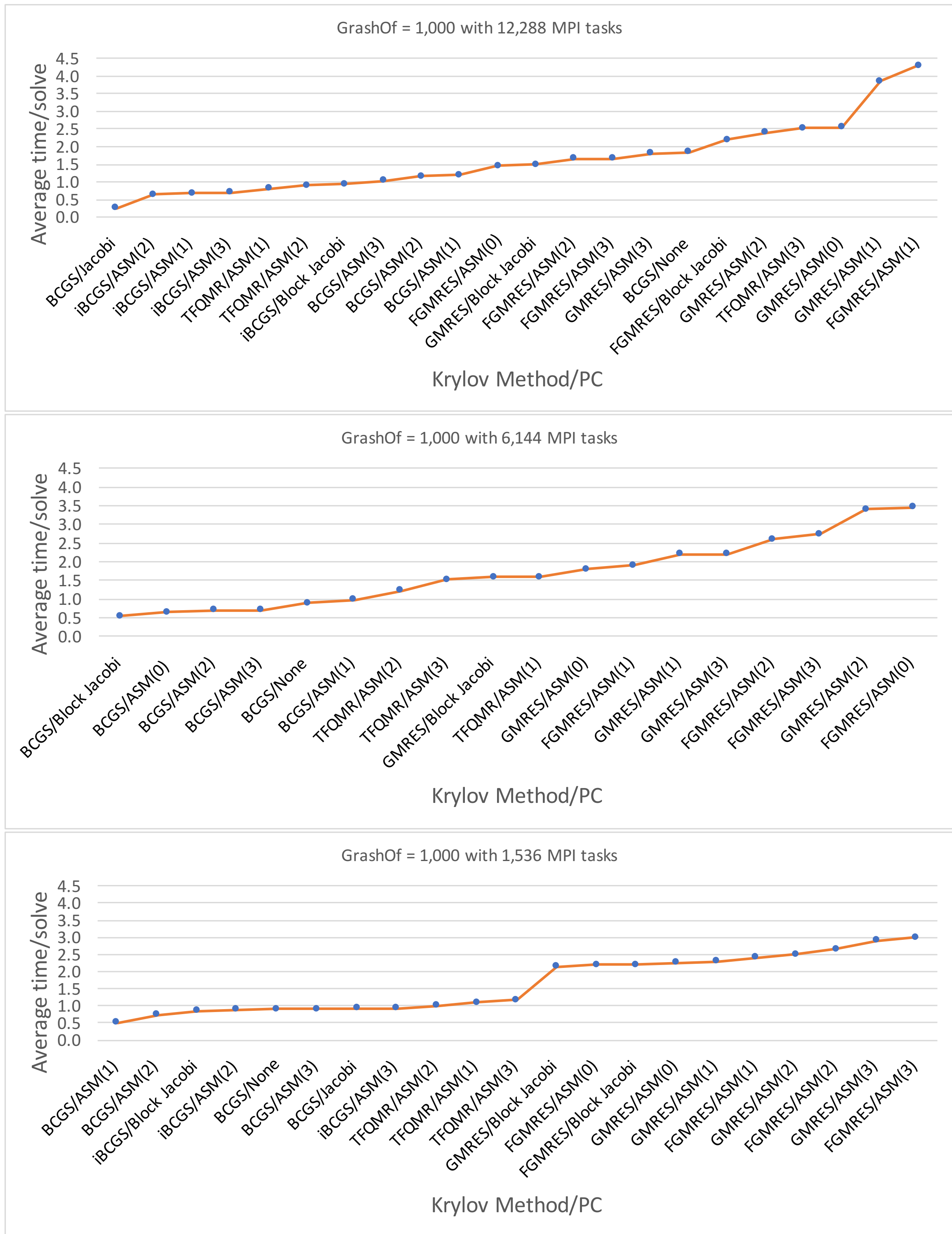


Figure 17. Ratio w.r.t. average time/solve for 12, 288, 6, 144 and 1, 536 MPI processor counts respectively sorted by average time per solve for the driven cavity problem on a 1000×1000 grid with Grashof$= 1,000$.

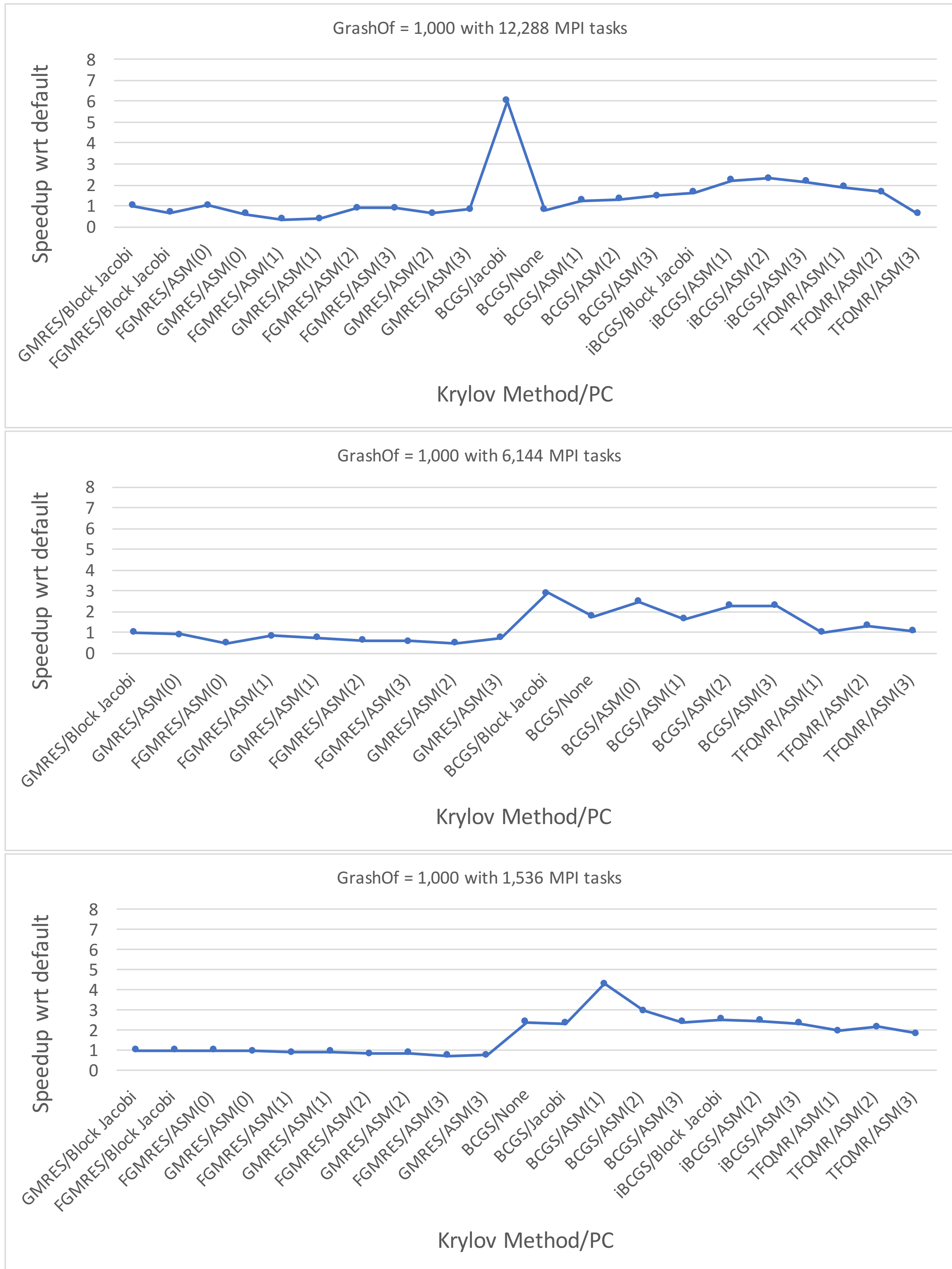


Figure 18. Speedup w.r.t. default solver-preconditioner for $12,288$, $6,144$ and $1,536$ MPI processor counts respectively for the driven cavity problem on a $1,000 \times 1,000$ grid with Grashof$= 1,000$.

Table 15. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$ for $1,536$ MPI tasks.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
1,536 MPI tasks:					
31	BCGS/ASM(1)	0.50	1.00	4.30	4
34	BCGS/ASM(2)	0.73	1.45	2.97	4
36	iBCGS/Block Jacobi	0.85	1.70	2.53	3
45	iBCGS/ASM(2)	0.88	1.75	2.46	4
26	BCGS/None	0.90	1.80	2.39	3
35	BCGS/ASM(3)	0.90	1.80	2.39	4
26	BCGS/Jacobi	0.93	1.85	2.32	3
46	iBCGS/ASM(3)	0.93	1.85	2.32	4
48	TFQMR/ASM(2)	1.00	2.00	2.15	4
47	TFQMR/ASM(1)	1.10	2.20	1.95	4
49	TFQMR/ASM(3)	1.18	2.36	1.82	4
8	GMRES/Block Jacobi	2.15	4.30	1.00	1
14	**FGMRES/ASM(0)**	**2.20**	**4.40**	**0.98**	**1**
8	FGMRES/Block Jacobi	2.20	4.40	0.98	1
14	GMRES/ASM(0)	2.25	4.50	0.96	1
17	GMRES/ASM(1)	2.30	4.60	0.93	1
16	FGMRES/ASM(1)	2.40	4.80	0.90	1
19	GMRES/ASM(2)	2.50	5.00	0.86	1
18	FGMRES/ASM(2)	2.65	5.30	0.81	1
21	GMRES/ASM(3)	2.90	5.80	0.74	1
19	FGMRES/ASM(3)	3.00	6.00	0.72	1

Table 16. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$ for $6,144$ MPI tasks.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
6,144 MPI tasks:					
26	BCGS/Block Jacobi	0.55	1.00	2.91	1
29	BCGS/ASM(0)	0.65	1.18	2.46	1
34	BCGS/ASM(2)	0.70	1.27	2.29	4
35	BCGS/ASM(3)	0.70	1.27	2.29	4
26	BCGS/None	0.90	1.64	1.78	3
31	BCGS/ASM(1)	0.98	1.77	1.64	4
48	TFQMR/ASM(2)	1.23	2.24	1.30	4
49	TFQMR/ASM(3)	1.53	2.78	1.05	4
8	GMRES/Block Jacobi	1.60	2.91	1.00	1
47	TFQMR/ASM(1)	1.60	2.91	1.00	4
14	GMRES/ASM(0)	1.80	3.27	0.89	1
16	FGMRES/ASM(1)	1.90	3.45	0.84	1
17	GMRES/ASM(1)	2.20	4.00	0.73	1
21	GMRES/ASM(3)	2.20	4.00	0.73	1
18	FGMRES/ASM(2)	2.60	4.73	0.62	1
19	FGMRES/ASM(3)	2.75	5.00	0.58	1
19	GMRES/ASM(2)	3.40	6.18	0.47	1
14	**FGMRES/ASM(0)**	**3.45**	**6.27**	**0.46**	**1**

Table 17. Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$ for $12,288$ MPI tasks.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
12,288 MPI tasks:					
26	BCGS/Jacobi	0.25	1.00	6.00	4
45	iBCGS/ASM(2)	0.65	2.60	2.31	4
44	iBCGS/ASM(1)	0.68	2.70	2.22	4
46	iBCGS/ASM(3)	0.70	2.80	2.14	4
47	TFQMR/ASM(1)	0.80	3.20	1.88	4
48	TFQMR/ASM(2)	0.90	3.60	1.67	4
36	iBCGS/Block Jacobi	0.93	3.70	1.62	4
35	BCGS/ASM(3)	1.03	4.12	1.46	3
34	BCGS/ASM(2)	1.15	4.60	1.30	4
31	BCGS/ASM(1)	1.20	4.80	1.25	4
14	**FGMRES/ASM(0)**	**1.45**	**5.80**	**1.03**	**1**
8	GMRES/Block Jacobi	1.50	6.00	1.00	1
18	FGMRES/ASM(2)	1.65	6.60	0.91	1
19	FGMRES/ASM(3)	1.65	6.60	0.91	1
21	GMRES/ASM(3)	1.80	7.20	0.83	1
26	BCGS/None	1.85	7.40	0.81	4
8	FGMRES/Block Jacobi	2.20	8.80	0.68	1
19	GMRES/ASM(2)	2.40	9.60	0.63	1
49	TFQMR/ASM(3)	2.53	10.12	0.59	2
14	GMRES/ASM(0)	2.55	10.20	0.59	1
17	GMRES/ASM(1)	3.85	15.40	0.39	1
16	FGMRES/ASM(1)	4.30	17.20	0.35	1

problem. If the amount of computation is greater or almost comparable to the amount of communication, computation becomes the dominant factor. Therefore, with the technique proposed in this chapter, a solver suggestion is based on the convergence model and the communication model together, because either of the models as a standalone, does not capture both, the computation and communication aspects.

The selection based on communication overhead ranking is expected to be more effective at larger processor counts, where communication plays a bigger role. While several of the ML-suggested solvers achieve significant speedups over the default solver-preconditioner configuration, selecting a solver technique among them based on the communication overhead does not always improve performance for smaller processor counts. To improve the quality of the convergence model predictions, other convergence methods can be investigated that allow ranking based on convergence instead of simple two-label classification.

5.4 Summary

To conclude, using the ML-based convergence and analytical communication models together can be used to estimate the performance of parallel preconditioned Krylov methods. This chapter illustrates the approach for 49 solver-preconditioner pairs including the non-preconditioned cases. The scalability ranking is more effective at larger processor counts. For evaluating the combined approach and using the convergence model and communication model together, on numerical cavity driven fluid flow speedups of up to 7.4 over the default solver configuration are achieved on $12,288$ processor counts. For these results, the convergence model uses training data collected on 72 processor counts alone to make solver recommendations up to $12,288$ processor counts.

CHAPTER VI

MATRIX-FREE FEATURE COMPUTATION

This chapter is based on an on-going collaboration between Boyana Norris (UO), Ben O'Neill (RNET Technologies Inc.), Elizabeth Jessup (CU Boulder) and myself. Ben O'Neill constructed the SS-lite library for matrix-free feature extraction. Boyana Norris constantly provided guidance for the matrix-free implementation, application, and testing. I constructed the training dataset, ML model, reduced feature set and extracted features using SS-lite library. I performed the machine learning classification performance, evaluation with MOOSE/SuiteSparse data, testing with run-time applications. In this chapter, we present results using estimates of the matrix, based on a matrix-free approximation for inexpensive simple and structural features such as Frobenius norm, and diagonal mean.

6.1 Motivation

Traditional sparse matrix data representations involve storing each nonzero element by using data structures, such as compressed sparse row/column, coordinate, diagonal, or hybrid dense/sparse representations. For certain types of computations, such as nonlinear PDE solution via finite-difference Newton-Krylov methods, where the memory requirements of explicitly storing the sparse matrix exceed available capacity, matrix-free approaches can be used [54]. The Krylov solution of the linearized system is computed by using approximations of matrix-vector products based only on the function computing the current discretized solution approximation at each grid point. Because the matrix is not stored explicitly, it is impossible to compute most of the features used in our ML-based

solver selection. Hence, a different set of features must be defined and computed for matrix-free approaches.

We consider the case of large matrices where the memory requirements exceed the available capacity and present a matrix-free feature computation approach for Krylov method selection. The matrix-free routines, implemented directly in PETSc, rely only on approximate matrix-vector multiplication to compute matrix features. They reduce storage by using a small sample of the matrix columns to inform estimates of overall matrix features. We present results using features based on matrix-free eigenvalue approximation, infinity norm, and structural problem features.

6.2 Multiphysics Simulations

Our approach has been demonstrated with multi-domains in [86, 51]. This research focuses on a single domain in which the problems come from a finite-element, multi-physics domain. The matrices used for training and testing of the system both belong to this category. Multi-physics problems are those simulations in which there are multiple physics phenomena involved simultaneously, for instance, thermal effect, fluid forces, and others. Multiphysics problems are of great interest to us because many problems in the field involve multiple physics forces, which generate a set of problems involving a different combination of these physics phenomena. For instance, the Terzaghi's problem of consolidation of a drained medium and the Mandel's problem of consolidation, which are described in detail in Chapter VII, Section 7.2.

6.3 Feature Computation

The primary goal of the machine learning model is to inform algorithm selection at run-time in advanced numerical simulations, thereby reducing overall

run-times and allowing for the efficient usage of our computational resources. To achieve this, it is important that we extract an informative, discriminating and independent set of features capable of inferring the performance of a linear solver routine on a given matrix.

Let t_{FE} represent the time taken to extract the features, let t_P represent the time required to predict the quasi-optimal solver using the convergence model, let t_S be the time required to solve the linear system using the quasi-optimal solver as predicted by the model and let t_{def} represent the time required to solve the linear system using a statically defined default solver. Then, the overall speedup that can be obtained at run-time using our model is:

$$speedup = \frac{t_{def}}{t_{FE} + t_P + t_S} \tag{6.1}$$

The solve times for the quasi-optimal (t_S) and default (t_{def}) solvers are domain and implementation dependent, therefore cannot be optimized in a general setting. Previous research has shown that the time required for model prediction is negligible when compared to the cost of a linear solve. Therefore, the optimization of the feature extraction algorithms is the only avenue towards the optimization of our convergence model. This section presents the two mechanisms used for improving the total time for solving a new linear system.

6.3.1 Reduced Feature Set. Ideally, we would like to compute all features for the incoming linear systems. However, some of the features, such as the eigenvalues, have high computation cost compared to others. Past research [51] has also shown that, in many cases, removing irrelevant features tends to improve the performance and accuracy of the convergence model. To reduce the overall system cost, we reduce the number of features that have to be computed for a new incoming system. Feature reduction stage involves applying

multiple attribute evaluators with different search methods to select the features that are significant and contribute the most towards the classification process. In particular, CfsSubsetEval with BestFirst search method, Gain Ratio, Info Gain, Principle Components evaluators with Ranker search method offered by Weka [49] were used. These attribute evaluators apply different techniques to assess the contribution of each feature. To generate a ranking, the evaluators assign a weight to each feature. The weight determines the contribution of the feature, with a higher weight signaling a higher contribution. Feature reduction is performed by collecting the features that are ranked highly by all or most of the attribute evaluators and removing the lowest ranked features. These highly ranked features form the Reduced feature set 1 (RS1), with only seven features for MOOSE and SuiteSparse datasets. The reduced features are shown in Table 18 and are described in Chapter III, Section 3.10 .

Table 18. Reduced Feature for MOOSE and SuiteSparse Datasets

Reduced feature set for MOOSE	Reduced feature set for SuiteSpare
Absolute Trace	Absolute Trace
Dimension	Dimension
One Norm	Column Diagonal Dominance
Symmetric Infinity Norm	Frobenius Norm
Trace	Trace
Diagonal Mean	Diagonal Mean
Diagonal Non Zeros	Absolute Non Zero Sum

6.3.2 Sample Based Feature Extraction - Efficient and Practical.

In our previous research [52], the matrix features were calculated independently using an external feature calculation library called Anamod [32] and/or directly inside PETSc using built-in PETSc function calls. There are two concerns with this approach. Firstly, the features are computed independently,

thereby demanding the matrix elements to be read into the memory each time a new feature is calculated. Secondly, the overall cost of accessing every matrix coefficient grows with problem size and will likely not scale well for larger systems.

To address these issues, we develop a sampling-based approach for extracting features, using a small $O(1)$ sample of the matrix columns, to provide estimates of overall matrix features. The principle behind the approach is that at a low level the machine learning models can provide informed *estimates* of the solve performance on a given matrix. Because they are estimates, features used by the machine learning model need not be exact. Rather, the feature set represents a collection of informative, discriminating and independent features capable of informing smart decisions with regards to the overall run-time of the given matrix-vector system.

The sampling-based feature extraction approach emerged from the need to extract features with a cost-effective technique. Instead of directly accessing the matrix coefficients, the matrix-free technique utilizes only matrix-vector multiplications to find the matrix inverse for feature computation. The memory requirements of the overall solve are reduced by not storing the matrix values explicitly, enabling the solution of larger systems that would otherwise not fit in the memory. The sole drawback of the approach is that the matrix elements cannot be accessed directly, which further accentuates the relevance of feature selection.

To summarize, the sample based feature extraction algorithm presented in this chapter represents a cheap, efficient algorithm capable of extracting features utilizing only the matrix-vector multiplications. However, there is no algorithmic difference in using the sample based feature extraction techniques on a matrix in which the matrix elements are known, and in a matrix-free system where only the

action of the matrix on a vector is available. For the results presented throughout this chapter, matrix-vector multiplications are used in all cases to extract the values of the sample columns from the given matrix.

Column Extraction with matrix-vector Multiplication Let us consider an $n \times n$ matrix A with columns $\mathbf{a}_j$ such that

$$A = \begin{bmatrix} | & & | \\ \mathbf{a}_1 & \dots & \mathbf{a}_n \\ | & & | \end{bmatrix},$$

The vector $\mathbf{a}_j$ can be extracted through a single matrix-vector multiplication with the jth Euclidean basis vector. The matrix-vector multiplication can be represented as follows:

$$\mathbf{a}_j = A\mathbf{e}_j$$

Here $\mathbf{e}_j = \{0, 0, \dots, 1, 0 \dots\}$ represents the jth standard basis vector. The set $S = \{\mathbf{a}_j : j \in [0, n]\}$ can be formed in a matrix-free environment where the only interaction with the matrix is through the matrix-vector multiplications. In theory, the entire matrix, A, can be built using n matrix-vector multiplications, however, in a practical setting using this technique for building up A is extremely expensive. Rather, the sampling-based approach estimates the feature values using a subset of the matrix columns, $S_s \cup S$, such that $m \ll n$, where m is the number of elements in S_s.

For the matrices that use grid-based PDE methods, each column of A can be roughly attributed to a node in the computational grid. In these cases, physical features such as boundary conditions can dramatically effect the size and number of elements in rows. Thus, when dealing with PDE-based matrices, it is favorable to select a sample that includes a good mix of columns linked to interior and

boundary based grid points. In fact, our testing has shown that it is beneficial to pick and choose the sample columns that are used in each feature calculation. For example, due to the effects of boundary conditions on the contents of some columns, it is best to use columns that translate to interior mesh points to estimate the number of non-zeros in the matrix, whereas for features such as the minimum number of non zeros per row, it is best to only inspect rows related to boundary values. Various other features make similar decisions, using different subsets of the overall sample data to calculate the final value.

One consideration that must be made is the notion of the symmetry of the matrix, A. Symmetry can be very important as some of the solver techniques do not converge for asymmetric matrices, for example, unmodified Conjugate Gradient method. In fact, Conjugate Gradient requires matrices to be symmetric positive definite. A feature such as symmetry cannot be confidently determined via sampling $O(1)$ columns of the matrix A and ideally would be known *a priori*. Additionally, many feature calculations become easier (or trivial) for a symmetric matrix, so our preliminary calculations take in "is symmetric" as an optional argument.

Where appropriate, all features calculated are scaled by the ratio n/m to ensure that the feature calculations reflect estimates of the overall matrix rather than just the sample set. Some features, such as those including maximums or minimums, do not use scaling but instead assume that the value calculated using the sample set is representative of the entire matrix.

Let S_s be a set of sample columns of size m, obtained from a matrix A with r rows and c columns, let B_s represent the subset of S_s whereby we estimate that the columns represent boundary terms in the computational grid and let I_s

represent the subset of S_s where the columns represent interior grid points. Let $A_{ji} = (a_i)_j$ represent the ith component in the ith column vector of A and let $\delta(x)$ be a function such that

$$\delta(x) = \begin{cases} 0 & x = 0 \\ 1 & x \neq 0 \end{cases} .$$

Below is a mathematical representation of feature calculation for some of the features in the sample based reduced feature set:

- Symmetric Infinity norm: The infinity norm of the symmetric part of the matrix is computed as follows:

$$||A||_\infty \approx \frac{c}{m} \max_{i \in [0,r]} \left[\sum_{j \in [0,m]} |(\mathbf{a}_j)_i)| \right]$$

- Diagonal non-zeros: The number of diagonal non-zeros is estimated as

$$dnz(A) = \frac{c}{m} \sum_{a_i \in S_s} \delta((\mathbf{a}_i)_i)$$

- Trace: The trace of the matrix is given by the sum of the diagonal elements of the matrix. Mathematically it can be represented as follows:

$$\frac{c}{m} \sum_{\mathbf{a}_i \in S_s} (\mathbf{a}_i)_i$$

- Absolute Trace: The absolute trace is given as the absolute value of the trace.

- One norm: This feature is the absolute sum of column for all the samples.

- Minimum non-zeros per row: Based on in depth testing, the minimum number of non zeros was calculated using the set of boundary samples, B_s as follows:

$$MNPR(A) \approx \frac{c}{m} \min_{\mathbf{a}_i \in B_s} \left[\sum_{j \in [0,r]} \delta((\mathbf{a}_i)_j) \right]$$

– Lower bandwidth: The lower bandwidth is calculated using the entire sample set as:

$$LB(A) \approx \max_{\mathbf{a}_i \in S_s} \left[\max_{j \in [i+1,r]} (j-i) * \delta((\mathbf{a}_i)_j) \right]$$

– Upper bandwidth: The upper bandwidth is calculated in the same way as the lower bandwidth.

– Column variance: The column variance is defined to be the average variance in the elements of each column, across all columns in the sample set. The variance in a sample column is calculated using the standard formula for variance:

$$var(\mathbf{a}_i) = \frac{1}{r} \sum_{i \in [0,r]} ((\mathbf{a}_i)_j - \mu)^2$$

where μ is the mean of the elements in $\mathbf{a}_i$.

– Nonzero pattern symmetry: A matrix is determined to have nonzero pattern symmetry if $(\mathbf{a}_i)_j = (\mathbf{a}_j)_i$ for all $\mathbf{a}_i \in S_s$ (see above for a discussion on the risks of estimating symmetry using a sample based approach).

– Number of non-zeros: The number of non-zeros is calculated as

$$nnz(A) \approx \frac{c}{m} \sum_{\mathbf{a_i} \in S_s} \left[\sum_{j \in [0,r]} \delta((\mathbf{a}_i)_j) \right]$$

In an effort to maximize efficiency, the sampling-based feature extraction routine has been implemented directly in PETSc using a single loop where, for each column in the sample set, a single matrix-vector multiplication, followed by a single loop through the values of the column is completed. If multiple cores are used, the feature extraction is completed in parallel. In the parallel setting, the only communication between processors is completed during the matrix-vector

multiplication, as required by PETSc. A single MPIReduce call is made at the completion of the feature extraction stage, at which point the root processor completes the final collation and calculation of the matrix-free features. In a practical setting, the root processor then feeds those features into the machine learning algorithm before scattering back the details of the quasi-optimal solver to the remaining processors.

Although this computation-then-communication pattern is extremely efficient in terms of latency, it is somewhat bandwidth heavy. For example, the entire square dataset, with m number of samples and m^2 elements, must be sent to the root process so that the symmetric features can be calculated. Given that we require m to be small, we do not foresee a problem because, in cases where the need for a large m arises, the cost of the m matrix-vector multiplications performed by the KSP method, both in terms of time and memory, will likely dominate any costs associated with the high bandwidth MPIReduce call.

6.4 Machine Learning Classification Performance

Machine learning has been a popular choice for supervised learning because of its capability to improve its performance on its own, without any instructions from humans. Another reason why machine learning is widely accepted is its ability to learn well, very quickly. Some of the applications of machine learning include face recognition, fraud detection, email spam, and others. In our work, we use machine learning to capture the convergence behavior of solver configurations. We apply supervised machine learning algorithms to classify solvers as "good" or "bad", based on their solve time. We test and compare the accuracy results for BayesNet [14], k-nearest neighbor [22], Alternate Decision Trees [44], Random Forest [17] and J48 [68] algorithms.

For each input matrix, we compute the features (as described in the previous section) and combine them with the unique solver-preconditioner id based on the solver-preconditioner combination that was used to solve the system. Once the system is solved, we assign a binary ("good" or "bad") label to each data point, based on the time taken to solve the system. The label is assigned using a threshold parameter. The value of this parameter is varied from $\{0, 0.5\}$ and how close the solver time of a given solver-preconditioner is, in comparison to the best-performing solver method. If the new solver time is within the threshold, then the data point is labeled as "good" and otherwise "bad". The matrix features, unique solver-preconditioner id, and class label are combined and fed as input to the convergence model.

Next, we perform supervised learning with the class label as the output. With our full feature set and the reduced sets, we train the convergence model and test it on our test data set to measure the accuracy of the model for correctly identifying the "good" solvers. The accuracy of the model is measured by the true positive rate (TPR) which is the probability that the classifier predicts a "good" entry as "good". It is computed as follows:

$\mathrm{TPR} = \mathrm{TP}/\mathrm{P} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$, where P is the actual number of positive instances, i.e., solvers labeled as good, TP are the number of true positives and FN are the number of false negatives. For testing the model, we perform two types of tests: 10-fold cross-validation and 66-34 % train-test data split. The dataset is comprised of data points which are obtained by solving the matrices from the MOOSE and SuiteSparse datasets, with various preconditioned Krylov methods from PETSc. Various ML algorithms are used for classifying the solvers namely BayesNet, Random Forests, J48, k-nearest neighbor with 10 neighbors, Random

Table 19. Convergence model accuracy and build time for 10 -fold cross-validation with RS1 set for SuiteSparse dataset.

Method	Good Accuracy (%)	Overall accuracy (%)	Build time (secs)
BayesNet	61.3	89.2	1.3
RF(100)	**77.3**	**93.8**	125.70
ADT	22.70	86.3	7.99
knn(10)	52.5	88.6	0.08
J48	75.5	93.3	10.52

Table 20. Convergence model accuracy and build time for $66 - 34\%$ train-test split with RS1 set for SuiteSparse dataset.

Method	Good Accuracy (%)	Overall accuracy (%)	Build time (secs)
BayesNet	59.2	89.0	1.46
RF(100)	**75.4**	**93.4**	101.31
ADT	22.4	86.2	9.61
knn(10)	49.6	88.0	0.02
J48	73.8	92.9	10.27

Forest with 100 trees and Alternate Decision Trees. Classifying with different ML techniques enables us to compare and choose the best-performing ML method.
To perform the classification analyses, two sets of features are used for both the datasets. The first set consists of the full feature set and the other set includes the reduced feature set. In each evaluation, we perform 10-fold cross-validation and 66-34 % train-test split, to ensure that the ML methods are evaluated with different rearrangements of the dataset.

6.4.1 Classification evaluation on SuiteSparse dataset. The best results for the full feature set and reduced feature sets are obtained from Random Forest. Due to the high cost of feature computation for all the features, using all features for classification is not preferable. In this work, we present results obtained with the reduced feature set used for ML classification. The solver timings were

computed using 38 processors on our *Arya* server, which is an 18-core Intel server with 256 GB DDR4 RAM, with two Intel Xeon E5-2699 v3 CPUs.

After performing data cleaning, We have a total of 110,709 data points, out of which 16,782 are labeled as "good" and the rest of them as "bad" using 38 processors. Table 19 and 20 show the accuracy for all these ML methods for the reduced set (RS1). For 10-fold cross-validation, for the reduced set (RS1)), Random Forest, with a total of 100 trees, achieved a "good" solver accuracy of 77.3% and overall accuracy of 93.8%. J48 achieved an accuracy of 75.5%, as the "good" solver accuracy and overall accuracy of 93.3%. The build time for Random Forest and J48 are as 125.7 seconds and 10.52 seconds respectively. For $66 - 34\%$ train-test split, Random Forest performed slightly better than J48 by achieving a "good" solver accuracy of 75.4% as compared to 73.8% as achieved by J48.

6.4.2 Classification evaluation on MOOSE dataset. The best results for the full feature set and reduced feature sets are obtained from the J48 classifier. Since using all features for building the classification model is not preferable, we focus on the reduced set classification accuracy. After data cleaning, we have a total of 62,702 data points, out of which 7,710 are labeled as "good" and 54,992 as "bad". The solver timings were computed using a single processor on the *Artemis* cluster at the University of Oregon. Table 21 and 22 show the accuracy for these ML methods for the reduced set (RS1).

6.4.3 Observations based on classification evaluations. In our previous work [87], we observed that Random Forest was a good classifier selector because of its high accuracy, with J48 slightly less accurate. Based on the classification evaluations performed in this work, we make two observations. First, for the SuiteSparse dataset, Random Forest performed the best, achieving

Table 21. Convergence model accuracy and build time for 10-fold cross-validation with RS1 set for MOOSE dataset.

Method	Good Accuracy (%)	Overall accuracy (%)	Build time
BayesNet	74.3	93.0	0.56
RF(100)	72.6	93.3	30.58
ADT	97.1	95.2	5.05
knn(10)	97.2	96.3	0.04
J48	**98.5**	**96.5**	0.72

Table 22. Convergence model accuracy and build time for $66 - 34\%$ train-test split with RS1 set for MOOSE dataset.

Method	Good Accuracy (%)	Overall accuracy (%)	Build time
BayesNet	67.0	92.7	0.38
RF(100)	74.7	93.9	24.42
ADT	97.0	95.4	4.21
knn(10)	95.9	96.3	0.01
J48	**98.6**	**96.6**	0.44

an accuracy of 77.3% and J48 was the second best at around 75.5% for 10-fold cross-validation. For $66 - 34\%$ train-test split, Random Forest was at 75.4%, whereas J48 was at 73.8%. Overall, there wasn't much difference between the two classifiers in both the scenarios. For the MOOSE dataset J48 outperformed Random Forest by achieving an accuracy of 98.50% for 10-fold cross-validation and Random Forest achieved an accuracy of 72.60%. For 66-34% train-test split, J48 correctly identified 98.60% of the "good" solvers, outranking Random Forest (74.70%) by a substantial difference. Secondly, the build time is substantially less for J48 classifier as compared to Random Forest. For these two reasons, we decided to use the most recent C implementation [66] of C5.0 algorithm (an equivalent of the J48 algorithm in Java).

6.4.4 Classification evaluation with C5.0 algorithm.

C5.0 is the latest implementation of the J48 algorithm that uses information gain to build decision trees. At each level, entropy/ information gain is computed to identify the class in the training set. The attribute with the highest information gain is selected and that attribute becomes the root node for the decision tree. The process is repeated until all the attributes have been used. The attributes can be used multiple times in the decision tree to make the decision.

Table 23. Classification model "good" accuracy comparison for C5.0 and J48 for SuiteSparse and MOOSE dataset

Dataset	SuiteSparse		MOOSE	
Test	C5.0 (%)	J48 (%)	C5.0 (%)	J48 (%)
10-fold cross-validation	73.7	75.5	**98.0**	98.5
$66 - 34\%$ train-test split	71.3	73.8	97.9	98.6

We use the C5.0 algorithm for classification for the MOOSE dataset with the reduced feature set. With the reduced set (RS1), with only 7 features, it achieved an accuracy of 98% for both validations: 10-fold cross validation and train-test split. For the SuiteSparse dataset, with the reduced set the accuracy was 73.7% and 71.3% respectively. Table 23 shows the "good" accuracy for C5.0 and J48 implementations. We prefer switching from Java to C mainly because C offers the most recent implementation of the C5.0 algorithm.

To validate the usability of C5.0, we also performed prediction analysis, where we train the classifier with 80% of the dataset and test it on the rest 20% of the dataset. For the RS1 MOOSE test set, there were $1,525$ "good" instances in the test set, out of which $1,501$ were correctly classified establishing a "good" solver accuracy of 98.42%. In the RS1 SuiteSparse test set, there were $3,348$ "good" instances, out of which $2,488$ were correctly classified, thus achieving a "good"

solver accuracy of 74.31%. Tables 24, 25 show the solver configurations that were most likely to perform well among all the configurations we tested and lists the most-frequently used solver preconditioner combination for our experiments for both the datasets. The numbers in the configuration denote the overlap value for ASM, GASM and the preconditioner factor level for ICC preconditioner.

Table 24. Top 5 solvers that were labeled as "good" as a percentage of all the "good" solvers for MOOSE dataset.

Occurrence (%)	Solver-PC configuration
28.04	iBCGS,ASM(3)
25.84	iBCGS,ASM(1)
19.65	FGMRES,ASM(1)
12.85	FGMRES,ICC(1)
12.12	FGMRES,ICC(3)

Table 25. Top 5 solvers that were labeled as "good" as a percentage of all the "good" solvers for SuiteSparse dataset.

Occurrence (%)	Solver-PC configuration
3.93	DGMRES, GASM(0)
3.21	FGMRES GASM(0)
2.97	Chebyshev,GASM(0)
2.97	GMRES,GASM(0)
2.73	GMRES,GASM(3)

6.4.5 Solver ranking and validation. With the ML model, we generate a list of "good" solvers for new linear systems. The "good" solvers list is an unordered list of solver and preconditioner combinations. In this work, we also implement a prediction model which predicts the run time of solver-preconditioner combinations. It is done by using the Ridge regression technique offered by Scikit-

learn [67]. Once the model is trained, we predict the solver time and sort the list of solvers based on the predicted times to obtain a ranking.

With this scheme, we obtain ranks for all the “good” solvers for each matrix. We also performed a comparison of the solver timing for the solvers that were ranked number 1 by this system and the baselines used by PETSc. The baseline (default) solver-preconditioner used by PETSc for sequential runs is GMRES with ILU, with a factor level of 0. The default solver configuration for parallel runs is GMRES with Block Jacobi. We obtain speedups from using the default solver versus the solver ranked number 1 by this ranking scheme. The speedup can be given as

$$Speedup = \frac{\text{Time taken by the default PETSc solver}}{\text{Time taken by the top solver (rank 1)}}$$

For the MOOSE dataset, the minimum, maximum, mean and standard deviation of speedups obtained are 1.0, 875.37, 7.58 and 36.21 respectively. For the SuiteSparse dataset, the minimum, maximum, mean and standard deviation of speedups obtained are $1.14, 92464.56, 747.16$ and 4962.50. For the majority of the cases, the top-ranked solver was indeed the solver with the best time. For the outliers, they were ranked second and third best out of more than thirty solvers, but without much speed loss relative to the top-ranked solver and definitely a speedup relative to the baseline.

We applied the solver ranking technique to predict “good” solvers for the dataset. The MOOSE dataset has substantially more “bad” solver data points as compared to the “good” solver data points. This is because we chose a tight threshold of 0.3 (for MOOSE dataset) and 0.4 (for SuiteSparse dataset), which allows only those solvers to be labeled as “good” that have a solve time of 1.3 times

of the best solver (fastest solver) timing for MOOSE dataset and a solve time of 1.4 times of the best solver (fastest solver) timing for the SuiteSparse dataset. Despite the uneven split of “good” and “bad” instances, the solver ranking technique does not rank any actual “bad” solvers as the best solver (rank: 1). This ensures that the best solver obtained from this ranking technique always returns a solver within an acceptable range of solver timing.

6.5 Testing in Run Time Applications

Once the convergence model has been tested for classification performance, the next step involves testing it in runtime applications. As stated earlier, the primary run-time cost of the machine learning models is feature extraction. For smaller matrices, the costs associated with feature extraction often outweighs any performance benefits that can be achieved through the models. However, for matrices where the number of non-zeros is larger, the cost of feature extraction is often small in comparison to the solve time. This is a good result, as the primary goal of the proposed models is to speed up the solution of the large sparse systems that arise in high fidelity numerical simulations.

To test the models in a runtime application, we developed a new PETSc KSP solver that can be used in existing PETSc based simulations using a simple command line parameter. The new KSP solver handles all aspects of using these smart algorithm selection models at runtime, including loading the pre-built serialized machine learning model and extracting the features from the matrix we are looking to solve. Once a quasi-optimal solver has been determined, the new KSP solver sets up the quasi-optimal solver as another internal KSP solver and uses that to solve the original system. In this way, we can support automatic runtime solver selection in most PETSc based applications.

For testing the automatic solver selection, we use the pre-built classification model in a transient nonlinear driven cavity in two dimensions, provided by PETSc. The two-dimensional driven cavity problem is solved in a velocity-vorticity formulation. The flow can be driven with the lid or with buoyancy or both. The lid velocity represents the dimensionless velocity of the lid. The grashof represents the dimensionless temperature gradient and the prandtl shows the dimensionless thermal/momentum diffusity ratio. The problem is modeled by the PDE systems in the unit square, which is uniformly discretized in each of x and y in the simple encoding, as shown below:

$$-Lap(U) - Grad_y(Omega) = 0$$

$$-Lap(V) - Grad_x(Omega) = 0$$

$$Omega_t - Lap(Omega) + Div([U * Omega, V * Omega]) - GR * Grad_x(T) = 0$$

$$T_t - Lap(T) + PR * Div([U * T, V * T]) = 0$$

We apply the convergence model based on the sampling-based approach, on the driven cavity flow simulation, which involves the solution of a nonlinear PDE discretized on a regular 100×100 grid and a 128×128 grid. We consider two different physical configurations with varying lid velocity to be 0.1 and 1.0, a constant Grashof number of 100, and a constant Prandtl value set to 1. Different lid velocities result in different numerical properties of the resulting linear system. During each simulation, at each nonlinear iteration, multiple sparse linear systems are solved.

We use our matrix-free feature selection approach for generating solver suggestions for the driven cavity application. For our column sample, we use 20 edge columns and 10% of the total columns as the number of 10 interior columns. PETSc allows using the explicit-matrix and matrix-free setting by providing a command line argument $-snes_mf$ for the matrix-free setting. We obtain and compare the results for both these settings on 24 processor count on Arya cluster at the University of Oregon.

6.5.1 Driven cavity problem 100×100 grid.

The results discussed in this part of the section employ a 100×100 grid with the following non-linearity parameters: Grashof of 100, lid velocity of 0.1, prandtl of 1. The solver suggestion by our convergence modeling approach based on the matrix-free feature computation is LGMRES with the GASM(0) preconditioner, where 0 is the overlap parameter which is used to extend the local subdomains. With the explicit-matrix setting in PETSc, the speedup obtained for 24 processor count is 1.00. In the matrix-free setting, the speedup obtained is 1.25, with respect to the PETSc default solver-preconditioner for parallel runs. The number of matrix-vector products saved as a result of applying the matrix-free technique in the explicit-matrix setting in PETSc are 99 and with the matrix-free setting are 360. Table 26 shows the speedup and matrix-vector multiplication reduction for all cases, with the values, rounded off to two decimal places.

6.5.2 Driven cavity problem 128×128 grid.

The results discussed in this part of the section employ a 128×128 grid with the following non-linearity parameters: Grashof of 100, lid velocity of 1, prandtl of 1. The solver suggestion made by the convergence modeling approach with the matrix-free feature computation is the LGMRES with GASM(0) preconditioner. The speedup

Setup	Grid	Grashof	Lid velocity	Speedup w.r.t default solver time	MatMult reduction
Explicit-matrix	100×100	100	0.1	1.00	99
Explicit-matrix	128×128	100	1.0	1.03	958
Matrix-free	100×100	100	0.1	1.25	360
Matrix-free	128×128	100	1.0	**1.81**	**133,702**

Table 26. Speedup w.r.t default solver time and number of MatMult operations saved.

obtained for 24 processor count (shown in Table 26) is 1.03, with respect to the default solver-preconditioner in the explicit-matrix setting. For the matrix-free setting with PETSc, a speedup of 1.81, with respect to the PETSc default solver-preconditioner is obtained, by using the solver recommended by our convergence model. The number of matrix-vector products saved as a result of applying the matrix-free technique is 958 and $133,702$ in the explicit-matrix and matrix-free setting respectively.

6.6 Summary

In this work, we introduce a matrix-free approach for computing matrix properties and demonstrate an ML approach for selecting well-performing methods. We present our results for the convergence model using the reduced sets which have seven inexpensive matrix features. We apply our technique to real-world applications for validation and developed a new KSP solver to automate runtime solver selection for a wide variety of PETSc applications. With the matrix-free setting, we were able to achieve a speedup of up to 1.81 with respect to the default solver time and were able to reduce up to $133,702$ matrix-vector products.

In the future, we will test our matrix-free approach on large real applications and include parallel runs for the ML model training for the MOOSE dataset. We will also explore building a dataset with only matrix-free applications. Another aspect of our future work includes expanding our solver-preconditioner subset to include other solvers including direct solvers.

CHAPTER VII

SOLVER SELECTION IN FINITE-ELEMENT MULTIPHYSICS SIMULATIONS

The work presented in Chapters III, IV, V is part of our Lighthouse project [61], which focuses on providing support for sparse linear solver selection based on specific problem features and solver performance on a given architecture. The solver selection in Lighthouse relies on classifying solvers based on their performance and the features of the input matrix and then predicting the best-performing solver configuration when solving new problems. As shown previously, we build models using the SuiteSparse matrix collection, which contains matrices from a wide variety of domains. However, it is limited to mostly small-scale problems. This chapter focuses on a set of use cases based on a single framework, the multiphysics object-oriented simulation environment (MOOSE) [46], which is a finite-element, multiphysics framework that leverages other toolkits, notably PETSc. MOOSE aims to make predictive modeling accessible and scalable, especially in the field of multiphysics framework by allowing fuels and materials scientists to develop numerous applications that predict the behavior of fuels and materials under operating and accident conditions.

The contributions of this work include the definition of a new set of linear system properties, which are used as the features in the machine learning problem specification. We then apply the classification to a set of examples in the MOOSE framework, achieving high accuracy when targeting problems in the more limited domain of finite element multiphysics applications.

7.1 Motivation

Our approach has been demonstrated with multi-domains in [86, 51]. The research focuses on a single-domain in which the problems come from finite-

element, multiphysics domain. The automated solver selection capabilities of Lighthouse are particularly useful to the target MOOSE users, who are scientists who don't have in-depth knowledge of computer science and would like to develop an application by leveraging the "plug and play" component organization of the MOOSE simulation platform. Automatic solver selection can improve the execution time and reliability of multiphysics simulation systems. The goal is to choose a solver that is appropriate for the sub-problem and is also efficient. In the past, we have used our approach for linear systems from multiple domains. This chapter focuses specifically on solving non-linear systems from a single domain.

7.2 MOOSE framework

Advanced modeling and simulation presently include multiphysics simulations, computational fluid dynamics, high energy physics, computational biology, and computational finance. In addition, numerical simulations are applicable across many other disciplines such as modeling and simulating aircraft, spacecraft, rocket, and propulsion systems. In physical sciences, non-linear systems are more interesting to scientists as most systems in these fields are nonlinear in nature. Nonlinear systems are more complicated, sophisticated and unpredictable than linear systems which make a good dataset for testing our approach with nonlinear systems.

Multiphysics problems are those simulations in which there are multiple physics phenomena involved simultaneously, for instance, thermal effect, fluid forces, and others. These physics forces impact the performance of the products and materials in use. Many problems involve coupled systems as well. Coupled systems are the problems in which the two systems under consideration, interact with each other simultaneously. For instance, the application of pore pressure with

mechanical forces to observe volume expansion is Poro mechanics coupling. In this section, we talk about the various physics forces involved in the simulation and the materials in use for the problems used in our experiments.

The matrices we consider are from the following modules: chemical reactions, phase field, tensor mechanics, Richard's and solid mechanics. The matrices used for training and testing the solver suggestion technique, both belong to the multiphysics domain. These are randomly selected among all test modules that explicitly form the non-linear system and/or preconditioner. Each of these modules describes the partial differential equations to be solved.

Figure 19 shows the MOOSE architecture mentioned in [46]. As shown in the figure, MOOSE leverages PETSc libraries through a massively parallel finite-element framework called libMesh. The heavy dependency of MOOSE on the PETSc library provides huge flexibility for MOOSE developers and users. The data types and function calls in PETSc can be directly used in MOOSE source code.

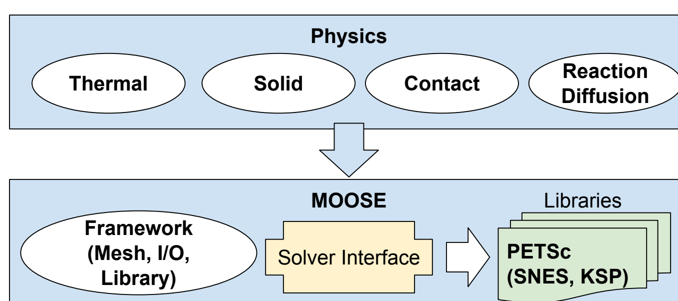


Figure 19. MOOSE Architecture

Multiphysics problems involve multiple physics forces, which generate a set of problems involving a different combination of these physics phenomena. For example, one of the problems we solve is Terzaghi's problem of consolidation of a drained medium. Figure 20 shows the visual representation of the consolidation problem. As per Terzaghi's Principle, when a solid material is subjected to stress,

it is opposed by the fluid pressure of pores in the rock. Terzaghi's problem involves a sample of saturated soil placed in a bath of water constrained on the sides and the bottom, leaving only the top open. The constrained sides and bottom are also impermeable. Initially, when pressure is applied, it is unstressed. After some more stress applied on the soil's top, it leads to slow compression of the soil squeezing the water out from the soil from the top. The situation involves the following physics phenomena: fluid displacement, fluid mobility, pore pressure, consolidation degree, poromechanics coupling, stress divergence tensors.

Another instance of a multiphysics problem in the problems set we consider is the Mandel's problem of consolidation. Mandel's problem involves a drained medium in which the porepressure within the sample is monitored. The sample is in plane strain. It is squashed with constant force by frictionless, impermeable plattens on its top and bottom surfaces. The fluid is allowed to leak out from the sides. Porepressure, velocity, fluid mobility, stress, force, and displacement are the physics phenomena involved in this problem. Mandel's problem is a two-dimensional problem that involves diffusion and time-derivate phenomenon for its simulation.

MOOSE supports predictive modeling by allowing fuels and materials scientists to develop applications that predict the behavior of fuels and materials, some of the problems focus on learning about the properties of the materials under consideration. For instance, conducting a test for identifying the default material interface in Derivative Material Interface. The test should pass only if the construction order of the materials using this interface does not influence the outcome. A comparatively simpler check would be conducting a test that validates the application of the chain rule correctly to coupled material properties within

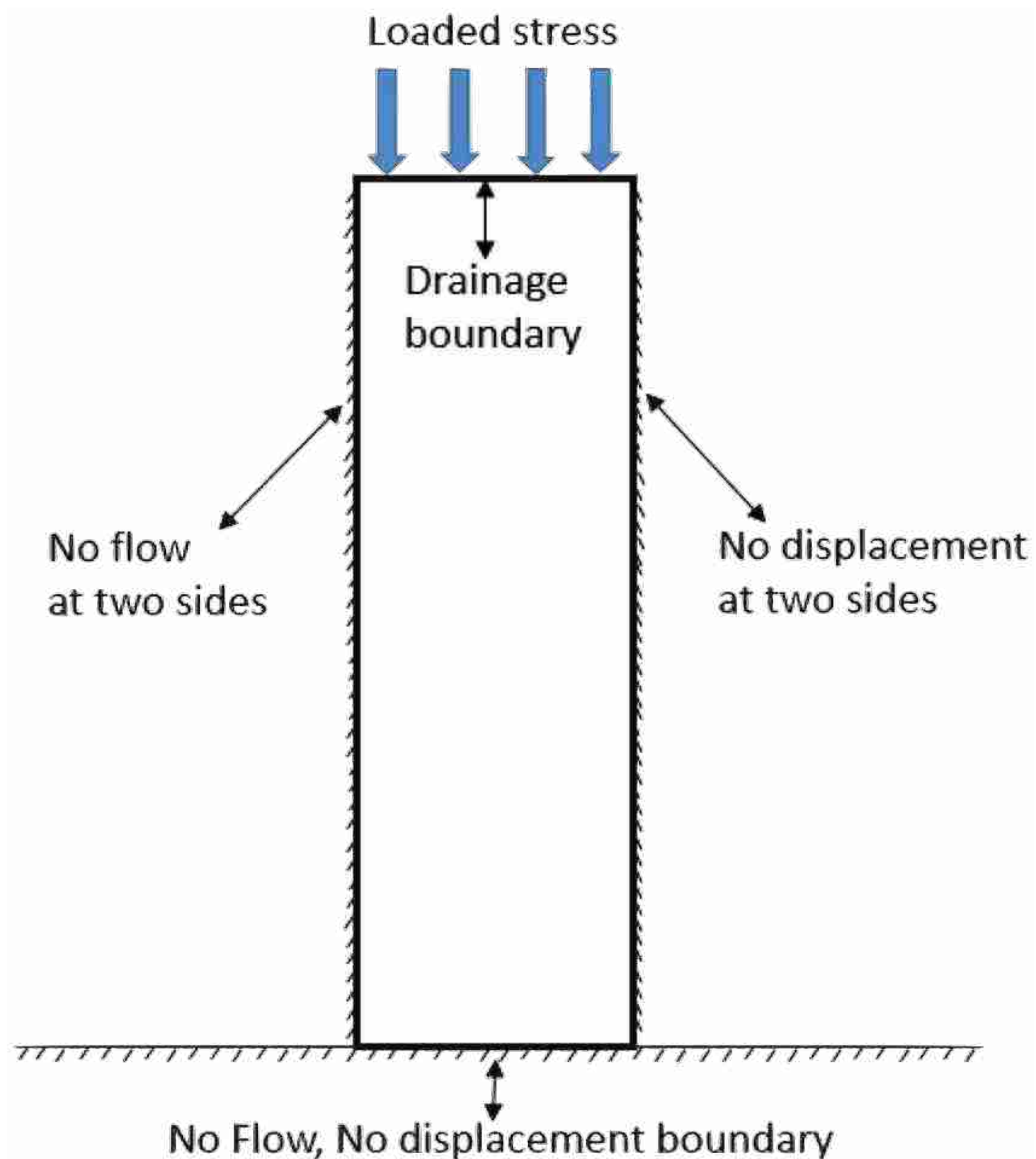


Figure 20. Terzaghi's problem of consolidation.

derivative parsed materials. Problems from the multiphysics domain involve physics forces namely, fluid displacement, fluid mobility, pore pressure, consolidation degree, poromechanics coupling, and stress divergence tensors. A one-dimensional problem would comprise investigating pressure pulse in one dimension with two

phases: steady and transient. A one-dimensional problem is not necessarily easier to solve than a higher dimensional problem. It rather depends on the physics of the problem that makes it easy or hard in comparison to other problems. Multiphysics problems, for instance, are harder than single physics problems. The other modules that the problem set belongs to are as follows:

1. Chemical reaction: The process that involves the rearrangement of the molecular or ionic structure of a substance is known as a chemical reaction. Examples of this module in MOOSE includes problems involving desorption and adsorption of fluids between a material and its porespace. These are general advection-dispersion-reaction equations.

2. Tensor mechanics: This module involves tensor equations of materials. A material changes its shape due to stress. The change in shape when compared with the original shape is called deformation or displacement. The ratio of deformation to original shape is called strain. To determine the deformed shape and the stress, an equation is solved to evaluate the displacement vector. Therefore the three tensors that are involved in a MOOSE tensor mechanics problem are elasticity tensor, strain, and stress.

3. Multiphase flow through porous media: This module involves highly diverse phenomenon. The module consists of problems that contain fluid flow through porous materials equations which involve fluid density, porosity, and fluid pressure. For example, a problem that belongs to this category ranges from the motion of immiscible fluids, through interaction with the medium through the heat exchange.

7.3 Classification Model

This section briefly describes the features that are computed for training the machine-learning model. The set of features has not been used previously for classifying linear solvers. Unlike Anamod, which is no longer updated or maintained, these features are leveraging the newest PETSc capabilities, reducing the feature computation overhead. Moreover, the new implementation was validated manually or by using MATLAB to ensure correctness. Section 3.10 describes the features that are used in the reduced feature sets throughout the thesis.

7.3.1 Feature Selection. In applications with dynamic mesh adaptation, such as the domain we consider, the structure of the mesh changes at run-time and therefore the best solver method may depend on the physical and geometric properties of the mesh. The dependency of a solver performance on the properties of the mesh becomes the basis of the types of features we compute for these problems. The full feature set contains 32 features which belong to different categories namely, simple or norm-like quantities, variability and structural. As using all the features of a linear system makes it expensive to solve an incoming linear system, the computation cost of each feature adds to the overall cost of the system. The computation time of each feature varies depending on its category, as the cost of computing certain features is relatively higher than others.

7.3.2 Feature Reduction. For reducing the overall cost, we reduce the number of features to participate in the classification process. Random selection of features may remove significant features. Therefore, we need a way to choose features that are significant and contribute the most to the classification process. First, we reduce the number of features by using Weka's RemoveUseless filter.

The filter removes the features for which the values are either constant or vary too much, having more than 99% variance. With this filter, we could remove two features out of the total 34 features, with 0% drop in accuracy.

We complete the selection with Weka by combining five attribute evaluators with two search methods. An evaluator has a mechanism to assign a weight to each subset of features. The search method determines what style of search is to be performed. These evaluators rank the features, giving a list of the features with ranking based on the algorithm that the evaluator follows. Using multiple evaluators ensure the selection of those features which are highly ranked by all or majority of the evaluators. The evaluators we use [56] are Gain Ratio, ChiSquared, CfsSubset, Information Gain, and Principle Component Analysis and the search methods we chose were Greedy Stepwise and Ranker. We generated two reduced feature sets for PETSc solvers, one of which is a subset of the other set. The top best features are most likely to remain the same with other problems from the same domain we considered an extensive set of data and high accuracy supports this statement.

7.3.3 Solvers and Preconditioners. PETSc has a collection of parallel algorithms for direct solvers, Krylov iterative methods, and preconditioners that can be used in application codes written in C, C++, Fortran and Python. We focus primarily on iterative Krylov methods and preconditioners to solve the sparse linear systems obtained from the MOOSE matrices. A total of 154 preconditioner-solver configurations were chosen from the options provided by PETSc (methods marked with asterisk in Table 3), with a unit right-hand-side vector. For the ILU and ICC preconditioners, the fill parameter is varied between 0 and 3 and for ASM the overlap is varied between subdomains parameter between 0 and 3.

As mentioned before, preconditioning is done to mainly make a linear system more suitable to be solved by a numerical solver method. The main idea behind applying a preconditioner is that, instead of solving $Ax = b$, solve $M^{-1}Ax = M^{-1}b$ using a non-singular matrix preconditioner M, which has the same solution as x.

7.3.4 Solver Classification. In machine learning, classification is the technique of predicting the class of the new instance from a set of categories. The class prediction is made on the basis of the training dataset that is used to train the classifier. We classify solvers based on the features of a linear system and select the solver configuration that performs best on similar systems during our system training. We use Weka to compare the performance of several classification algorithms. As mentioned in the previous chapters, Weka allows us to choose different classifiers. In this work, we examine Bayesian networks, Alternating Decision Trees(ADT) (with 50 boosting iterations), K-nearest neighbor, Random Forests, J48 and Support Vector Machines(SVM).

7.4 Experiments

We collected solver performance data with PETSc version 3.5.3 on two supercomputers: a Blue Gene/Q at Argonne National Laboratory and the Aciss cluster at the University of Oregon, which has nodes containing 2 hex-core 2.66GHz Intel Westmere (X5650) processors and 72 GB of RAM. Each experiment used a single node. We performed binary labeling('good' and 'bad' labels) for the PETSc solvers. We decide whether a solver is 'good' or 'bad' for a given problem based on the time the solver takes to solve the system in comparison with the fastest solver for that problem. We choose a threshold value of b by varying it from a range of 0.0 - 0.5. For our experiments, we chose the value of b as 0.3, based on trying different

values and then choosing the one which performed the best. In order to assign a label to the solver for a problem, if the solve time of the solver in consideration is less than solve time of the best solver multiplied by the b value it is labeled as 'good', otherwise labeled as 'bad'. In other words:

$$\text{If solve-time} \leq 1.30 * b, \text{ then label: 'good'}$$
$$\text{Else label: 'bad'}$$

Predicting the best solver for any given problem is not possible by using a purely analytical approach. Hence, we adopted the empirical approach described here. The accuracy of the classifier is determined by measuring true positives (TP) and false negatives (FN). We focus on the true positive rate (TPR) because the goal is to identify solution methods that are likely to perform well. Therefore, the accuracy measures presented in the next Section are computed using the usual true positive rate formula which can be defined as:

$TPR = TP/P = TP/(TP + FN)$

Here P is the actual number of positive instances, i.e., solvers labeled as 'good'. Some of the solvers have substantially more data points than others because of the random selection method we used for the solvers. In order to balance the amount of data for different solvers in the complete dataset, some of the data points (i.e., the solvers for which fewer than 10 timing results are available) are removed. The operation of removing such datapoints, leaves us with a total of $30,151$ datapoints, out of which $2,026$ datapoints are "good" and $28,125$ are labeled as "bad". The data is further split into training and test subsets in the two types of the validation described next. We used 10-fold cross-validation and 66%-34% train-test split for validating the classification results of the dataset. We used various machine-learning algorithms to do the classification which is described in

the previous section and chose the one which was the most accurate in making the solver predictions based on the accuracy.

7.4.1 Data Collection. Our dataset consists of problems from a single domain and is generated using the sample applications in MOOSE [46]. We instrumented the MOOSE code to save the matrices (non-linear systems) that are being solved with KSP solvers in PETSc. The MOOSE matrices we have are very small, with only a few matrices with more than thousand non-zeroes. In most cases, the default solver converges in a single iteration. For expanding our dataset, we varied the size of the MOOSE example problems and auto-generated bigger problems. We enlarged the mesh and executed in parallel to produce more realistic use cases. The number of processors and threads for these meshes are 1 and 12 respectively. As a result, we have up to three-dimensional meshes, with up to $643,204$ rows and columns and $23,232,048$ number of non-zeros. The larger matrices we use, have been generated by scaling the original 157 matrices with a factor of $1,000$. The minimum number of non-zero elements is set to $10,000$ to ensure the matrices are more realistic. MOOSE input files are hierarchical, block-structured files with a customizable syntax. Each block can have any number of name-value pairs. A simple problem uses around six blocks. The basic blocks of a MOOSE file are:

1. Mesh: The mesh block has information about the block, like dimensions, number of elements in the X, Y, Z direction. The mesh has two categories, file mesh, and generated mesh. File mesh can read any normal mesh format from a file. Generated mesh type can read the automatically generated mesh files. For our experiments, all files had generated mesh type.

2. Variables: The variables block declares the variables that will be solved along with their meta data. The meta data includes the order and family of the variables.
3. Kernels: The kernels block declares the operators that will be operated on the variables along with the type and variable name.
4. Boundary condition: The boundary condition block declares the boundary conditions that will be used in the simulation and the type of the condition.
5. Executioner: The executioner block declares the executioner for the simulation and the type of the executioner.
6. Output: Lastly, the output block declares the various output styles, like for file and console.

Table 27. Dimensions for original MOOSE matrices.

Matrix name	Dimensionality
AC_mobility_derivative_test, bl20, bl20_lumped, bl20_lumped_fu, cinterfaceposition_test, direct, direct_order4_test, direct_order6_test, direct_order8_test, direct_temp, gh01, gh02, gh03, gh04, gh05, gh06, gh07, gh08, gh09, gh10, gh11, gh12, gh13, gh14, gh15, gh16, gh17, gh18, gh20, gh21, gh22, gh23, gh_fu_01, gh_fu_02, gh_fu_03, gh_fu_04, gh_fu_05, gh_fu_06, gh_fu_09, gh_fu_10, gh_fu_11, gh_fu_12, gh_fu_17, gh_fu_18, gh_fu_20, gh_fu_22, gh_lumped_07, gh_lumped_08, gh_lumped_17, gh_lumped_18, langmuir_desorption, mass_lumping, mass_lumping_jacobian, mollified_langmuir_desorption, pp, pp01, pp02, pp21, pp22, pp_fu_01, pp_fu_02, pp_fu_21, pp_fu_22, pp_fu_lumped_22, pp_lumped_02, pp_lumped_22, split, split_order4_test, split_order6_test, split_order8_test, split_temp	One

Table 27 – continued from previous page

Matrix name	Dimensionality
AC_mobility_derivative_coupled_test, applied_strain_test, bw02, bw_lumped_02, CH_BndingBoxIC_test, CH_CircleIC_test, CH_CrossIC_test, CH_RndBndingBoxIC_test, CH_RndCircleIC_test, CHParsed_test, ConstructionOrder, DerivativeSumMaterial, derivativetwophasematerial, diffusion, GBEvolution_mob_test, GBEvolution_test, gradientcomponent, GrGr_boundingbox_test, GrGr_OffDiag_test, GrGr_particle_test, GrGr_test, GrGr_test_explicit, GrGr_thumb_test, GrGr_wTGrad_test, kobayashi, latticesmoothcircleIC_small_invalue_test, matdiffusion, material, MathEBFreeEnergy_test, MathFreeEnergy_test, matproptest, mobility_derivative_direct_coupled_test, mobility_derivative_direct_test, mobility_derivative_split_coupled_test, mobility_derivative_test, multiphasestress, nonsplit, nonsplit_gradderiv, nonsplit_gradderiv_action, rsc02, rsc_fu_01, rsc_fu_02, s01, s02, s03, s04, s05, s_fu_01, s_fu_03, s_fu_04, split_math_test, SplitCHParsed_test, thermal_expansion_test, TotalFreeEnergy_2var_test, TotalFreeEnergy_test, twophasestress, variable, variable_finite, wli02	Two

Table 27 – continued from previous page

Matrix name	Dimensionality
bh02, bh03, bh04, bh05, bh_fu_02, bh_fu_03, bh_fu_04, bh_fu_05, bulk_modulus_shear_modulus_test, cosserat_shear, cosserat_tension, lambda_shear_modulus_test, mandel, material_tensor_on_line_test, pp_generation, pp_generation_unconfined, pp_generation_unconfined_action, SmoothCircleIC_3D_test, ss, st01, terzaghi, unconsolidated_undrained, undrained_oedometer, uni_axial1_small_strain, vol_expansion, vol_expansion_action, youngs_modulus_poissons_ratio_test	Three

7.5 Results

Table 28 shows the set of features we chose as reduced feature sets. The first reduced feature set (RS1), consists of 6 features and the second reduced feature set (RS2) has only 4 features. RS2 is a subset of RS1 which is generated by further reducing the first reduced set. Table 29 summarizes the most-frequently used solvers and preconditioners along with their configurations that were most likely to perform well among all the configurations we tested during our experiments. The first column of the table shows their occurrence frequency.

7.5.1 Construction timing of each classifier. Table 30 shows the time that was taken to build each classifier. Most of the classifiers chosen are relatively fast except LibSVM, which takes substantially more time as compared to

Table 28. Reduced feature set for MOOSE dataset.

Feature name	Reduced Feature Set 1 ($RS1$)	Reduced Feature Set 2 ($RS2$)
RowVariance	X	X
AntiSymmetricFrobeniusNorm	X	
InfinityNorm	X	X
AvgNNZperRow	X	X
ColLogValSpread	X	
Symmetric	X	X

Table 29. Top 10 good solvers for PETSc with their occurrence percentage in the dataset.

Occurrence	Krylov Method	Preconditioner
10.56%	FGMRES	ICC(3)
10.56%	FGMRES	ICC(0)
10.51%	FGMRES	ICC(1)
5.75%	FGMRES	ILU(1)
5.75%	FGMRES	ILU(0)
5.75%	FGMRES	ASM(1)
5.75%	FGMRES	ASM(0)
5.75%	FGMRES	ASM(3)
5.75%	FGMRES	ASM(2)
5.70%	FGMRES	ILU(3)

Table 30. Time taken (in seconds) for constructing each classifier method using all features T_{All} and two different reduced feature sets T_{RS1} and T_{RS2}.

Method	T_{All}	T_{RS1}	T_{RS2}
RF	20.02	9.33	7.73
BayesNet	0.49	0.15	0.16
knn	0.001	0.001	0.001
ADT	6.95	1.02	0.8
J48	0.91	0.16	0.12

Table 31. Prediction accuracy of each classifier method using full feature set

Method	10 CV		Train-test 66-34% split	
	Overall Accuracy	"Good" Solver accuracy (TPR)	Overall Accuracy	"Good" Solver accuracy (TPR)
RF	99.6	97.6	99.7	97.8
BayesNet	72.8	64.7	75.1	64.4
knn(k=10)	99.2	92.6	99.2	94.3
ADT	96.4	52.6	96.4	52.3
J48	99.7	99.3	99.7	99.9

other classifiers. The reason we prefer to exclude it is because our results indicate that most of the time other classifiers superseded this classifier.

7.5.2 Prediction accuracy of each classifier. For 10-fold cross-validation (10CV) of the classification, using the full feature set, containing all the 32 features computed by PETSc, the J48 classifier had the best true positive rate (TPR) of 99.2% (Table 31). The best TPR of 99.4% was delivered again by J48 when we redid the classification with the 6 features of Reduced Feature Set 1 (RS1) shown in the Table 32. The reduced sets are shown in the Table 28. The best accuracy for Reduced Set 2 with 4 features was delivered by J48 classifier with an accuracy of 99.5%. For Reduced Set 2, LibSVM also had an accuracy of more than 99%. However, we do not consider this accuracy as it appears that LibSVM

Table 32. Prediction accuracy of each classifier method using Reduced Set 1 features

Method	10CV		Train-test 66-34% split	
	Overall Accuracy	"Good" Solver accuracy (TPR)	Overall Accuracy	"Good" Solver accuracy (TPR)
RF	99.6	97.1	99.7	98.1
BayesNet	95.9	65.1	95.7	65.3
knn(k=10)	99.6	97.5	99.7	98.1
ADT	96.4	52.6	96.4	52.3
J48	99.7	99.5	99.7	99.9

Table 33. Prediction accuracy of each classifier method using Reduced Set 2 features

Method	10CV		Train-test 66-34% split	
	Overall Accuracy	"Good" Solver accuracy (TPR)	Overall Accuracy	"Good" Solver accuracy (TPR)
RF	99.6	97.8	99.7	98.0
BayesNet	95.8	62.8	95.8	62.9
knn(k=10)	99.6	99.5	99.5	99.9
ADT	96.9	60.9	97.0	63.8
J48	99.7	99.5	99.7	99.9

Table 34. Validation with 10-fold cross-validation train-test data split for the dataset with the best classifier J48.

Class Labels	All Features		Reduced Feature Set 1		Reduced Feature Set 2	
Predicted Label	good	bad	good	bad	good	bad
True label (good)	2011	15	2016	10	2015	11
True label (bad)	62	28063	62	28063	59	28066

classifier classifies most of the data as a single class, i.e. either all of them are classified as “good” or all of them are classified as “bad”, which makes the reliability of this classifier questionable. The confusion matrix for the best classifiers in all cases can be seen in 34.

For 66-34% train-test split classification, using all 32 features computed by PETSc, J48 classifier had the best true positive rate (TPR) of 99.9% (Table 31). The best TPR of again 99.9% was delivered by J48 when we redid the classification with the 6 features in Reduced Feature Set 1 (RS1)shown in the Table 28. The best accuracy for Reduced Set 2 with 4 features was delivered by Random Forests classifier with an accuracy of 99.9%. The confusion matrix for the best classifier can be seen in 35

7.6 Summary

The matrix features presented in this chapter are computed by using PETSc instead of Anamod, which was used for the initial work presented in Section 3.12 and by many other researchers. Firstly, Anamod computes each feature individually, which results in high computation time for computing a large number of features. Secondly, using Anamod creates a dependency on an

Table 35. Validation with 66%-34% train-test data split for the dataset with the best classifier J48.

Class Labels	All Features		Reduced Feature Set 1		Reduced Feature Set 2	
Predicted Label	good	bad	good	bad	good	bad
True label (good)	684	1	684	1	684	1
True label (bad)	25	9541	21	9545	25	9541

external library. For both these reasons, we eliminated using Anamod for this work. The results indicate the benefits of the application of machine-learning based classification model on a domain-specific dataset. This chapter illustrates that high accuracy can be achieved in single domain problems due to the fact that problems from the same domain are highly similar, thus causing the classification model to train well on problems from the domain. The classification accuracy obtained in this work indicates promising results for problems coming from any single domain.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1 Conclusion

In many applications, the solution of large sparse linear systems is a key computation whose performance dominates the overall solution time. For example, applications that solve nonlinear partial differential equations through numerical approximations such as the Newton-Krylov family of methods, spend most of their time in the iterative linear system solution, which can be performed by any of the many preconditioned Krylov methods available. While they are functionally equivalent and have the same asymptotic complexity, their performance (how fast a solution is found) varies greatly depending on the characteristics of the input linear system. The proliferation of available solution methods makes the task of selecting a specific algorithm that performs well extremely challenging.

For solving large sparse linear systems, solvers are used in combination with preconditioners. There is a variety of solvers and preconditioners that are offered by different libraries, for instance PETSc alone offers more than three hundred pairings of Krylov methods and preconditioners shown in Table 3. Many solvers and preconditioners have parameters that can be varied and as a result, the number of possible solver-preconditioner combinations further increase. In addition, with the advancement of time, more solver techniques and preconditioners are getting developed to support more complex problems than ever before. Although the addition of new solving techniques enhances the power of the numerical libraries to handle complex problems better, it also grows the perplexity for a user to choose which combination of solver and preconditioner with which configuration, should be used for a given problem. The drawback of having numerous options is that

it becomes extremely hard for the user to make a sound choice as the decision requires expertise in high-performance computing, numerical analysis and domain knowledge. The challenge is where a recommendation system becomes a relief. These problems promote the need for a system, which reduces the task of the user to simply provide his problem as input and get a result from the system, a recommendation of what solver and preconditioner with which parameter configurations is the most suitable, depending on the characteristics of her problem.

This thesis explores that the performance of different solver techniques at small scales can be modeled using a small number of features based on structural and numerical properties of the input linear system. The results for the SuiteSparse dataset demonstrate that an efficient model can be built, despite the small size of the dataset. The top ranking features include average diagonal distance, number of nonzeros, norm1, column variability, minimum number of nonzeros, row variability and number of diagonal nonzeros, and kappa.

The research presented in this thesis illustrates the application of machine learning for selecting a "well-performing" preconditioned solver technique built on the validation from computational fluid dynamics applications and multiphysics simulation. As a matter of fact, machine learning classification techniques can identify quasi-optimal solvers efficiently for new problems, based on the model learning from the training data. During the training, the model observes the convergence behavior of various Krylov methods and preconditioners and utilizes that information for making new predictions. Since the solution time of the linear systems often prevails the overall scientific simulation time, reducing the solve time can significantly transform the use of machine learning for making such non-trivial decisions.

Although machine learning facilitates the predictions of quasi-optimal solver-preconditioner pairs, it is considerably dependant on the training data for learning the behavior of the solver techniques and preconditioners on different linear systems. Further, there is a training cost associated with the application of machine-learning, and using an ML-based approach for solver recommendations on large scale problems may incur an inordinate cost, which maybe prohibitively large and undesirable. Consequently, we present the communication-based scalability model that captures the parallel overhead of solving large sparse systems by quantifying the differences in inter-process communication for all the Krylov methods inspected. The preconditioned Krylov methods are ranked by computing and comparing the number of matrix-vector operations that perform communication, across the Krylov methods.

We combine the application of the convergence model and the scalability model in conjunction, to enable solver recommendations at different scales of parallelism. The model-based ranking is validated by comparison with empirical results on a numerical simulation of driven fluid flow in a cavity. This dissertation shows that the scalability model captures the communication overhead sufficiently well and can be used in combination with the machine-learning model. In general, this research shows the comparison of the performance achieved by different machine learning classification techniques for solver selection. As discussed, there are numerous solver techniques, numerical libraries, so are the number of machine learning algorithms that can be deployed. Therefore, exploring other solver techniques, numerical libraries and machine learning algorithms has immense potential in supporting the application of machine learning for solver selection further.

The conventional way of computing features involves storing a entire matrix explicitly. Our approach is rather different than the traditional way of computing features where we use a small sample of the matrix columns to get informed estimates of overall matrix features. Earlier, by using an external library like Anamod, which was used for our prior work and by many other researchers. To reduce the overall cost further, we present the sampling-based feature extraction approach for Krylov method selection, by using the matrix features using estimates of matrix properties. These estimates are based on matrix-free approximation for inexpensive, simple and structural features. The results indicate that the machine-learning based classification technique, which uses the matrix features estimates, can be used for suggesting solver techniques for problems from different domains and a domain-specific use case (MOOSE). The results point towards promising results for problems arising from any single domain or from a collection of variety of domains.

For reducing the overall cost of the system, we reduce the feature set by eliminating features that negatively impact the learning process and the features that do not contribute towards the model. Although we drastically reduce the number of total features to be computed for a new system, to a handful of comparatively inexpensive features, choosing a good set of features is one of the most relevant tasks while constructing a classifier. The feature set tends to change for different datasets, hence exploring an automated feature selection strategy can be extremely useful in overcoming this deficiency.

To conclude, this dissertation demonstrates that it is possible to select preconditioned solvers techniques, based on the convergence behavior and the parallel overhead of preconditioned Krylov methods. The machine-learning model

captures the convergence behavior of solvers and preconditioners. For modeling the parallel overhead, the analytical approach generates a scalability-based ranking for the preconditioned Krylov methods. Combining the convergence model with the communication model, enables more accurate solver recommendations at different parallelism scales. On the whole, this dissertation contributes towards the advancement of new modeling techniques for application from a wide variety of domains.

8.2 Future Work

In the future, this research presented can be expanded in several directions. So far, most of the linear systems are used for the SuiteSparse matrix collection and the MOOSE application. Although the SuiteSparse collection includes problems from a variety of domains, and the matrix sizes are sufficiently large, the real-world problems may be comparatively larger. Therefore, one of the expansions of this work can involve using training matrices bigger than the matrices offered by SuiteSparse collection. Linear systems used in this research from the multiphysics-domain (MOOSE) are varied by enlarging the mesh size, testing applications from other domains that generate more realist use cases is another future aspect of this work.

While the convergence model is largely automated, the analytical ranking used for comparing the amount of communication across different preconditioned Krylov methods still involves manual effort. Currently for generating the solver ranking, each Krylov method and preconditioner is individually analyzed to identify the number of matrix-vector operations that perform communication. Therefore, another aspect of this work that can be explored in the future involves investigating techniques to automate the solver ranking when given specific input features.

Machine learning is a fast-evolving field. When new techniques are developed, they may outperform the existing methods at the time of this writing. Therefore, in the future, including the new techniques to verify and update the technique would be useful. Although this thesis explores using a variety of classification techniques, due to time constraints, Neural networks was mostly excluded from the convergence model, which is yet another aspect of this work that can be analyzed.

For the matrix-free feature selection, explicit matrices were used for training. For testing the modeling technique, matrix-free applications offered by PETSc were used. Due to time restraints, building the training set on matrix-free applications was eliminated. However, a strong recommendation to enhance the sampling-based approach is to train on matrix-free applications and test on matrix-free applications. Therefore, it is important to research the matrix-free approach with a sufficiently large training dataset completely composed of matrix-free applications.

.3 Appendix

1. Diagonal matrix: A matrix in which the non-diagonal elements are 0. For instance, the matrix below is diagonally dominant, as the only non-zero elements are present at the diagonal locations.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & -6 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

2. Diagonally dominant matrix: A matrix is diagonally dominant if for each row, the magnitude of the diagonal entry in that row is larger than or equal to the sum of the magnitudes of all the other non-diagonal entries in that row.

3. Order of a matrix: The number of rows and columns of a matrix are referred to as the order of the matrix. For instance, a matrix with 4 rows and 5 columns has an order of $4x5$.

4. Singular matrix: A matrix whose determinant is zero. For instance, the matrix given below has a determinant 0, which makes it singular matrix.

$$\begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix}$$

The determinant of the matrix can be given by the Laplace formula:

$$determinant(A) = \sum_{\sigma \in S_n} -1^{N(\sigma)} \prod_{i=1}^{n} a_{i,\sigma_i}.$$

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma_i}.$$

5. Triangular matrix: A square matrix with the following special characteristics: Lower triangular matrix: The square matrix in which all the elements above the diagonal are zero. Upper triangular matrix: The square matrix in which all the elements below the diagonal are zero.

6. Square matrix: A matrix with the same number of rows and columns.

7. Symmetric matrix: A square matrix that is equal to its transpose, so $A = A^T$. An example of a symmetric matrix is shown below:

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & -3 & 8 \\ 4 & 8 & -9 \end{bmatrix}$$

8. Factorization: Original matrix is decomposed into multiple smaller matrices. The original matrix can be obtained by the product of the smaller matrices.

9. Ill conditioned matrices: Matrices, which have a very large condition number, are called as Ill conditioned matrices. Matrices with a small condition number are referred to as well-conditioned matrices.

10. Bidiagonalization: The process of converting a matrix into a bidiagonal matrix, which is, a matrix in which the non-zero elements are along the main diagonal of the matrix and either the diagonal below or above the main diagonal. For example, the matrix shown below is bidiagonal.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 0 & 1 & -6 & 0 \\ 0 & 0 & 3 & 2 \end{bmatrix}$$

11. Tridiagonal matrix: A matrix in which the non-zero elements are along the main diagonal of the matrix and along the diagonal below and above the main diagonal. For example, the matrix shown below:

$$\begin{bmatrix} 1 & 5 & 0 & 0 \\ 2 & 4 & 4 & 0 \\ 0 & 1 & -6 & 1 \\ 0 & 0 & 3 & 2 \end{bmatrix}$$

12. Orthogonalization: The process of finding a set of orthogonal vectors in a given subspace.

13. Symmetric positive definite matrix: A matrix is symmetric positive definite if $A = A^T$, A^{-1} exists, all its Eigenvalues are positive and all elements of A are greater than zero.

REFERENCES CITED

[1] Enrique Alba and José M Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[2] Ed Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy DuCroz, Anne Greenbaum, Sven Hammarling, A McKenney, et al. Lapack usersâĂŹ guide. society for industrial and applied mathematics, philadelphia, pa. Technical report, ISBN 0-89871-447-8 (paperback), 1999.

[3] Allison H Baker, Elizabeth R Jessup, and Thomas Manteuffel. A technique for accelerating the convergence of restarted gmres. *SIAM Journal on Matrix Analysis and Applications*, 26(4):962–984, 2005.

[4] Satish Balay, Kris Buschelman, Victor Eijkhout, William D Gropp, Dinesh Kaushik, Matthew G Knepley, Lois Curfman McInnes, Barry F Smith, and Hong Zhang. Petsc users manual. Technical report, Technical Report ANL-95/11-Revision 2.1. 5, Argonne National Laboratory, 2004.

[5] Richard Barrett, Michael Berry, Jack Dongarra, Victor Eijkhout, and Charles Romine. Algorithmic bombardment for the iterative solution of linear systems: a poly-iterative approach. *Journal of Computational and applied Mathematics*, 74(1):91–109, 1996.

[6] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[7] Jacob Berlin and Amihai Motro. Database schema matching using machine learning with feature selection. In *International Conference on Advanced Information Systems Engineering*, pages 452–466. Springer, 2002.

[8] S Bhowmick, L McInnes, B Norris, and P Raghavan. Robust algorithms and software for parallel pde-based simulations. In *Proceedings of the Advanced Simulation Technologies Conference, ASTC*, volume 4, pages 18–22, 2004.

[9] Sanjukta Bhowmick. *Multimethod solvers: algorithms, applications and software*. Pennsylvania State University, 2005.

[10] Sanjukta Bhowmick, D Kaushik, L McInnes, B Norris, and P Raghavan. Parallel adaptive solvers in compressible petsc-fun3d simulations. In *Proceedings of the 17th International Conference on Parallel CFD*. Elsevier, 2005.

[11] Sanjukta Bhowmick, L McInnes, Boyana Norris, and Padma Raghavan. The role of multi-method linear solvers in pde-based simulations. In *Computational Science and Its ApplicationsâĂŤICCSA 2003*, pages 828–839. Springer, 2003.

[12] Sanjukta Bhowmick, Padma Raghavan, L McInnes, and Boyana Norris. Faster pde-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20(3):373–387, 2004.

[13] Sanjukta Bhowmick, Padma Raghavan, and Keita Teranishi. A combinatorial scheme for developing efficient composite solvers. In *ComputationalScienceâĂŤICCS 2002*, pages 325–334. Springer, 2002.

[14] Concha Bielza and Pedro Larrañaga. Discrete Bayesian Network classifiers: A survey. *ACM Comput. Surv.*, 47(1):5:1–5:43, July 2014.

[15] Randall Bramley, Dennis Gannon, Thomas Stuckey, Juan Villacis, Esra Akman, Jayashree Balasubramanian, Fabian Breg, Shridhar Diwan, and Madhusudhan Govindaraju. The linear system analyzer. Technical report, Technical Report TR511, Indiana University, 1998.

[16] Randall Bramley and Xiaoge Wang. Splib: A library of iterative methods for sparse linear systems. *URL address: http://www. elk. itu. edu. tr/dag/lssmc. html*, 1995.

[17] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001.

[18] Xiao-Chuan Cai, David E Keyes, and Leszek Marcinkowski. Non-linear additive schwarz preconditioners and application in computational fluid dynamics. *International journal for numerical methods in fluids*, 40(12):1463–1470, 2002.

[19] Tony F Chan and Henk A Van Der Vorst. Approximate and incomplete factorizations. In *Parallel numerical algorithms*, pages 167–202. Springer, 1997.

[20] Gregory F Cooper and Edward Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine learning*, 9(4):309–347, 1992.

[21] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[22] Padraig Cunningham and Sarah Jane Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, pages 1–17, 2007.

[23] Timothy A Davis. *Direct methods for sparse linear systems*, volume 2. Siam, 2006.

[24] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[25] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R Clint Whaley, and Katherine Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[26] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *International Conference on Computational Science*, pages 759–767. Springer, 2003.

[27] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–131, 2003.

[28] JJ Dongarra, CB Moler, JR Bunch, and GW Stewart. Linpack userâĂŹs guide: Society for industrial and applied mathematics. *Philadelphia, Pennsylvania*, 1979.

[29] Maksymilian Dryja and Olof Widlund. *An additive variant of the Schwarz alternating method for the case of many subregions*. Ultracomputer Research Laboratory, Univ., Courant Inst. of Mathematical âĂę, 1987.

[30] Iain S Duff, Albert Maurice Erisman, and John Ker Reid. *Direct methods for sparse matrices*. Clarendon press Oxford, 1986.

[31] Todd Dupont, Richard P Kendall, and HH Rachford, Jr. An approximate factorization procedure for solving self-adjoint elliptic difference equations. *SIAM Journal on Numerical Analysis*, 5(3):559–573, 1968.

[32] V Eijkhout and E Fuentes. Anamod online documentation, 2007.

[33] Victor Eijkhout and Erika Fuentes. Multi-stage learning of linear algebra algorithms. In *Machine Learning and Applications, 2008. ICMLA'08. Seventh International Conference on*, pages 402–407. IEEE, 2008.

[34] Victor Eijkhout, Erika Fuentes, Naren Ramakrishnan, Pilsung Kang, Sanjukta Bhowmick, David Keyes, and Yoav Freund. A self-adapting system for linear solver selection. In *Proc. 1st intâĂŹl workshop on automatic performance tuning (iWAPT2006)*, pages 44–53, 2006.

[35] Stanley C Eisenstat, Howard C Elman, and Martin H Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 20(2):345–357, 1983.

[36] Jocelyne Erhel, Kevin Burrage, and Bert Pohl. Restarted gmres preconditioned by deflation. *Journal of computational and applied mathematics*, 69(2):303–318, 1996.

[37] Alexandre Ern, Vincent Giovangigli, David E Keyes, and Mitchell D Smooke. Towards polyalgorithmic linear system solvers for nonlinear elliptic problems. *SIAM Journal on Scientific Computing*, 15(3):681–703, 1994.

[38] Robert D Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[39] Charbel Farhat and Francois-Xavier Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32(6):1205–1227, 1991.

[40] Matthieu Ferrant, Simon K Warfield, Arya Nabavi, Ferenc A Jolesz, and Ron Kikinis. Registration of 3d intraoperative mr images of the brain using a finite element biomechanical model. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 19–28. Springer, 2000.

[41] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[42] Ian Foster, Carl Kesselman, and Steven Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[43] Roland W Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. *SIAM journal on scientific computing*, 14(2):470–482, 1993.

[44] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *icml*, volume 99, pages 124–133, 1999.

[45] D Fuentes, JT Oden, KR Diller, JD Hazle, A Elliott, A Shetty, and RJ Stafford. Computational modeling and real-time control of patient-specific laser treatment of cancer. *Annals of biomedical engineering*, 37(4):763–782, 2009.

[46] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandie. Moose: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design*, 239(10):1768–1778, 2009.

[47] Gene H Golub and Richard S Varga. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order richardson iterative methods. *Numerische Mathematik*, 3(1):157–168, 1961.

[48] A Hadjidimos. Successive overrelaxation (sor) and related methods. *Journal of Computational and Applied Mathematics*, 123(1-2):177–199, 2000.

[49] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[50] Magnus Rudolph Hestenes and Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS, 1952.

[51] Elizabeth Jessup, Pate Motter, Boyana Norris, and Kanika Sood. Performance-based numerical solver selection in the lighthouse framework. *SIAM Journal on Scientific Computing*, 38(5):S750–S771, 2016.

[52] Elizabeth Jessup, Pate Motter, Boyana Norris, and Kanika Sood. Performance-based numerical solver selection in the lighthouse framework. *SIAM Journal on Scientific Computing*, 38(5):S750–S771, 2016.

[53] Richard F Katz, Marc Spiegelman, and Benjamin Holtzman. The dynamics of melt and shear localization in partially molten aggregates. *Nature*, 442(7103):676, 2006.

[54] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: A survey of approaches and applications. *J. Comput. Phys.*, 193(2):357–397, January 2004.

[55] Chao Li and Raj Mittra. A preconditioning technique based on singular value decomposition and its comparision with the characteristic basis function method. In *2016 USNC-URSI Radio Science Meeting*, pages 55–56. IEEE, 2016.

[56] Tao Li, Chengliang Zhang, and Mitsunori Ogihara. A comparative study of feature selection and multiclass classification methods for tissue classification based on gene expression. *Bioinformatics*, 20(15):2429–2437, 2004.

[57] X Sherry Li, J Demmel, J Gilbert, L Grigori, M Shao, and I Yamazaki. Superlu numerical software library. *URL http://crd-legacy.lbl.gov/xiaoye/SuperLU*.

[58] L McInnes, B Norris, S Bhowmick, and P Raghavan. Adaptive sparse linear solvers for implicit cfd using newton-krylov algorithms. In *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, volume 2, pages 1024–1028, 2003.

[59] Richard Tran Mills, Glenn E Hammond, Peter C Lichtner, Vamsi Sripathi, G Kumar Mahinthakumar, and Barry F Smith. Modeling subsurface reactive flows using leadership-class computing. In *Journal of Physics: Conference Series*, volume 180, page 012062. IOP Publishing, 2009.

[60] Keiichi Morikuni, Lothar Reichel, and Ken Hayami. Fgmres for linear discrete ill-posed problems. *Applied Numerical Mathematics*, 75:175–187, 2014.

[61] Boyana Norris, Sa-Lin Bernstein, Ramya Nair, and Elizabeth Jessup. Lighthouse: A user-centered web service for linear algebra software. *arXiv preprint arXiv:1408.1363*, 2014.

[62] Yvan Notay. Flexible conjugate gradients. *SIAM Journal on Scientific Computing*, 22(4):1444–1460, 2000.

[63] Carlos Ordonez. Comparing association rules and decision trees for disease prediction. In *Proceedings of the international workshop on Healthcare information and knowledge management*, pages 17–24. ACM, 2006.

[64] Christopher C Paige and Michael A Saunders. Algorithm 583: Lsqr: Sparse linear equations and least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 8(2):195–209, 1982.

[65] Christopher C Paige and Michael A Saunders. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM transactions on mathematical software*, 8(1):43–71, 1982.

[66] Su-lin Pang and Ji-zhang Gong. C5. 0 classification algorithm and application on individual credit evaluation of banks. *Systems Engineering-Theory & Practice*, 29(12):94–104, 2009.

[67] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[68] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[69] John R. Rice. A polyalgorithm for the automatic solution of nonlinear equations. pages 179–183, New York, NY, USA, 1969. ACM.

[70] John R Rice. A polyalgorithm for the automatic solution of nonlinear equations. In *Proceedings of the 1969 24th national conference*, pages 179–183. ACM, 1969.

[71] J.R. Rice. On the construction of poly-algorithms for automatic numerical analysis. In *Interactive Systems for Experimental Applied Mathematics. M. Klerer and J.Reinfelds*, pages 301–313. Academic Press, 1968.

[72] Lewis Fry Richardson. Ix. the approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 210(459-470):307–357, 1911.

[73] Lubomír Říha, Tomáš Brzobohatý, Alexandros Markopoulos, Ondřej Meca, and Tomáš Kozubek. Massively parallel hybrid total feti (htfeti) solver. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 7. ACM, 2016.

[74] Youcef Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

[75] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[76] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[77] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[78] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[79] Yousef Saad and Brian Suchomel. Arms: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical linear algebra with applications*, 9(5):359–378, 2002.

[80] Patrick Sanan, Sascha M Schnepp, and Dave A May. Pipelined, flexible krylov subspace methods. *SIAM Journal on Scientific Computing*, 38(5):C441–C470, 2016.

[81] Patrick Sanan, Sascha M Schnepp, and Dave A May. Pipelined, flexible krylov subspace methods. *SIAM Journal on Scientific Computing*, 38(5):C441–C470, 2016.

[82] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[83] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[84] Frédéric Sicot, Guillaume Puigt, and Marc Montagnac. Block-jacobi implicit algorithms for the time spectral method. *AIAA journal*, 46(12):3080–3089, 2008.

[85] Gerard LG Sleijpen and Henk A Van der Vorst. An overview of approaches for the stable computation of hybrid bicg methods. *Applied numerical mathematics*, 19(3):235–254, 1995.

[86] Kanika Sood, Boyana Norris, and Elizabeth Jessup. Lighthouse: A taxonomy-based solver selection tool. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems*, pages 66–70. ACM, 2015.

[87] Kanika Sood, Boyana Norris, and Elizabeth Jessup. 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)*, pages 26–33. IEEE, 2017.

[88] Andreas Stathopoulos and James R McCombs. Primme: preconditioned iterative multimethod eigensolverâĂŤmethods and software description. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):21, 2010.

[89] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[90] John Todd. *Survey of numerical analysis*. McGraw-Hill, 1962.

[91] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[92] Eli Turkel. Preconditioning techniques in computational fluid dynamics. *Annual Review of Fluid Mechanics*, 31(1):385–416, 1999.

[93] Henk A Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[94] Eric W Weisstein. Biconjugate gradient method. 2003.

[95] Charles A Williams, Brad Aagaard, and Matthew G Knepley. Development of software for studying earthquakes across multiple spatial and temporal scales by coupling quasi-static and dynamic simulations. In *AGU Fall Meeting Abstracts*, 2005.

[96] Nigel Williams and Sebastian Zander. Evaluating machine learning algorithms for automated network application identification. 2006.

[97] Nigel Williams and Sebastian Zander. Evaluating machine learning algorithms for automated network application identification. 2006.

[98] Xuemin Xu, Qing Huo Liu, and Zhong Wing Zhang. The stabilized biconjugate gradient fast fourier transform method for electromagnetic scattering. In *IEEE Antennas and Propagation Society International Symposium (IEEE Cat. No. 02CH37313)*, volume 2, pages 614–617. IEEE, 2002.

[99] JY Yuan, Gene H Golub, Robert J Plemmons, and WAG Cecilio. Semi-conjugate direction methods for real positive definite systems. *BIT Numerical Mathematics*, 44(1):189–207, 2004.