Hardware Awareness for the Selection of Optimal Iterative Linear Solvers

by

Pate Allen Motter

B.S., University of Arkansas, 2011

M.S., University of Colorado, Boulder, 2013

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2017

This thesis entitled:
Hardware Awareness for the Selection of Optimal Iterative Linear Solvers
written by Pate Allen Motter
has been approved for the Department of Computer Science

_____
Prof. Elizabeth Jessup

_____
Prof. Xiao-Chuan Cai

_____
Prof. Clayton Lewis

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Motter, Pate Allen (Ph.D., Computer Science)

Hardware Awareness for the Selection of Optimal Iterative Linear Solvers

Thesis directed by Prof. Elizabeth Jessup

Solving sparse systems of linear equations is a commonly encountered computation in scientific and high-performance computing applications. Applications that depend on solving sparse linear systems as part of their workflow can spend a large percentage of their total runtime solving sparse systems. However, selecting the best iterative solver and preconditioner for solving a given sparse linear system, especially for novice users, is not a simple task. To address this problem, previous works have used machine learning techniques to find similarities between sparse matrices and the corresponding performance that solver-preconditioner pairs have on solving the resulting linear systems. This dissertation expands on existing work by introducing techniques that incorporate hardware information into the prediction of ideal iterative linear solver and preconditioners for sparse linear systems. By accounting for hardware, it is possible to create more specially tailored solver-preconditioner recommendations for a novice user.

Acknowledgements

I am most grateful to my dissertation advisor, Elizabeth Jessup, for her guidance and support throughout the dissertation process.

Boyana Norris, who served as a co-advisor in all but name.

Ian Karlin, for teaching me how to be a better computer scientist.

Xiao-Chuan Cai and Clayton Lewis, for their belief and support of this work.

My students, for making me become a better teacher and listener.

The numerous PhD students, HPC researchers, REU students, and interns I have worked with and met over the years, for providing an incredibly vast amount of knowledge. My friends Danny Brill and Dylan McKinney for providing lighthearted relief, even in the most trying of times.

My parents, Mark and Kristi Ozbun, for always supporting me and my academic desires.

And most of all, to my wife Barry, for going through this journey with me and being a constant pillar of support, even while pursuing her own PhD.

I also could not have completed this work without the computer systems and support staff of the Extreme Science and Engineering Discovery Environment (XSEDE), supported by NSF Grant ACI-1548562, as well as the RMACC Summit supercomputer, a joint effort of the University of Colorado Boulder and Colorado State University, supported by NSF awards ACI-1532235 ACI-1532236.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Solving sparse systems of linear equations is a commonly encountered computation in scientific and high-performance computing applications. These linear systems are of the form $Ax = b$ where A is an $m \times n$ coefficient matrix, b is a known solution vector of dimension m, and x is an unknown vector of dimension n. Many of these linear systems are created as a result of discretizing partial differential equations (PDEs), a prevalent computation in applications such as the finite difference or finite element methods commonly used in the fields of computational fluid dynamics and structural analysis [1, 2]. Applications that depend on solving sparse linear systems as part of their workflow can spend a large percentage of their total runtime solving sparse systems [1]. Because solving linear systems is integral to many scientific and high-performance computing problems, solving a linear system faster or more efficiently can have a widespread impact in a variety of disciplines.

Choosing an iterative solver is not the only component required when choosing how to solve a sparse linear system. In addition to the solver, a preconditioner is often selected to improve the chosen solver's performance. Preconditioners convert the target linear system into a similar system, which has more appealing numerical properties that are better suited for being solved via iterative means. Nowadays it is assumed that when choosing an iterative solver, a preconditioner is also chosen. Therefore, now a user is no longer responsible for simply choosing a solver, but also a preconditioner. The added dimensionality caused by the addition of the preconditioner drastically increases the number of possible choices for solving a given linear system. Unfortunately, choosing the optimal solver-preconditioner pair for a linear system can be a difficult task not only

due to the number of applicable solvers and preconditioners for a given problem, but also because the performance of the solvers is not guaranteed **a priori**. A non-optimal choice of either the iterative solver or preconditioner can have a drastic negative impact on the overall runtime of the solve. In some cases, non-optimal choices result in a solution not being found, even though another solver-preconditioner pair may converge towards a solution. Due to the prevalence and importance of solving sparse linear systems in applications, improving the time-to-solution by choosing the optimal solver and preconditioner pair can have a significant and far-reaching impact.

However, choosing an optimal iterative solver-preconditioner combination for a given linear system is not always an easy or straightforward task [3]. Although there are ways to group or sort the available iterative solvers based on the types of matrices they support, it is often not enough since there are still multiple appropriate methods for each type of matrix. For instance, a square non-symmetric matrix can be solved using various iterative solvers including GMRES, TFQMR, and BiCGSTAB [1]. Depending on the numerical properties and structure of the non-symmetric matrix, any one of the previously mentioned solvers is capable of outperforming the others. Unfortunately, choosing a solver from several available solvers, which seem similar at face-value, can be difficult for novice users.

Previous work [4–7] has been performed using machine learning classification in order to help determine the best solvers or solver-preconditioner pairs for a given sparse linear system. These efforts find correlations between the layout and numerical properties of a coefficient matrix A and the resulting performance of various solvers and preconditioners associated with that linear system. The machine learning classification algorithms are trained using both the matrix attributes and the time it takes to converge to a solution for the available solver-preconditioner pairs. Once the classifier has been trained, it is then used to determine which solver-preconditioner pairs are optimal for finding a solution to certain types of linear systems. Once trained, the classifier is able to determine the best solver-preconditioner pairs for a newly presented linear system based on the classifier's training data by finding similarities between a new matrix and the collection of matrices that have been previously identified.

Previous work, using machine learning techniques to predict iterative solver performance, has collected the wallclock timing data of solver-preconditioner pairs on a given linear system using solver code running serially or across a single fixed-size CPU count. An optimal solver-preconditioner prediction for a fixed-size CPU count is not universally the best choice when running on a different number of cores than the specified fixed-size. Similarly, the system hardware being used for the linear solves can also be responsible for differences in solver-preconditioner performance. The best choice of solver and preconditioner differs when solving the same problem serially on a modest laptop or when solving that same problem in parallel across an entire cluster node. To demonstrate this, we measure the performance of solvers and preconditioners at multiple CPU counts and on different computer systems in order to create a more dynamic recommendation system based on a user's compute capabilities.

1.1 Related Work

There have been various efforts throughout the years to address the problem of selecting optimal numerical methods for a given problem type. Weerawarana et al. introduced PYTHIA [8], a system for solving elliptic PDE problems with the ELLPACK [9] library, based on user-specified variables and the type of problem being solved. The Self Adapting Numerical Software (SANS) [10] is another effort designed for the optimal selection of algorithms based on a user's data and hardware. As a part of SANS, the Automatically Tuned Linear Algebra Software (ATLAS) [11] uses empirical search techniques to create customized kernels of the basic linear algebra subprograms (BLAS) [12] tailored to the system's hardware.

There are also efforts with the specific goal of identifying optimal iterative linear solvers and/or preconditioners for arbitrary linear systems. Bhowmick et al. present results in multiple papers [4, 6, 13–15] that explore the use of machine learning techniques and their applications to solving linear systems and to numerical methods in general. Their approach uses a binary classification scheme to recommend solvers and preconditioners based on a training set of data containing matrix properties and the associated runtimes of those matrices with various solvers and

preconditioners. The solvers and preconditioners used throughout their work come from the PETSc library [16].

The Lighthouse project [5], of which the author is a contributor, is a joint effort between the University of Oregon and Colorado and has the overarching goal of creating an online taxonomy for numerical libraries. Lighthouse is capable of directing and aiding users in choosing the best numerical methods for a given task and problem. Users are able to enter information about their problem, the library they wish to use, and even their matrix into the system which then computes an appropriate solution for the given problem. Lighthouse uses a similar approach to that of [4,6,13–15] for training machine learning classifiers on matrix features and the runtime performances of various solvers and preconditioners. One of the key differences in Lighthouse is its incorporation of multiple numerical libraries and problem types. In addition to sparse linear systems, Lighthouse also provides optimal functions or methods for solving dense linear systems and eigen problems. Lighthouse also differs by providing support for various numerical libraries including PETSc, SLEPc [17], LAPACK [18], and Trilinos [19].

Jeom et al. have performed recent work [7] in determining the best solvers and preconditioners from the Trilinos numerical framework. Similar to Lighthouse, their work is also designed with the mindset of aiding novice users in the selection of solver-preconditioner pairs for arbitrary matrices. Their work largely differs from the two previously mentioned research efforts through their machine learning techniques and use of regression rather than classification. The machine learning method used in [7] is neural word embedding (NWE) [20], an algorithm designed primarily for natural language processing. They use a specific NWE model Word2Vec [21], a neural network capable of determining not only the correlations of input vectors, but also their relationship to each other. Word2Vec works by creating a many dimensional space with vectors in the space representing input data. Unlike other works by [5] and [4, 6, 13–15], which use binary classification to predict "good" and "bad" solver-preconditioner pairs, [7] predicts the numerical runtimes of solver-preconditioner pairs for a given linear system.

One of the key similarities across all of the aforementioned research is their collection of

wallclock timing data obtained using a fixed CPU core count on a single computer system. Both the work present in [7] and [4, 6, 13–15] obtain the runtimes on the solvers and preconditioners based on solving the linear systems serially on a single core. Work from the Lighthouse project [5] also obtains runtime data using a fixed core count with most experiments being performed serially; one experiment has been run on 12 cores. This dissertation expands upon these existing works by collecting the runtimes of solving linear systems across multiple core counts in a variable manner rather than fixed.

1.2 Overview and Outline of the Document

The new techniques presented in this dissertation allow novice users to easily choose the best combination of iterative linear solvers and preconditioners from existing numerical linear algebra libraries and applications to solve their specific problem on their specific hardware. Users such as domain scientists or other researchers lacking knowledge in numerical mathematics and hardware performance, can therefore avoid spending their time, energy, and resources on concerning themselves with incorrectly chosen or poorly performing solver-preconditioner choices by using the methods described in this dissertation. Thus, the described methods allow researchers to save time and important resources in solving their sparse linear systems.

The rest of the dissertation is organized as follows. Chapter 2 describes the numerical methods involved with iterative linear solvers and preconditioners, as well as the software used to implement them both. Chapter 3 introduces the key concepts associated with machine learning and the classification techniques used throughout the dissertation. Chapter 4 contains the core methodologies used throughout the dissertation including the training data and data collection schemes for collecting features from both the training matrices and computer systems. Chapter 5 describes the experiments and results associated with the performance prediction using different numbers of CPU cores. Chapter 6 describes the experiments and results associated with the performance prediction using different computer systems. Chapter 7 discusses the experiments and results obtained by combining the experiments of Chapters 5 and 6 into a single prediction problem.

Chapter 8 explores the applicability of the classifier created in Chapter 7 and applying it to the prediction of larger, real-world problems. Chapter 9 discusses the overall results of the work and possible directions for future work.

Chapter 2

Numerical Background

This chapter presents the core linear algebra techniques and numerical methods used throughout the rest of this dissertation. Section 2.1 provides an introduction to systems of linear equations and how they can be solved via direct and iterative methods. Section 2.2 describes preconditioners and their role in aiding iterative solvers solve systems of linear equations. Section 2.3 is an overview of the numerical library framework Trilinos, which provides the implementations of iterative solvers and preconditioners used throughout the numerical experiments in this dissertation.

2.1 Systems of Linear Equations

Systems of linear equations, or linear systems, are a commonly encountered mathematical problem found in a wide number of scientific and engineering related fields. Most commonly, linear systems are byproducts from the discretization of partial differential equations (PDEs) [1, p. 47]. These PDEs can occur from many sources, but are often associated with solving engineering or physics problems via the finite element and finite volume methods. However, it is also possible for linear systems to arise from many other non-PDE related problems such as circuit design [22] and medical imaging [23]. A linear system represents a set of m linear equations with n variables and is written as $Ax = b$ where A is the matrix of coefficients, x is the column vector of unknown variables,

and b is the column vector of solutions, as seen below.

$$Ax = b \quad \equiv \quad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \quad A \in \mathbb{C}^{m \times n}, \quad x \in \mathbb{C}^n, \quad b \in \mathbb{C}^m \qquad (2.1)$$

For a given linear system, the coefficient matrix A and the solution vector b are known, while the vector x is unknown and must be solved for. Computing the unknown vector x is non-trivial and has resulted in numerous mathematical methods designed to find the solution efficiently. In order for a linear system to be considered solved, the variable vector x must be computed such that all m equations of the system are simultaneously satisfied.

Solving systems of linear equations within an application can be responsible for a large percentage of the overall runtime of scientific applications [24]. Solving a linear system is often not the only task in scientific applications but rather a step that gets called numerous times throughout the application as part of a loop. Because of the widespread use and dependence of solving linear systems, any improvements to the time it takes to solve a linear system can have a large impact on the overall performance of scientific applications [25].

Coefficient matrices are commonly categorized into two groups: sparse or dense, which are useful for determining the best methods for solving the given linear system. As their names suggest, a dense matrix is filled completely or almost completely with nonzero entries. A sparse matrix, however, consists predominately of entries which are equal to zero. Because of this difference, the methods used to solve linear systems with dense coefficient matrices can be very different than those used for sparse matrices. The work contained within this dissertation is focused only on solving sparse linear systems.

The methods for solving a system of sparse linear equations can be largely separated into two distinct categories: direct and iterative. These two methods are described in more detail in Sections 2.1.1 and 2.1.2 respectively. The work presented throughout the rest of this dissertation is primarily concerned with iterative solvers, but direct methods are described in order to provide a

more complete overview of linear solving techniques.

2.1.1 Direct Solvers

Direct solvers, at their core, are based on the basic idea of Gaussian elimination. Conceptually, Gaussian elimination is the process of repeatedly performing elementary row operations on the linear system until A has been transformed into an upper-triangular matrix U. The linear system still retains its data during this transformation, but becomes easier to manage and solve due to its new layout.

$$U = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ 0 & & & a_{mn} \end{bmatrix}$$

Gaussian elimination is based on three basic row operations:

Row swapping: Exchanging two rows with each other, $R_i \leftrightarrow R_j$,

Row multiplication: Multiplying a row by a scalar, $cR_i \rightarrow R_i$,

Row addition: Adding one row to another, $cR_i + R_j \rightarrow R_j$.

In order to create an upper-triangular matrix, all elements below the diagonal must be equal to zero. To transform the given matrix into this upper-triangular matrix we iteratively traverse the columns from left to right and perform Gaussian elimination to make the column elements below the diagonal equal to zero. Rather than always using the original diagonal element as a basis for introducing zeros, we can choose any element within the current column to be used as the **pivot**. The pivot entry x_{ij} is often selected to be the largest nonzero element in the current column. Once the pivot has been determined, the row containing the pivot is then swapped with the row containing our current "original" pivot element at x_{ii}. After the swap, the algorithm proceeds as normal and removes multiples of the pivot row from the rows below the diagonal. The process is continued for

the x_{i+1} column through the x_n column until the triangular matrix has been created. The pivoting process is depicted in Figure 2.1. In practice, when rows are exchanged the permutation matrices P_i interleave with the lower triangular L_i matrices. In this case it is possible to factor the matrix as $PA = LU$ where P is a $m \times m$ permutation matrix [26].

$$\begin{pmatrix} x & x & x & x \\ & x & x & x \\ & x_{ij} & \boldsymbol{x} & \boldsymbol{x} \\ & x & x & x \end{pmatrix} \rightarrow \begin{pmatrix} x & x & x & x \\ & x_{ij} & \boldsymbol{x} & \boldsymbol{x} \\ & x & x & x \\ & x & x & x \end{pmatrix} \rightarrow \begin{pmatrix} x & x & \boldsymbol{x} & \boldsymbol{x} \\ & x_{ij} & x & x \\ & 0 & \boldsymbol{x} & \boldsymbol{x} \\ & 0 & \boldsymbol{x} & \boldsymbol{x} \end{pmatrix}$$

Figure 2.1: The process of selecting a pivot element, exchanging its row, and then removing elements below the diagonal.

The resulting augmented, upper-triangular matrix U can then be used to solve for the variables in the column vector x using the straightforward method of back-substitution. However, by simply storing the row operations performed during the Gaussian elimination, the simple Gaussian elimination is transformed into the more powerful and capable LU decomposition which is the basis for many direct linear solvers [27, sec. 3.2]. The series of elementary row operations during Gaussian elimination is equivalent to multiplying the original matrix A by a series of lower triangular matrices, $L_0 L_1 \dots L_n A = U$. These successive lower-triangular matrices are then compiled into a single lower-triangular matrix L which is then multiplied with the resulting matrix U such that $LU = A$ where L is lower-triangular and U is upper-triangular. L and U can then be used to solve the original problem of $Ax = b$ by transforming the problem into $LUx = b$. The linear system can then be solved by solving two smaller problems:

(1) Solving $Ly = b$ for y where $y = Ux$ via forward substitution

(2) Solving $Ux = y$ for x via backward substitution.

Solving a rectangular $m \times n$ matrix A via the LU decomposition creates an L of size $m \times m$ and U of size $m \times n$. The cost of performing the full factorization is $O(\frac{2}{3}m^2 n)$ operations for the

matrix decomposition and another $O(m^2)$ and $O(mn)$ operations for the forward and backward substitutions respectively. The benefit of retaining the L and U matrices, as opposed to only storing U, is that it greatly simplifies any repeated linear solves in which the coefficient matrix A remains the same, but the solution vector b changes. This scenario creates efficient subsequent computations by only needing to perform the forward and backward substitution operations since the matrix has already been decomposed. Because of their reliance on matrix decomposition methods, direct linear solvers do not produce any form of "intermediate" results during the solve. Direct solvers only produce a solution, or any useful information, once all the operations have been successfully performed.

Although the LU decomposition is encountered frequently, direct solvers are not limited to its use; solvers are also based on the QR and Cholesky factorizations [27, sec. 5.2, 3.2]. Each of these factorizations produce two matrices from the original coefficient matrix A in a manner similar to that of the LU factorization. The QR factorization method factors the matrix A into an orthogonal matrix Q and an upper triangular matrix such that $A = QR$. The solution x can then be determined by first transforming the linear system into the form $Rx = Q^*b$. Once in this modified form x can then be solved using the two sub-problems: $y = Q^*b$ and $Rx = y$.

The Cholesky factorization transforms A into a lower triangular matrix L and its conjugate transpose L^* such that $A = LL^*$. However, unlike the LU and QR factorization methods, the Cholesky factorization method is only applicable to the subset of matrices which are both Hermitian and positive-definite. A matrix is Hermitian if it is equal to its own conjugate transpose $A = A^*$ and a matrix is considered positive-definite only if all of its eigenvalues are positive. Once the matrix has been factored, the system can then be solved using the sub-problems: $Ly = b$ and $L^*x = y$, in a fashion similar to that of the LU factorization. Both the LU and QR factorization methods can be performed in $O(\frac{2}{3}mn^2)$ floating-point operations (flops), while the Cholesky factorization, exploiting the symmetry and positive-definiteness of the matrix, can be performed in $O(\frac{1}{3}n^3)$ flops.

Regardless of what direct linear solver is used, each will arrive at the same solution for a reasonably conditioned linear system. The condition of a matrix describes how susceptible the

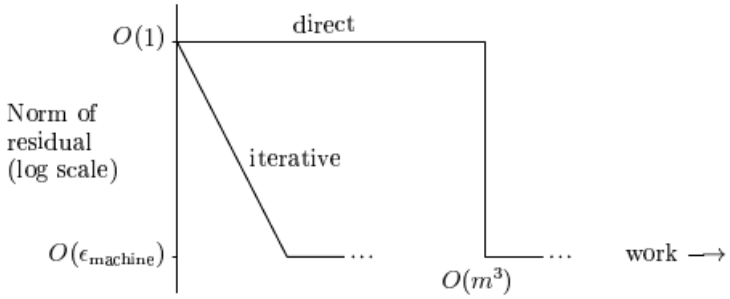
output is to small perturbations to the input. The main disadvantages of direct solvers is that both their memory usage and floating-point operation counts are based entirely on the dimension of the matrix. This is not an issue in regards to full-rank matrices or matrices which are nearly full of nonzero entries, however, the problem becomes more apparent when solving linear systems with large, sparse matrices. Since the complexity growth of direct solvers is based on the dimension of the matrix as opposed to how many relevant nonzero entries it has, the floating-point operations and memory cost of a direct solver can become prohibitive when working with sparse matrices of a large dimension. This cost is based on dimension since each step of a direct solver runs the risk of introducing fill-in into the matrix. Fill-in occurs when data that was originally a zero entry within the matrix has been changed into a non-zero entry as the result of a row operation. The cost to perform a direct linear solve is thus asymptotically the same for a dense matrix and a sparse matrix of the same dimensions, even though the sparse matrix contains significantly fewer nonzero entries.

Due to the potentially prohibitive cost, solving large sparse matrices using direct solvers is not always the ideal choice. To address this problem, researchers have developed various methods which create a new class of **iterative** solvers, specifically designed for solving large sparse systems. Iterative solvers use a vastly different approach to solving a linear system than direct solvers which does not grow in complexity based on the dimension of the linear system, but rather the number of nonzeros contained in the coefficient matrix.

2.1.2 Iterative Solvers

Compared to the numerical factorization methods used in direct solvers, the core numerical ideas that form the basis of iterative solvers are incredibly different. Rather than determining a solution after a finite number of steps using factorization methods, iterative solvers produce a series of increasingly accurate approximations to the unknown vector x. The difference between an approximate solution vector and the actual solution vector is called the **residual**. An iterative solver terminates when either a predetermined number of iterations has been reached or when the solution is less than the convergence tolerance, where the approximate solution to x is within a

user-specified range of the exact solution for x. Figure 2.2 shows how the residual associated with an iterative solver descends towards zero consistently as opposed to a direct method which happens suddenly only after a certain amount of work has been performed.



[26, p. 247]

Figure 2.2: Graph of the theoretical amount of work to converge to a solution via iterative and direct methods. The iterative method should decrease geometrically while the direct method makes no progress until all of its $O(m^3)$ steps have been performed

The amount of storage, $O(mn)$, and floating-point operations, $O(mn^2)$, used in a direct solve are entirely dependent on the dimension of the $m \times n$ coefficient matrix A. The cost of performing the direct solves is manageable as long as the matrix is dense such that the number of nonzero entries within A is close to $m \times n$. Iterative solvers are useful when solving linear systems where direct methods are too expensive to compute and/or store in memory. Using a direct solver on a linear system with a sparse coefficient matrix A can be prohibitively expensive since the relatively small amount of "useful" data is ignored since only the dimensionality plays a role in the asymptotic performance. Iterative solvers on the other hand, attempt to exploit sparse matrices by concerning themselves with only the nonzero entries of the matrix rather than also processing the large number of zero entries.

Unlike direct solvers, the solutions produced by iterative solvers can differ slightly from solver to solver while still being "correct." The variability in the solutions are a result of the size of convergence tolerance, underlying numerical algorithms, and floating-point rounding. Choosing both the number of iterations and the convergence tolerance for solving a given linear system can be a difficult task as it requires in-depth knowledge of the current problem and the iterative methods

being used to solve it. The convergence tolerance is determined largely by the type of problem being solved and the level of precision needed in the final solution. Because iterative methods execute in a non-deterministic fashion it is difficult to know if a given problem can be solved within a set number of iterations. Selecting the number of iterations is also difficult since it is dependent not only on the size of the convergence tolerance, but also the size and layout of the coefficient matrix A and choice of iterative solver.

Each iterative solver has distinct differences from the others, but they all follow the same general structure. At each iteration, i, the solver produces an approximate solution vector x_i which is closer to the true value of x than the previous iteration x_{i-1}. Commonly the zero vector is chosen for the initial guess of x_0 since the starting point of the algorithm is largely arbitrary and it is difficult to assume any "best" initial approximations *a priori*. Iterative solvers create approximations to the optimal solution vector x_* with each iteration i which become more accurate as the number of iterations increases. By stopping the iterative process before reaching our exact solution, execution time can be saved by exiting the iterative process early once we have deemed that our iterative solver has converged towards a solution within an acceptable degree. An iterative method has converged when the approximation vector x_i is found to be within some acceptable difference of the solution vector x_*. At each iteration, the selected convergence tolerance is compared to the relative residual at that iteration, r_i. The relative residual is recomputed at each iteration and compares the current residual of x_i, calculated by $b - Ax_i$, to the norm of the original solution vector b. The resulting relative residual for a given iteration i is therefore represented as $r_i = \frac{\|b - Ax_i\|}{\|b\|}$. Once the residual is less than the convergence tolerance the solver terminates.

A large proportion of modern sparse iterative methods are based on the important idea of Krylov subspaces. However, before describing the Krylov subspace we must first briefly understand what a Krylov sequence is. A Krylov sequence, depicted below, is a series of vectors created by repeatedly applying a matrix A to a vector b,

$$Kseq_n \;=\; \{b,\; A(b),\; A(A(b)),\; \ldots\} \;=\; \{b,\; Ab,\; A^2b,\; \ldots,\; A^{n-1}b\}.$$

The corresponding Krylov subspace is created from the space spanned by the vectors contained in the Krylov sequence. Each Krylov subspace can therefore be thought of as the polynomial of all the possible linear combinations created from its vectors,

$$K_n \;=\; \alpha_0 b + \;\alpha_1 A(b) + \;\alpha_2 A^2(b) + \;\ldots\; + \alpha_{m-1} A^{m-1}.$$

Krylov subspaces are an integral component in many of the modern popular sparse iterative methods because they are efficient methods for iteratively approximating a solution to a linear system using a much smaller dimensional space than the space spanned by the original problem. By starting with a single vector space and adding one additional vector at a time into it, we have a relatively small dimension of a search space which may contain an accurate approximation to the solution vector. By reducing the dimension of the problem it is possible to find an appropriate approximation in much less time than if we used a direct solver, depending on the rate of convergence. Although each iterative solver varies in its exact algorithm, computing each additional vector in the Krylov sequence only requires a single sparse matrix-vector multiply. This matrix-vector multiply takes $O(mn)$ floating-point operations in general, but if the matrix is sufficiently sparse it can be performed in far fewer operations.

Within each iteration of a Krylov-based linear solver, an additional vector is created and added into the Krylov subspace thus increasing the order of the Krylov polynomial by 1. Each additional vector increases the search space and can allow for a more accurate approximate solution than the previous iteration's search space. It is proven that the solution to a given nonsingular linear system exists within a Krylov subspace of the same dimension as the minimal polynomial of the matrix [28]. Since all previously created Krylov subspaces are wholly contained with any additionally created subspace, $K_1 \subset K_2 \subset K_3 \subset \ldots \subset K_n$, each iteration of the linear solver contributes a single vector to the subspace in order to expand it. As the Krylov space becomes larger, more solution vector options are available for selection. In order for an iterative method to be efficient, the approximate solution needs to converge in far fewer steps than the dimension of the linear system. Unfortunately this cannot always be guaranteed due to a variety of reasons including rounding errors and ill-conditioned

matrices. The speed at which a Krylov solver tends to converge is largely associated with a matrix's condition number which is directly tied to the layout of its eigenvalues. Condition numbers are discussed in more detail in Section 2.2.

There are many possible iterative solvers to choose from when solving a linear system. Below are brief overviews of each of the iterative methods used in the experiments discussed later in the dissertation. Each solver's implementation comes from the Trilinos mathematical library discussed further in section 2.3.

2.1.2.1 Conjugate Gradient

The conjugate gradient method (CG) [1, p. 196] is used for solving linear systems which are both symmetric and positive-definite. Because the system is both symmetric and positive-definite, the shape of the function is convex and the solution to the system Ax=b can be reached by minimizing the equation $f(x) = 1/2x^T Ax - b^T x + c$. This minimization is possible since the derivative of the function is equal to the original Ax=b.

Each iteration of the conjugate gradient method contributes to creating a sequence of vectors which are all conjugate to each other. The vectors in the Krylov sequence are created from the residual obtained from the approximate solution at each iteration, $r_i = b - Ax_i$. The conjugate gradient method is similar to the method of steepest descent but is quicker to approach the solution due to always choosing an optimal and unique direction rather than simply the direction where f(x) decreases the most quickly. Choosing conjugate vectors prevents any of the descending directions from overlapping with other vectors which is a common occurrence in steepest descent. The method was originally conceived as a direct method, but can be considered an iterative method by finding a good approximate solution after relatively few iterations $\ll n$.

2.1.2.2 Minimum-Residual

The minimum-residual method (MINRES) [1, p. 145] is a Krylov method applicable to symmetric systems. MINRES's structure and execution are similar in design to the conjugate

gradient method, but MINRES is more general in nature by not being limited solely to positive-definite systems. At each iteration the MINRES method determines an approximate solution vector x_i that minimizes the residual in the 2-norm: $r = \|b - Ax\|_2$.

2.1.2.3 Generalized Minimum-Residual

The generalized minimum residual method (GMRES) [1, p. 171] is a general Krylov method capable of solving non-symmetric linear systems. GMRES is the generalized version of the MINRES method for non-symmetric systems and similarly computes a sequence of orthogonal vectors. At each iteration GMRES computes the new approximation solution by solving a least squares problem involving the current Krylov subspace. With each additional iteration, GMRES requires both more floating-point operations and memory than the previous iteration. These additional costs are caused by the sequence of orthogonal vectors being explicitly stored [2, ch. 2]. Because of these increasing costs it is common to routinely "restart" the method after a number of iterations have completed. Restarted versions of GMRES limit the amount of computational and storage costs by using the most recently computed solution approximation x_i as the initial guess for a new run of GMRES.

2.1.2.4 LSQR

The LSQR method [29] is a general Krylov method capable of solving non-square linear systems. LSQR is heavily based on the conjugate gradient method and is equivalent to applying the CG method to the symmetric positive-definite equation: $A^T Ax = A^T b$.

2.1.2.5 Transpose-Free Quasi-Minimal Residual

The transpose-free quasi-minimal residual method (TFQMR) [1, p. 247] is a general Krylov quasi-minimal residual (QMR) method with many similarities to the GMRES method. The term Quasi-minimal in this context refers to the created Krylov subspace being bi-orthogonal instead of orthogonal like in the GMRES method. The original QMR method involves matrix-vector multiplies using both the coefficient matrix A and its transpose A^T, while the modified TFQMR method only

requires the former.

2.1.2.6 Biconjugate Gradients Stabilized

The biconjugate gradients stabilized method (BiCGSTAB) [1, p. 244] is a Krylov method capable of solving non-symmetric linear systems. BiCGSTAB is a modified version of the biconjugate gradients (BiCG) Krylov method which is itself a modified version of the conjugate gradient (CG) method. BiCGSTAB enhances the BiCG method by stabilizing its somewhat erratic convergence rate and removes its dependence on the transpose matrix A^T. The stabilization technique smooths convergence by performing a GMRES iteration after each k iterations of the BiCG method.

2.2 Preconditioners

Choosing an iterative linear solver does not necessarily guarantee that a solution will be found for a given linear system. Even if a solution is found, it may not be found within a reasonable amount of time. A linear solver not being able to find the solution within a reasonable amount of time is typically related to the linear system being ill-conditioned. The **condition** of a linear system describes, roughly, how easily it can be solved using iterative linear solvers.

The convergence rate of a system describes how quickly the residual of the problem approaches zero. is slower than the rate of convergence for a . An ill-conditioned system is more likely to encounter solving errors or to have a slow convergence rate, while a well-conditioned system is more likely to have a high convergence rate and require relatively small number of iterations.

Although it is common to refer to linear systems simply in terms of ill- or well-conditioned, the more definitive **condition number** of the system can be computed. The condition number of a matrix is the value associated with a linear system which measures to what extent the vector x, in the equation $Ax = b$, can lose precision in the worst-case scenario. A condition number κ must be within the range

$$1 \leq \kappa \leq \infty.$$

The condition number itself describes how susceptible the vector x, in $Ax = b$, will be to any changes

in the solution vector b. More specifically, the condition number measures the ratio of the maximum relative stretching to the maximum relative shrinking performed by the matrix A on an arbitrary vector. The condition number of a linear system is computed by multiplying the norms of the matrix A and its inverse,

$$\kappa = \|A\|\|A^{-1}\|.$$

The condition number is dependent not only on the coefficient matrix A, but also on the specific norm used in the computation.

The larger the condition number of a matrix, the closer the matrix is to being singular. Therefore, a matrix with a condition number of ∞ is singular while a matrix with a condition number of 1 is indicative of the matrix being the identity matrix I. The vast majority of linear systems are therefore somewhere in the large zone between these two extreme cases.

Preconditioners are specifically designed numerical algorithms which aim to lower the condition number of a linear system, thus making the system easier to solve with an iterative method, without changing the original solution to the problem. A preconditioner is created by using a non-singular matrix M which is an approximation of the matrix A such that the original linear system $Ax = b$ can be be re-written as

$$M^{-1}Ax = M^{-1}b.$$

Since the matrix M^{-1} has been applied to both sides of the equation, the original solution to the problem can still be determined, but the structure and values of the system have changed. Ideally the preconditioner is the exact inverse of the matrix A, thus reducing the left-hand side of the equation to simply be x and creating a trivial solve. However, this is not practical in most cases due to the high cost, $O(n^3)$, of computing the inverse of an arbitrary matrix. Instead, preconditioner algorithms create a matrix as close to A's inverse as possible while still being computationally cheap and easy to produce. There are many ways to create a preconditioner and each algorithm has performance and generality trade-offs. A simplistic preconditioning example involves removing all non-diagonal entries from the matrix A.

Once the preconditioner has been applied, instead of the matrix A's properties dictating the convergence rate, the properties of the newly formed matrix $M^{-1}A$ are now responsible for the convergence rate of the linear system. By carefully choosing the values and layout of the preconditioner matrix M we can create a more well-conditioned system than given in the original problem, while still being able to approximate the original solution.

The preconditioning equation $M^{-1}Ax = M^{-1}b$ is an example of only one type of preconditioning: left preconditioning. However, it is also possible to apply right preconditioning to the linear system by solving

$$AM^{-1}y = b, \text{ with } x = M^{-1}y$$

or to perform split preconditioning which applies a preconditioner to both sides of the matrix in order to preserve symmetry

$$M_L^{-1}AM_R^{-1}u = M_L^{-1}, \text{ with } x = M_R^{-1} \text{ where } M = M_L M_R$$

In practice, it is not necessary to create the full preconditioner matrix M. Similar to the methods used in creating Krylov spaces, the matrix M itself is rarely created in full and is instead applied by performing a series of linear operations on the necessary vectors of the system.

The most important attributes of a newly created preconditioned system are its spectral properties, the organization of its eigenvalues and eigenvectors. In general, a preconditioner matrix M is considered to be good if "$M^{-1}A$ is not too far from normal and its eigenvalues are clustered" [26, p. 314]. This clustering of the eigenvalues is important because it indicates that the matrix is far from non-singular as evidenced by having multiple, nearly evenly-weighted transformation directions.

Much like iterative solvers, preconditioning techniques can be broken down into smaller subsections based on the types of matrices which they support. Included in this section are brief overviews of the few preconditioning techniques used throughout this dissertation.

2.2.1 Jacobi

The simplest form of preconditioning a matrix is Jacobi preconditioning and consists of replacing the coefficient matrix A of a linear system with only its diagonal entries. Jacobi preconditioning is far from a general-use preconditioning algorithm, but can prove effective especially when dealing with matrices that are diagonally dominant. A matrix is considered to be diagonally dominant when every diagonal entry of a square matrix is of a larger magnitude than the sum of the magnitude of all other non-diagonal entries on that row,

$$|a_{ii}| \ \geq \ \sum_{i \neq j} |a_{ij}|, \ \text{ for all } i \text{ in } A.$$

2.2.2 Incomplete LU Factorization

One of the most popular methods for preconditioning sparse matrices today is based on the LU factorization methods discussed in Section 2.1.1. Rather than attempting to fully solve the equation $A = LU$, the incomplete LU factorization (ILU) instead attempts to find an approximation of L and U such that $A \approx LU$. By approximating both the upper and lower portions of the factorization it is possible to achieve results similar to the full factorization, but by using less computational power. The major component that contributes to these savings is the result of removing various portions of the fill-in that normally occurs during the LU process, therefore approximations to the upper and lower matrices result in an "incomplete" version of the factorization. Choosing what values of fill-in to remove or not has spawned a collection of algorithms which are based on the same general LU factorization scheme.

Two important variables used throughout most incomplete LU factorization algorithms are the **level of fill** and **threshold**. The level-of-fill associated with an incomplete LU implementation represents the allowable number of zero elements in A which become nonzero during the factorization. Level of fill is based on the sparsity pattern of the matrix A, a binary pattern which represents if a given value is a zero or nonzero entry. For instance, a zero level of fill, does not keep any of the fill-in produced throughout the algorithm. Therefore zero fill results in an incomplete LU factorization

which contains the same number of nonzero entries as in the original matrix A. For an arbitrary level-of-fill k, the amount of nonzero entries allowed in the LU decomposition is equal to the number of entries in the sparsity pattern for A^{k+1}.

The other important variable in incomplete LU factorization is the threshold, which is yet another method for reducing a certain amount of fill-in which occurs during the factorization process. For a given threshold t, any fill-in which occurs during the factorization process is set to zero for any entry whose value is less than that of the current threshold:

$$a_{ij} = \begin{cases} 0 & \text{if } a_{ij} < t \\ a_{ij} & \text{otherwise} \end{cases}$$

2.2.3 Chebyshev

Chebyshev polynomials are a sequence of recursively defined polynomials $\{T_0, T_1, ..., T_n\}$, with each polynomial in the sequence designed to be orthogonal to all of the subsequently created polynomials [30] [31, Ch. 3]. The Chebyshev preconditioner is an application of Chebyshev polynomials to a matrix A which has been transformed such that the spectral range of the matrix is no longer between its smallest and largest eigenvalues, $[\alpha, \beta]$, but is shifted to be $[-1, 1]$. Using estimates of the minimum (α) and maximum (β) eigenvalues of A, a matrix Z is created which shifts the spectral range of A to be $[-1, 1]$. This collection of polynomials can then be used as a preconditioner because they are capable of solving for the inverse of A using the equation

$$A^{-1} = \frac{c_0}{2} I + \sum_{i=1}^{i=\infty} c_i T_i(Z).$$

2.3 Trilinos Library

Trilinos [19,32] is an open-source library, maintained by Sandia National Laboratory, providing a framework for "the solution of large-scale, complex multi-physics engineering and scientific problems". Trilinos contains numerous discrete packages which offer unique functionality related to numerical and scientific computing. The computational areas covered by these packages include

solving linear systems, preconditioning, discretizing PDEs, and load balancing. Many of the mathematical routines in Trilinos make use of existing established numerical libraries such as LAPACK [18] and BLAS [12] while also offering interfaces with newer libraries such as SuperLU [33], Mumps [34], and PETSc [16]. Trilinos, which is written largely in C++, also supports a variety of common scientific computing programming languages such as Fortran, Python, and C through interfaces.

Although there are many unique packages available within Trilinos, the work presented in this dissertation focuses on solving linear systems and therefore depends primarily on three Trilinos packages: Tpetra for creating and distributing dense vectors and sparse matrices, Belos for solving linear systems, and Ifpack2 for preconditioning sparse matrices.

2.3.1 Tpetra

Tpetra is a Trilinos package focused on the creation and manipulation of distributed data structures representing linear algebra objects such as sparse matrices and dense vectors [35]. Tpetra is one of the core packages within Trilinos and is used or supported by a large percentage of the total packages due to providing the data structures for common elements. Tpetra is the successor of another Trilinos package, Epetra, and improves upon it in a variety of ways but the most important are the inclusion of complex data types for scalars, support for 64-bit global and local indices, and more extensive support for shared-memory parallelism.

Tpetra uses user-created objects called maps for dictating how a given object should be distributed across the available processes. A Tpetra map object can use up to a 64-bit integer to represent the number of global and local indices, although it is not required that they be the same size. The global indices correspond to the number of rows or columns in a sparse matrix or the number of rows in a multi-vector. Each row or column can subsequently contain as many elements as supported by the size of the local indexing type. Typically 64-bit indices are only used in cases of more than ~ 2 billion (2^{31}) unknowns. Matrix and vector entries can be filled randomly, read from matrix file formats, or assigned explicitly.

Tpetra uses hybrid parallelism through the use of MPI [36] for distributed-memory parallelism, and another Trilinos package, Kokkos [37], for shared-memory parallelism. Kokkos provides a programming model which acts as an abstraction layer allowing the code to target multiple shared-memory models with the same interface. Kokkos is currently capable of targeting OpenMP [38], POSIX threads [39], and CUDA [40]. One of the key features of Kokkos is its ability to alias memory indexing in the background to best suit the architecture it is being run on. Because of this hybrid parallelism structure, Tpetra scales well to a high number of parallel processes and has been successfully tested in computations using 512k cores [35].

2.3.2 Belos

Belos [41, 42] is a Trilinos package that provides iterative linear solver routines. The solver methods within Belos consist almost exclusively of various Krylov subspace methods, but Belos also contains a few routines based on fixed-point iterative methods. Belos' reliance on the underlying Tpetra library and consequently the Kokkos library allows its methods to be more flexible in how they target shared-memory architectures by abstracting away portions of the algorithms which can then target specific architectures or programming models. This flexibility allows for changing the representation of the underlying matrix data into a more optimized format for the given problem and available computer architecture, allowing for more efficient use depending on cache sizes, core counts, etc.

In addition to the more traditional iterative methods which solve a single linear system for a given right-hand side, $Ax = b$, Belos also provides **block** methods for solving multiple linear systems $AX = B$ with the same coefficient matrix, A. Many of the solvers contained within Belos are designed to be a hybrid of the single right-hand-side solvers and block solvers. These hybrid iterative solvers called **pseudoblock** methods. Pseudoblock and block solvers fundamentally solve the same problems of type $AX = B$ where multiple right-hand sides can be solved for simultaneously, but pseudoblock solvers have the added advantage of being able to solve single-vector right-hand sides. Pseudoblock solvers are an attempt at bridging the gap between methods which perform strictly

block solves and those which strictly solve single-vector right-hand sides. When solving for multiple right-hand-sides, pseudoblock solvers work by executing the single-vector iterative solver in lock-step with each right-hand side. This lock-step algorithm is performed by applying any preconditioners and the matrix A to all vectors in a block at once and then using block vector operations. Once one or more of the linear systems have reached convergence, they are removed from the current block-view and the iterative process continues onward for the remaining unconverged systems.

2.3.3 Ifpack2

Ifpack2 [43, 44] is the Trilinos package that provides preconditioning methods for Tpetra based linear objects. The available methods in Ifpack2 include incomplete factorizations, relaxations, and domain decompositions. For our purposes, the routines available within Ifpack2 are used for preconditioning sparse matrices prior to being solved with the routines available in Belos. Similar to Belos, Ifpack2 uses the underlying Tpetra and Kokkos packages to support distributed- and shared-memory MPI + X systems.

Chapter 3

Machine Learning Background

This chapter presents the core machine learning techniques and algorithms used throughout the rest of this dissertation. Section 3.1 provides an introduction to the basics of machine learning techniques and the usage of supervised machine learning methods. Section 3.2 describes the specific machine learning classification methods and algorithms used in the experiments of later sections. Section 3.3 goes over the methods used to select the importance of the features in a machine learning dataset.

3.1 Supervised Learning

Machine learning, as a general concept, uses statistical techniques and algorithms to predict future outcomes based on one or more inputs. In machine learning the input variables are commonly referred to as predictors, features, or independent variables, while the output variables may be referred to as responses, classifications, or dependent variables.

There are two main divisions within the world of machine learning methods. The first division is found between supervised and unsupervised machine learning methods. Supervised learning occurs when there is a distinct output variable y for one or more input variables contained in a matrix X such that $f(X) = y$. Supervised methods attempt to approximate the mapping function $f()$ such that the correct output is predicted for a given input. Supervised learning occurs when the machine learning algorithm is trained on a series of example, or training, data that contains both the vector of input data x_i and the corresponding classification output y_i, for each input-output

instance example i.

Unsupervised learning methods can contain the same input data X, but the main difference between the two methods is that there is no corresponding output variable y associated with X. Therefore, an unsupervised machine learning algorithm's goal is to create a model of the general structure, distribution, or connections of the available data, without determining a specific numerical or class-based output variable. In general, unsupervised methods are used for finding similarities or patterns within the input data. Since there is no corresponding correct or incorrect output, groups of items with similarities will be more closely correlated than those without. One of the key differences between unsupervised and supervised learning is that there are no methods available for examining the accuracy of an unsupervised algorithm since there is no correct or incorrect output associated with the input vector.

In addition to the division between supervised and supervised learning, a second division in machine learning lies between machine learning algorithms that produce output based on regression, while other algorithms are based on classification. Simply, regression is used when our output variables are continuous, while classification describes scenarios where the only options for an output variable comes from a finite set of possible classes. More formally, given the function $f(X) = y$, for some input matrix X and corresponding output y, regression describes machine learning methods where y is a continuous variable, such as time. Classification, however, occurs when the output y is one or more labels selected from a finite group, $\{y_1, y_2, \ldots, y_n\}$. The work contained within this dissertation is focused on experiments where the machine learning methods used are both supervised and classification based.

3.2 Binary Classification Methods

Classification based machine learning methods come in many flavors, but the commonality between them is that input data are mapped to one or more labels chosen from a finite set of labels. Classification algorithms come in three main varieties: binary, multi-class, and multi-label.

(1) A binary classifier exists when there are only two possible output labels for a given problem and only one label may be selected,

$$Input = \begin{cases} Out_1 & \text{if } Input \text{ is labeled as true} \\ Out_0 & \text{if otherwise} \end{cases}$$

(2) Multi-class classifiers map the input vector to only one of more than two output classes.

$$Input = \begin{cases} Out_1 & \text{if } Input \text{ is labeled as 1} \\ Out_2 & \text{if } Input \text{ is labeled as 2} \\ \vdots & \\ Out_n & \text{if } Input \text{ is labeled as } n \end{cases}$$

(3) Multi-label classifiers map the input vector to one or more of the possible output classes.

$$Input \quad \subseteq \quad \{label_1, label_2, \ldots, label_n\}$$

In order to transform our existing problem into a binary classification problem, we replace the continuous output of time, associated with each solver-preconditioner pair solving a given linear system, with one that is binary. This transformation is done by replacing the specific times with either "good" or "bad" labels. Determining whether or not a time should be replaced with "good" or "bad" is not an objective task and as such, there are multiple ways of performing this task. For our experiments we classify a solver-preconditioner pair as "good" for a linear system if it is the fastest performing pair that converged to a solution for that system. Since there may be multiple pairs that have similar wallclock times to the fastest pair, it is acceptable to consider these other pairs "good" as well. Throughout the experiments, any pair that performs within 125% of the fastest pair for that given linear system, is considered "good." All other slow pairs that converged are deemed to be "bad" along with any pairs that did not converge, or pairs that encountered some form of solving error.

Many binary classification techniques exist, each with their own unique performance and prediction characteristics. In the experiments described in later chapters, various binary classification techniques are examined to determine their success on predicting the best solver-preconditioner pairs. The highest performing classification algorithms for the solver-preconditioner problem are described below.

3.2.1 k-Nearest-Neighbor

The k-nearest-neighbor (KNN) [45, 46] is one of the more simplistic classification methods. KNN does not require any form of predictive modeling, and instead relies on statistical "memory-based" methods, where all training data is stored locally and the analysis is deferred until a classification needs to be made. Each data point from the training set exists in a multi-dimensional space called the "feature space." The KNN algorithm operates by selecting the $k \geq 1$ closest training data points to the location of the currently queried data point within the feature space. The closest neighbors to the desired data point are determined using the Euclidean distance, $d_i = \|y - x_i\|$, between the input vector y and each training data vector x_i within the feature space. The resulting classification prediction for the input vector is determined to be the majority classification label assigned among the nearest neighbor. Ties for the majority case are decided by random selection.

3.2.2 Decision Tree

Decision trees, also known as classification and regression trees (CARTs) [45, 47], are a classification method based on identifying key features of data and creating splits of the variables such that certain input values result in certain output labels. These splits take place in the feature space of the input data, and divide the space into different classification regions based on recursive binary decisions. Each tree is created by determining what input variables are best for splitting the classifications into two regions based on some threshold of the variable. The regions that are created as part of the tree are subdivided further, if needed, to create more fine-grained classifications based on other splits determined by the input variables. Decision trees can also be split in ways

other than binary, but these multi-splits tend to be avoided because each multi-split is equally performed as a series of binary splits. Finding the absolute best splits for a given data set is known to be a NP-complete problem and is therefore not feasibly computed. Instead, decision trees are created using a greedy algorithm that continues to create more fine-grained subdivisions until a given number of decisions have been determined, or when adding binary splits no longer provides any significant increases in classification performance. One of the main disadvantages associated with using a decision tree method for classification is that the greedy nature of the core algorithm can result in non-optimal choices, which can lead to performance degradation.

3.2.3 Random Forest

Improving upon the success and capabilities of the decision tree methods, random forests [47] create classifications by creating multiple decision trees and combining their respective results into a single classifier. Rather than training a single decision tree on all of the available input data, a random forest creates multiple decision trees that each train and create classifiers using only a given subset of the input data. By allowing each decision tree to learn on different portions of the dataset, multiple binary splits are created, which can result in some strong agreeing with one another, while others are complete opposites. Decision tree algorithms use a technique called "bagging" (Bootstrap aggregating) that samples the input dataset in a uniform manner with replacement, allowing for multiple instances of the same input vector in a given sample subset. Bagging is considered to be a form of ensemble learning, a term used to describe any machine learning algorithm that combines more than one algorithm to achieve higher performance. Ideally, for large input sets, each training data subset will consist of $(1 - \frac{1}{e})$, approximately 63.2%, unique data points from the input data, with the remaining percentage consisting of duplicates. The overall variance of each decision tree can be lowered by combining and averaging together the various trees such that

$$f(x) = \sum_{i=0}^{N} \frac{1}{N} f_i(x),$$

where f_i represents the ith decision tree created.

3.2.4 Gradient Boosting

Gradient boosting [47,48],is another type of ensemble method. Boosting, in this context, refers to a technique that incrementally builds and creates a classifier that adapts from the information obtained of previous iterations of the classifier. Special emphasis is placed on adjusting the classifier to correct for any mis-classifications obtained from the previous predictions training instances. At each iteration of the gradient boosting algorithm, the next iteration can be modeled as

$$F_{i+1}(x) = F_i(x) + h(x) = y$$

for some iteration i, the model F, and estimator h. Gradient boosting can best be viewed as a generic framework for solving the classification problem and requires three pieces:

(1) Loss function to be optimized

(2) Weak learner to make predictions

(3) Additive model to add weak learners to minimize the loss function.

The loss function is a differentiable function that attempts to quantify the overall cost of mis-classification. Commonly used statistical loss functions include the squared error and absolute error. In the context of the gradient boosting algorithm, weak learners are often depicted as the individual decision trees created and modified throughout the modeling process. In theory, the weak learners can be of many non-tree types, however, in practice it is common for the learners to be decision trees. The motivation for using the gradient descent algorithm, in this context, is to allow for the more generalized minimization of loss functions, no matter how complicated the loss functions may be.

3.3 Feature Selection

It is common in machine learning for data to contain more features, and therefore, information than is actually necessary for determining the resulting classification of a given entry. Reducing

the overall number of features, or columns, contained within a dataset allows for a smaller and simpler dataset to be create, which contains only those features which are important in contributing to the overall classification of the entry. There are two main types of features that can be easily removed, those that are redundant and those that are irrelevant. Redundancy occurs when more than one set of features are equivalent throughout a dataset. All but one of the redundant features can be excluded without having any impact on the prediction accuracy of the classifier. Irrelevant features occur when a feature does not have any impact whatsoever on the accuracy of predicting any entry's classification.

As part of the experimental process used throughout the rest of this dissertation, feature analysis and reduction is performed using the Least Absolute Shrinkage and Selection Operator (Lasso) [49]. In statistics, a Lasso is a linear regression method that uses a regularization process to rank the importance of features in a dataset. The most important features can often provide similar results to the full dataset, but since it is using only a subset of the original dataset, it can be processed and predicted in a shorter amount of time using a smaller amount of memory. Lasso uses a penalty system based on the L_1 norm of the feature data. The less importance the feature has on the resulting classification of an entry its resulting score from the Lasso will be closer to zero.

Mathematically the Lasso can be modeled as a least squares problem where

$$min_\beta : \quad h(\beta) = \frac{1}{2}\|y - X\beta\|_2^2 + \lambda\|\beta\|_1, \quad \text{where } \lambda \geq 0,$$

and where X is an $m \times n$ matrix representing the data collected on m entries with n features, y is an $m \times 1$ vector containing the resulting classifications of the data contained in X, and β is an $n \times 1$ vector representing the value attributed to each feature's importance. The second part of the equation above represents the penalty cost associated with the L_1 norm, the sum of the absolute values, of vector β. Increasing the *lambda* penalty results in more feature values being set to zero.

Throughout my experiments I use the Randomized Lasso [50], which uses the original Lasso as a sub-problem of a larger algorithm. Randomized Lasso, also known as stability selection, repeatedly selects random subsets of the feature data as well as different randomized subsets of the features

themselves. The regular Lasso algorithm is then applied on this smaller subset of data and the corresponding feature scores are computed and recorded. Each of these regular lasso executions on the randomly selected subset data is considered to be one iteration of the Randomized Lasso. As the number of Randomized Lasso algorithm iterations grows, the most important features associated with accurate classification tend to be selected more often from the randomly selected data and feature subsets.

The final importance scores produced for each feature represent the percentage of iterations that the feature was selected as an important feature in the total number of iterations wherein that feature was available. A score of 1.0, for instance, represents a feature selected as important in each of its possible iterations, while a score of 0.0 represents a feature that was never determined to be important.

Chapter 4

Methodology

This chapter presents the generalized methodology that is present throughout the various experiments contained in this dissertation. Section 4.1 describes the HPC Challenge benchmarks which determine various performance attributes for a given computer system. Section 4.2 discusses the software used to build and execute the experiments. Section 4.3 describes Anamod, a program which computes a variety of metrics for each linear system. Section 4.4 presents the machine learning and data analysis tools used throughout the experimentation process. Section 4.5 contains information about the collection of matrices used in the experiments.

4.1 Measuring System Performance

Computers are complex machines built from a variety of discrete components including memory, interconnects, and processors. Each of these components is responsible for aiding the computer in moving, storing, and processing data, respectively. The computer system's overall performance is determined not only by the performance of its individual components but also the performance characteristics of how the components interact with each other.

Computer hardware is constantly changing to make improvements to performance, reliability, efficiency, or some combination of the three. Due to the constantly changing landscape of computer hardware, it is expected that internal components of one system differ, at least in some ways, when compared to another system. It is not expected that two computer systems will perform in exactly the same way across a variety of different problems due to the large number component permutations

and the unique performance characteristics of each component and their interactions together.

Differences between a component's architecture, speed, or size can alter the performance of a given algorithm or application. Not every algorithm or application uses hardware in the same way or with the same performance. Some algorithms can be limited by the speed at which the processor can complete floating-point operations (compute-bound), however, other algorithms can be limited by the memory bandwidth available to move data from main memory to the processor (memory-bound).

Due to the unique nature of each computer system caused by its components, many libraries and applications are built and tuned to perform the best on the available hardware. In the case of computer clusters, it is expected that the machine's administrative team and other experts tune applications and libraries to a specific machine. However, there has been progress in automating the building and tuning process while still delivering comparable performance to the original hand-tuned builds. The Automatically Tuned Linear Algebra Software (ATLAS) [11] is a computational library containing a full implementation of the Basic Linear Algebra Subprograms (BLAS) [12] as well as some functions from Linear Algebra PACKage (LAPACK) [18]. ATLAS obtains information by probing the computer's hardware and, using the results of its tests, generating code from many possible options to best suit the given machine. This information can then be used to more appropriately use the computer's hardware. The machine-tuned version of the code allows for the kernels to take advantage of the hardware and thus perform better than an unoptimized reference BLAS implementation, but it will not perform as well as a hand-tuned code created by an expert user.

Determining the overall performance of a given computer is a non-trivial task. Performance can be based on metrics such as the number of floating-point operations per second, the amount of memory bandwidth, or the size of various caches. However, focusing on only one performance metric does not produce a generalizable description of the computer's capabilities. Instead, it is useful to combine multiple metrics, each representing a different aspect of the computational process, in order to obtain a clearer understanding of the system as a whole. The HPC Challenge is one example of a

benchmark with multiple system-performance metrics.

The HPC Challenge (HPCC) [51] is an open-source benchmark suite that analyzes a computer system's performance using metrics including floating-point performance, memory bandwidth, and random memory updates. Rather than creating new benchmarking methods, the HPCC consists of several established benchmarks combined into one code. This makes the input, output, and building simpler than it would be to perform for each benchmark. The seven sub-benchmarks available within the HPCC benchmark are described in detail throughout the rest of this Section. All of the collected system features used in the experiments are briefly described in Appendix A.2.

4.1.1 HPL

The High-Performance Computing Linpack Benchmark (HPL) [52] measures a computer's ability to perform floating-point operations. The number of floating-point operations a computer can complete in a second (FLOPS) is determined in HPL by performing parallel solves of dense linear systems using the BLAS-based LINPACK [53] subroutines. The dense linear systems used in these solves have randomly generated square coefficient matrices of user-specified dimension and are divided into blocks of user-specified dimension nb. The memory used to store the double-precision matrix A is designed to take up at least half of the system's main memory.

Each block is assigned to a process using a block-cyclic data distribution scheme as depicted in Figure 4.1. This results in each process being assigned sections of the matrix which are separated by a fixed stride width and height. Each block is assigned to one process based on a numbering scheme created from the layout of the processes specified by a two-dimensional rectangle of size $P_{rows} \times P_{cols}$. The block-cyclic distribution is commonly used in numerical linear algebra in an attempt to equally balance the workload across all processes and reduce the the overall communication overhead

After the matrix data has been distributed to the processes, each processor then computes the LU factorization of the various linear systems it is responsible for using row partial pivoting. The total number of floating-point operations required for the process is $\frac{2}{3}n^3 - \frac{1}{2}n^2$ for the factorization and $2n^2$ for the back/forward substitution solves. Using the resulting time-to-solution of the solve,

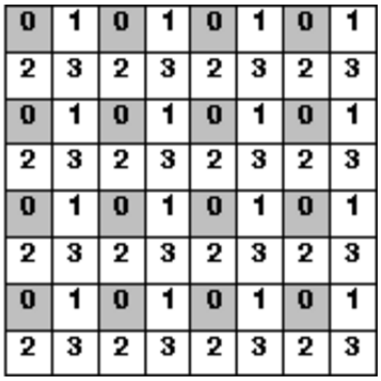


Figure 4.1: Block-cyclic distribution of a matrix of dimension (n=16) and block size (nb=2), across (p=4) processes laid out in a 2x2 process grid ($p_{cols} = p_{rows} = 2$) [54]

it is relatively simple to determine the total number of FLOPs required to perform the solve.

4.1.2 DGEMM

Double-precision general matrix-matrix multiply (DGEMM) [55] is a popular level-3 BLAS routine that is used for matrix multiplication. The algorithm, depicted in Equation (4.1), multiplies two scaled matrices together, A and B, and then optionally adds another scaled matrix C to the solution.

$$C := \alpha A \times B + \beta C. \quad A, B, C \in R^{n \times n} \tag{4.1}$$

The performance results of the DGEMM benchmark are based on how quickly a time-to-solution can be found to perform the required $2n^3$ double precision floating-point operations. The DGEMM routine is performed in an embarrassingly parallel manner across all processors, with no communication occurring between processes. Each processor performs its own identical version of the DGEMM benchmark and the resulting arithmetic average of each processor's performance represents the final result.

4.1.3 STREAM

STREAM [56, 57] measures the amount of sustainable bandwidth and floating-point double precision performance based on four vector kernels. Each kernel moves data from one location to

another and in some cases performs a floating-point operation on the data. The four kernels used in the benchmark are

- Copy: $a = b$
- Scale: $a = \alpha * b$
- Sum: $a = b + c$
- Triad: $a = b + \alpha * c$.

The Copy and Scale kernels perform operations on two data vectors (a,b) while the Sum and Triad kernels use three data vectors (a,b,c). The Scale and Triad kernels also rely on a scalar value α. The dimension of each vector is determined by the user, but should be larger than the cache size and ideally of size

$$max \begin{cases} 4 \times \text{largest cache size} \\ 10^6 \text{ double-precision elements.} \end{cases}$$

Similar to the DGEMM benchmark, the STREAM benchmark is also performed in an embarrassingly parallel manner with each processor performing its own version of the benchmark on the same data. The results are based on the average obtained across all available processors.

4.1.4 PTRANS

PTRANS is a routine which performs a parallel transpose of a dense matrix. PTRANS is just one routine in a collection of routines known as the PARallel Kernels and BENCHmarks (PARKBENCH) [58]. It tests the communication capacity of the network by moving sections of a large dense matrix between processes. The dimensions of the test matrix and its corresponding block sizes, as well as the distribution scheme of the blocks are determined by the user.

4.1.5 RandomAccess

RandomAccess [59] is a benchmark designed to determine how quickly a system can update randomized memory locations. Each update consists of a randomized address location being read from memory, the data being modified via an integer operation, and then writing the new value back to memory. This benchmark determines the total number of updates per second on the system. Due to the large number of updates, the final result is presented as the number of giga updates per second (GUPS).

4.1.6 FFT

The FFT benchmark performs a one-dimensional fast Fourier transform [60] on a vector of contiguous data containing complex double precision values. The length of the input vector is a power of 2 and determined by the user. The specific implementation of the FFT being used in the benchmark is taken from the FFTE package [61]. This benchmark measures the floating-point rate of execution on double precision complex data located in contiguous memory locations.

4.1.7 Latency and Bandwidth

The latency and bandwidth tests are a modified version of the Effective Bandwidth (b_eff) Benchmark [62]. The latency portion of the test measures the time it takes to send an 8-byte message between processes, while the bandwidth portion measures the time to transmit a two-million-byte message between processes. Each test is performed twice, once using simultaneous communication and once using non-simultaneous communication. These two types of communication methods represent the two extremes of communication that can occur in an application.

4.1.8 lscpu

In addition to the HPC Challenge benchmarks, data is also collected from each system using the Linux command **lscpu**. This command provides information about the CPU of the machine

including its cache sizes, clock speed, and the number of available cores. These features are described in Appendix A.2 along with the features collected using the HPC Challenge benchmarks.

4.2 Computing Matrix Features

When solving a linear system, the features of the coefficient matrix A determine the performance of the solvers and preconditioners that are used to solve the system. These matrix features are a group of easily computable and include common structural properties including properties of the matrix's symmetry and the number of nonzero entries and rows. Numerical features are also computed and include various normative calculations as well as dominance and variance. A full list of the computed features and their descriptions can be found in Appendix A.1.

The Anamod software package [63] is used for computing the individual features of each tested matrix. Anamod computes and records different types of features associated with each matrix. Structural features correspond to features that describe the layout of the matrix as a whole, such as the dimension, number of nonzeros, symmetry, and bandwidth. Simple features are statistics that are easily computable from the matrix, such as its various normative properties. Variance features are those that depict how far the matrix is from an idealized matrix. These include the variance along the diagonal and the row and column variability. Computationally expensive features from Anamod such as the condition number are not included due to their cost.

For some matrices, expensive features such as the condition number, are exceptionally expensive to compute. Because of their size, the matrices discussed in Chapter 8 had their features computed using a PETSc-based approximation of a subset of Anamod, written by the Lighthouse group [64]. The code computes only the subset of features needed from Anamod, while ignoring the more expensive features that are not used in this work.

4.3 Computing Solver Runtimes

The most important part of predicting the best solver-preconditioner pairs for a given linear system, is determining how long previous pairs took to correctly solve the linear system. Timing

data is collected during each solve of a linear system, therefore, times are obtained multiple times for each coefficient matrix A based on all the possible solver-preconditioner pairs available. All the timing data is then combined with other data, such as the features of each matrix, to train a machine learning classifier capable of correlating this other data with the final resulting solve time.

4.3.1 Matrices

The matrices used to create the linear systems used in my experiments are obtained from the University of Florida's (UF) Sparse Matrix Collection [65]. The UF collection of sparse matrices contains thousands of individual sparse matrices with dimensions ranging from 5 to over 100 million and with numbers of nonzero entries ranging from 1 to approximately 2 billion. Matrices from the UF collection are submitted from a variety of users, companies, and institutions, and therefore come from a large collection of real-world applications including structural problems, fluid dynamics, and circuit design. The amount and diversity of the matrices in size, sparsity, symmetry, and application provide a plethora of individual linear system use-cases. A high number of unique use-cases is important in creating a more generalized approach for predicting a high-performing solver and preconditioner combination for an arbitrary matrix. If the matrices were created from the same problem, had the same sparsity layout, or were all symmetric-positive-definite, they would be too similar to each other. Having a collection of similar input data makes it difficult to generalize recommendations to matrices that do not fit within the narrow spectrum of our use-cases.

4.3.2 Software Stack

An important decision when writing numerical code is the underlying software stack that compiles and supports the code. Computer clusters are often populated with optional pre-built binaries, called modules, which are installed and maintained by each cluster's system administrators. These modules often include a variety of compilers (GCC, Intel, PGI), mathematical libraries (Intel's MKL, OpenBLAS), and implementations of MPI (MPICH, MVAPICH, OpenMPI). Since each individual cluster is managed by its own team of administrators it is not common for two clusters to

be populated with the exact same installed software. Even if two machines do in fact have the same software it is unlikely for them to be the same version. Since different compilers produce machine code with varying levels of efficiency and optimization, there can be discrepancies in performance. Not only does the performance of the same application differ based on the compiler being used, but even multiple versions of the same compiler can produce differently performing results. Aside from compilers, these same notions are also applicable to the performance of the MPI and mathematical libraries being used. Unfortunately, since each cluster has its own particular set of software and versions of that software installed, there can be uncontrollable differences between two clusters not only at the hardware level but also at the software level. Minimizing the differences between each of the cluster environments used for running experiments is important in producing repeatable results which are more directly comparable with each other. Since we are unable to change the hardware of a cluster, our only option is to control the software that is used.

In order to address the concern of performance differences between two clusters, I created a self-installing software stack that allows for the same software environment to be built and used, regardless of the system and its default software. The software stack includes a single version of the compiler as well as each library. Controlling the software stack ensures that results obtained from one machine are comparable to those from another machine while being able to ignore any differences between the machines' pre-installed compilers and libraries. Doing this allows for more focus to be placed on the hardware differences between clusters rather than software versions.

A few of the components contained within the software stack are immutable and have no alternatives i.e. Trilinos and Boost, however, other components such as BLAS, LAPACK, MPI, and the compiler itself come in various versions and implementations. When choosing these components, two main things were important to me: they each needed to be supported on the target machines, specifically the MPI implementation, and each component must be free and open-source for all users. Requiring that the components work on the machines being used is self-explanatory. The reason that I chose only open-source options is that it would prevent issues with licensing something like the Intel compiler across multiple machines, and will allow anyone to replicate or expand upon

my work with no paywall. The libraries and compilers that are used as the software stack for my experiments are listed below.

- GNU Compiler Collection [66]

  The GNU Compiler Collection contains a series of open-source compilers. In our software stack it is used for the compilation of Fortran, C, and C++ code.

- OpenMP [38]

  OpenMP is a library that provides support for shared memory parallelism in C, C++, and Fortran code. Through the use of pragma statements, code can run in parallel without the need to manually code lower level threads such as Posix threads.

- MPI [36]

  MPI is a messaging standard responsible for sending messages between distributed processes in high-performance computing. These messages allow for a large distributed program across many nodes and CPUs which can communicate information with each other in order to solve problems too large or impractical for a shared memory approach.

- BLAS and LAPACK [18]

  The Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) are two open-source libraries containing high-performing mathematical functions. The BLAS are created as a series of mathematical "building blocks" which can then be used to create more complex numerical functions. BLAS is based on a three-level hierarchy, where each operation's placement in the hierarchy is based on its computational complexity. Level 1 BLAS functions are responsible for computing vector operations while levels 2 and 3 perform matrix-vector and matrix-matrix operations respectively.

  LAPACK is a series of high-performance routines for solving a variety of numerical linear algebra problems. capable of solving a variety of dense linear algebra problems. Some of the problem types supported by LAPACK include solving systems of linear equations,

determining the eigenvalues of a linear system, and solving the least-squares problem. Each routine available within LAPACK is created using calls to the BLAS library to perform the more simplistic underlying linear algebra operations of the given algorithm.

- Boost [67]

  The Boost library is a large collection of open-source C++ libraries that provide data structures and routines for many disparite areas including regular expressions, graph algorithms, and statistical distributions. Boost is a requirement for compiling Trilinos.

- Trilinos [19]

  Trilinos is a large open-source mathematical and scientific library with many different individual packages addressing a variety of different areas including linear solvers, load balancing, and data structures. The main packages used throughout this work are Tpetra, Belos, and Ifpack2, as discussed in more detail in Section 2.3.

The compilers and libraries are constantly changing and receiving updates. For my experiments the specific versions of the software are listed in Table 4.1.

Table 4.1: The specific versions of compilers and libraries used throughout this dissertation's experiments.

Software	Version
GCC [68]	v6.2
OpenMPI [69]	v1.10.2
OpenBLAS [70]	v0.2.19
Boost [71]	v1.60
Trilinos [72]	v12.8.1

4.4 Data Analysis

This section briefly describes the two main tools used for the data analysis and machine learning methods found throughout this dissertation.

4.4.1 Pandas

Pandas [73] is a popular data science library available for Python. Pandas' primary purpose is to aid the user in data analysis and manipulation through the various data structures and algorithms it provides. For my needs, Pandas serves as the main tool for storing, loading, and analyzing raw data that are being stored from experiments in multiple csv files. Pandas provides many statistical tools for visualizing the breakdown of data into its discrete components, providing key insights into the organization of the data. The primary data structure in Pandas is the DataFrame, a 2D labeled data structure with columns and rows similar to an SQL table. DataFrames can be appended and merged in multiple ways to create larger dimensional DataFrames from multiple existing DataFrames. Pandas is high-performing due to much of the code being written either in Cython [74] or C.

4.4.2 Scikit-learn

Scikit-learn is a popular machine learning library available for Python. It provides support for various types of machine learning approaches including classification, regression, and clustering. Similar to Pandas, Scikit-learn is high-performing due to its internals being written predominately in Cython, C, and C++.

4.5 Matrix Dataset

The matrices used in the experiments described in future chapters were obtained from the University of Florida's Matrix Collection [65] (UF). The UF collection is made up of sparse matrices created from various types of real-world problems. Having matrices from different applications and domain areas is desirable in our case since we are focused on creating a generalized classifier rather than one that is domain specific. For example, a classifier trained only on matrices from a structural finite-element code may not be generalizable to work on other matrices obtained from circuit design.

The matrices selected for these experiments consist of all matrices in the UF collection that

are real, square, and of dimension $\geq$ 1,000. In total, there are 1,586 unique matrices from the UF collection that meet this criteria. Figure 4.2 depicts the distribution of the matrices, chosen from the UF collection, based on their dimension. Figure 4.3 shows the distribution of the selected UF

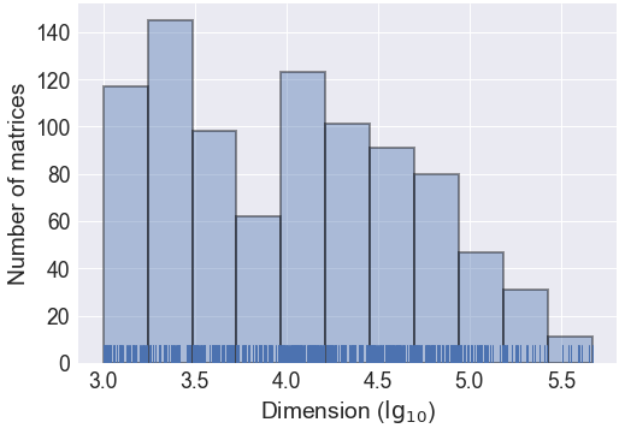


Figure 4.2: Distribution of matrices from the UF collection used in our experiments based on dimension.

matrices based on the number of nonzeros the matrix contains.

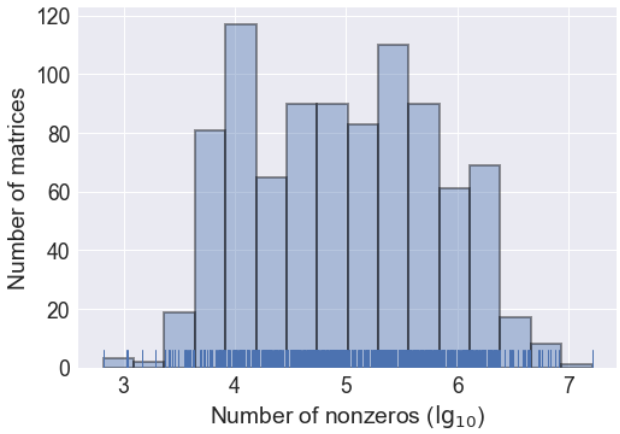


Figure 4.3: Distribution of matrices from the UF collection used in our experiments based on the number of nonzero entries.

Chapter 5

Multicore Prediction

The goal of this dissertation is to examine the prediction performance of incorporating system hardware information into recommending the best performing iterative solvers and preconditioners for solving a given sparse linear system. In this chapter, we examine how solving the linear system with different numbers of CPU cores affects the overall prediction performance of choosing the best solver-preconditioner pair. Existing solver-preconditioner recommendation methods depend on timing data generated by solving linear systems using a fixed number of CPU cores. However, it is ideal for the best solver-preconditioner pair to be determined based on the number of CPU cores available or desired by the user. Therefore, the experiments in this chapter recommend solvers based not only on the features obtained from the coefficient matrix of a user's linear system, but also on the number of CPU cores. Section 5.1 of this chapter presents the methodology used for designing and performing the experiments associated with multicore prediction. Section 5.2 presents the results obtained from the multicore experiments.

5.1 Problem Description

The focus of the work contained in this chapter is to expand upon previous work by incorporating the number of available CPU core counts as part of the recommendation process. Every solver and preconditioner has its own unique memory movement, floating-point arithmetic, and message passing. Solvers and preconditioners are often run in parallel, but there also exist cases where a given linear system is solved using just a single core. Each solver and preconditioner's

unique performance characteristics make their overall serial and parallel performance difficult to predict **a priori** when solving a linear system. Choosing the best solver-preconditioner pair for a single threaded application on a modest personal computer may not be the best choice when running in parallel on a node of a bleeding-edge supercomputing cluster.

In general, to predict if a solver-preconditioner pair is a good choice for solving a linear system created from a given coefficient matrix A, there are four pieces of data that must be computed for each training matrix and then combined together to represent a singular linear solve:

(1) The matrix's numerical and structural features

(2) Time to solve each matrix's linear system using each solver and preconditioner combination

(3) Whether the solver-preconditioner pair is "good" or "bad" for solving the given linear system

(4) The number of CPU cores used to solve the linear system.

Existing work is based on measuring the wallclock time of solving a sparse linear system with a given solver-preconditioner combination and then coupling all the resulting wallclock times with the unique features derived from the coefficient matrix. By associating certain numerical properties of each matrix with its corresponding runtime at varying number of processors, it is possible to form a basic understanding of how well a classifier can predict good solver-preconditioner pairs across multiple possible core counts. Adding this level of hardware awareness as part of the recommendation system, allows for more accurate recommendations, which are more specifically tailored for an end user.

When solving a linear system, the properties of the coefficient matrix A determine how well a given solver and preconditioner combination performs when solving the linear system. For each matrix, a list of features is computed that detail certain numerical and structural attributes. These attributes are easily computable and include common structural properties such as the matrix's symmetry, the number of nonzero entries, and the dimensions of the matrix. Numerical features are

also computed and include various normative calculations, as well as dominance and variance. A full list of all the computed matrix features and their descriptions can be found in Appendix A.1.

After each matrix's features have been determined, the matrix's runtimes are then computed. The runtimes, or wallclock times, of each matrix are defined to be the total time it takes to solve the matrix with each of the available solver-preconditioner pairs. Computing the wallclock times for each matrix involves solving the system $Ax = b$ where A is the coefficient matrix currently being tested, $b = \mathbf{0}$, and x is unknown. Not every preconditioner or solver will be capable of solving each of the presented linear systems. Some solver-preconditioner pairs will fail to find a solution within the maximum number of iterations while other solver-preconditioner pairs may not be applicable to the given matrix. For instance, the conjugate gradient algorithm requires the matrix to be both symmetric and positive-definite [1]. Rather than being ignored completely, these solver-preconditioner pairs are classified as taking an infinite amount of time to find a solution. Any solver-preconditioner pair that fails to run or fails to converge to a solution within the specified number of iterations is automatically classified as being a "bad" solver-preconditioner pair to use for solving a linear system with that particular coefficient matrix.

All numerical runtime experiments in this dissertation are performed using the Trilinos framework [19]. The matrix and vector objects are created from Trilinos's Tpetra package [35], while the solvers and preconditioners are provided from the Belos [42] and Ifpack2 [44] packages, respectively. The preconditioners used from the Ifpack2 package are

- Chebyshev Polynomial
- Jacobi Method (Relaxation)
- Relaxed ILU Factorization w/ k Fill (RILUK)
- ILU Factorization w/ Thresholding (ILUT).

The iterative linear solvers used from the Belos package are

- Biconjugate Gradient Stabilized (BiCGSTAB)
- Conjugate Gradient (CG)

- Fixed Point
- General Minimal Residual (GMRES)
- LSQR
- Minimal Residual (MINRES)
- Transpose-Free Quasi-Minimal Residual (TFQMR).

The solvers and preconditioners mentioned above remain the same throughout the experiments contained in this dissertation. General algorithm details for the solvers and preconditioners are described in Chapter 2, while specific implementation details are available in their respective online documentation.

Each linear system is solved using each of the available solver and preconditioner combinations that can be created from the available options, so long as the linear system satisfies the basic requirements of the given solver and preconditioner. There is also a "no preconditioner" option included within the experiments, which simply runs each of the solvers without any preconditioning. All the Ifpack2 preconditioners are run with their original and default parameters and options. The Belos solvers are run with their own respective default parameters and options except for the maximum number of iterations and convergence tolerance, which are set to 10,000 and 10^{-6} respectively.

Each of the linear system solves are executed on a single node of the University of Colorado's Summit cluster [75] at varying numbers of processors, $np = \{1, 4, 8, 12, 16, 20, 24\}$. The experiments in this chapter were all performed on Summit. The hardware specifications of each Summit node are briefly described in Table 5.1, while a more thorough description of the hardware contained within Summit is available in Table B.4. The software stack containing the compilers and libraries used for running the experiments is depicted in Table 5.2.

The total wallclock time required to solve each linear system with each of the solver-preconditioner pairs is then recorded and stored. Combining the resulting timing information with a matrix's features allows for correlations to be found between matrix features and the re-

Table 5.1: Hardware specifications for the standard Summit node

Hardware	Specifications
Processors	2x Intel Xeon E5-2680 (12 cores ea.)
Memory	128GB @ 2133MT/s
Interconnect	Intel Omni-Path

Table 5.2: The specific versions of compilers and libraries used throughout this dissertation's experiments.

Software	Version
GCC [68]	v6.2
OpenMPI [69]	v1.10.2
OpenBLAS [70]	v0.2.19
Boost [71]	v1.60
Trilinos [72]	v12.8.1

sulting performance of solvers and preconditioners on that matrix. These correlations can then be generalized by finding patterns and connections between multiple matrices with similar features and how those features impact the performance of the various solver-preconditioner pairs.

Although it is easy for each failed solve to be classified as a "bad" solver-preconditioner pair, the classification is made more complicated when the pair converges to a solution. After a linear system has been successfully solved, the solver-preconditioner pair now has to be determined as being either "good" or "bad" in its overall performance. Determining if a solver-preconditioner pair should be chosen for a given linear system is an ultimately subjective decision since there can be any number of possible ways to determine success in this context. Ideally, each recommended "good" solver-preconditioner pair would guarantee not only convergence, but also guarantee to find a solution faster than all other available pairs. However, it is extremely likely for one or more solver-preconditioner pairs to be able to solve a given linear system within an acceptable time-difference from the fastest solver-preconditioner pair. Therefore, for this experiment, any solver-preconditioner pairs that are high-performing, but not necessarily the fastest, can also be

accepted as long as they perform within some percentage of the fastest pair's time.

After recording solver timings, collecting matrix features, and classifying each matrix-solver-preconditioner combination as either "good" or "bad," Scikit-Learn [76] is used to create a classifier based on this information. There are multiple binary classification methods available in Scikit-Learn, eight methods were chosen and tested to determine which of the many available methods should be used for the totality of our experiments. The binary classification methods that were explored and tested in the preliminary experiments are listed in Table 5.3. Each of the binary classifiers

Table 5.3: The binary classifiers from Scikit-Learn examined in the experiments of this chapter. [76, 77].

Classifier Name	Classifier ID
Gradient Boosting	0
Random Forest	1
Gaussian Naive Bayes	2
Decision Tree	3
Logistic Regression	4
Multilayer Perceptron	5
AdaBoost	6
K Nearest Neighbor	7

were tested using the same representative test dataset taken from the larger, complete matrix dataset described in Section 4.5. All of the classification methods were trained and tested using three-fold cross-validation methods. A three-fold cross-validation divides the dataset into three "splits" with the classifier training on one of the splits and then predicting the classifications based on the non-labeled data of the other two splits; this process is repeated such that each split is used as the training data once. The ROCs, as well as the AUROCs, are computed for each split and then averaged together to determine the overall performance of each classification method. Cross-validation is used to help avoid overfitting the classifiers to specific data; overfitting can result in classifiers that do not generalize well to new types of information.

One of the critical components of this research is developing and creating methods capable of determining the success of current prediction methods when the number of cores used for solving

a linear systems differs between the training and testing data. As an example, the diagram in Figure 5.1 shows the difference between existing methods, which train and test on the same fixed number of processors, while the methods presented in this work are trained and tested on multiple numbers of processors.

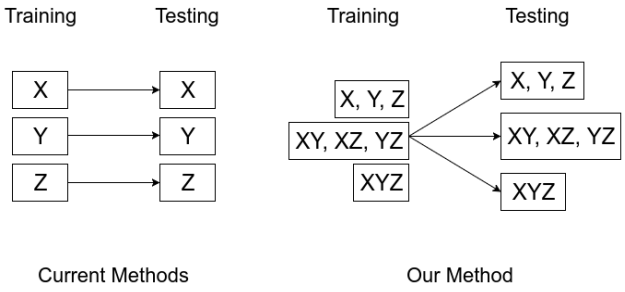


Figure 5.1: Example comparison of our method to existing prediction methods. Each variable X, Y, Z represents the "good" and "bad" solver-preconditioner data at a single processor count. Each left column represents the data which is used to train the classifier, while the right columns represent the data points on which the classifier is tested. Our method is capable of training and testing on any possible permutation of the number of processors in order to examine and improve prediction performance.

By training on a single fixed processor size and the corresponding "good" and "bad" labels that go with it, it is possible to determine how well the results can be generalized to other processor counts. The generalization can be accomplished by training the data as normal on one or more processor counts and then testing the classifier's predictions on other processor counts. By including multiple processor counts into the training and testing data, the relative "good" and "bad" solver-preconditioner pairs change as more information is added and includes or excludes previous "good" and "bad" classification labels.

By training and testing the resulting classifiers on groups of "good" and "bad" results from different numbers of processors it is possible to determine how well the classifier is able to use information from one set of data and use it to predict for another. For example, training the classifier using the wallclock times obtained from running the linear systems in serial, and then testing the resulting classifier to predict the best solver-preconditioner pairs at a different number of processors. By measuring the prediction accuracy using multiple testing and training datasets it is possible

to determine how well a given training dataset extrapolates out to other core counts. This data determines how well the new classification methods, as well as the existing classification methods, perform when used with training and testing data which are not necessarily the same, or at only one specific fixed-core count.

Due to the imbalanced ratio presented between the number of instances of "bad" solver-preconditioner pairs compared to "good" solver-preconditioner pairs, there is not much value to be had in measuring the pure accuracy of the machine learning predictions. Accuracy is a common metric for more evenly distributed datasets, and can be determined using the formula

$$Acc = \frac{TP + TN}{P + N}$$

which represents the total number of true-positive (TP) and true-negative (TN) predictions divided by the total number of positive and negative instances in the data. Accuracy is not an ideal measure of success for imbalanced datasets because a classifier can have a high accuracy by always classifying input, regardless of what it is, into the more prevalent of the two classes. For instance, if only 1% of the data consists of output belonging to the X class and the other 99% belong to the Y class, a classifier can simply choose the Y class in every case regardless of input and have an overall accuracy of 99%. Therefore, accuracy is not an ideal metric for success with such an imbalanced dataset. Instead, I use the receiving operating characteristic (ROC) [78] as the measurement of how well the the binary classifiers perform. The ROC is a curve which plots the true positive rate against the false positive rate at different thresholds. An example of how an ROC curve is generated from the true-positive and true-negative instances of a classifier is depicted in Figure 5.2.

A popular metric, which can be obtained from a given ROC curve, is the area under the receiver operating characteristic (AUROC) [80]. The AUROC is a scalar value equal to the integral of an ROC curve. The AUROC represents how well the classifier can accurately classify two input instances, each from one of the output labels, into their respective outputs. Therefore an AUROC of 1.0 is equivalent to a perfect classification method that is always capable of choosing the correct class for an input instance. Conversely, an AUROC of 0.5 is equivalent to randomly guessing when

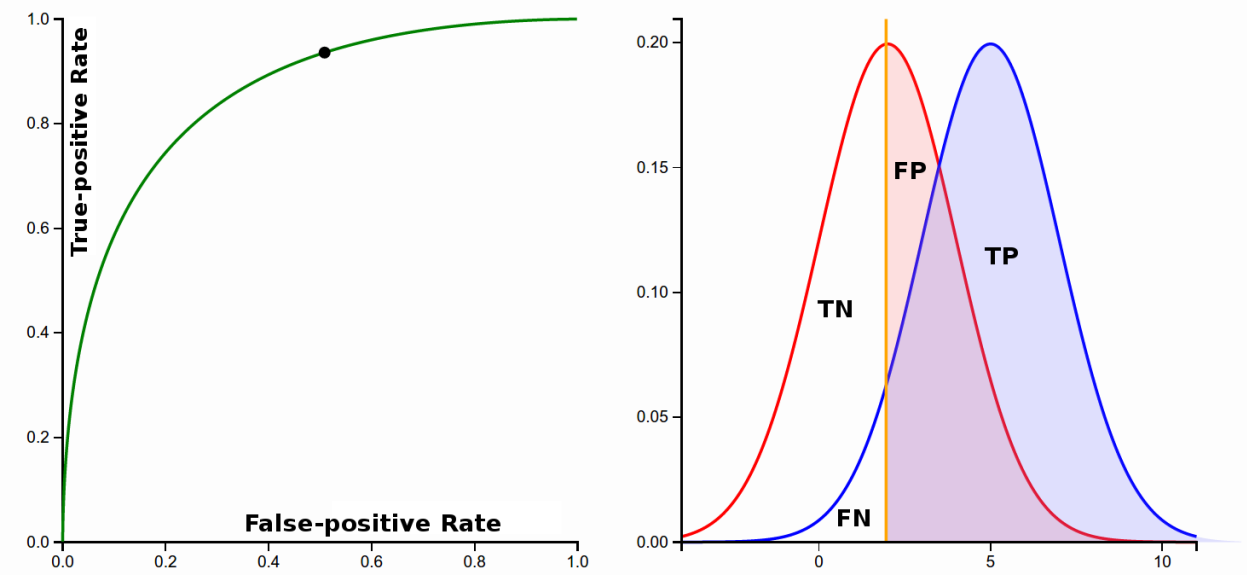


Figure 5.2: An example ROC curve, created from plotting the true-positive rate against the false-positive rate at various thresholds. The threshold at the given time is depicted by the vertical line in the image on the right [79]. The rightmost image also shows the proportions of the results where TP = true positive, FP = false positive, TN = true negative, and FN = false negative.

classifying any given input. An AUROC of 0.5 is the minimum attainable value since anything less can simply be flipped by swapping whatever outputs are given by the classifier, resulting in a more accurate classification scheme.

When designing a classification scheme that needs to be trained on an imbalanced dataset, the training data can be skewed due to the small number of one class being dominated by the other. This has resulted in a variety of sampling methods that attempt to give more representative information between the two by generating multiple instances of the input data corresponding to the underrepresented output class [81]. Another common solution that addresses this problem is to use stratification methods to ensure that each subset of data being trained contains a representative number of entries from each of the two possible classes [77, p. 559]. Stratification is often used in conjunction with k-fold cross-validation to obtain folds that each contain representative instances for each output class.

Because many solver-preconditioner pairs are either not applicable to a given matrix, or because they take too much time to converge to a solution, the dataset, as a whole, is imbalanced

and heavily biased towards most solver-preconditioner pairs being "bad." The number of "good" pairs compared to the number of "bad" pairs at the various CPU counts can be see in Figure 5.3. In order to combat the severe imbalance in the dataset, the cross-validation folds are created using the 'stratified shuffle split' method [82], which returns randomized folds that are representative of their respective classes.

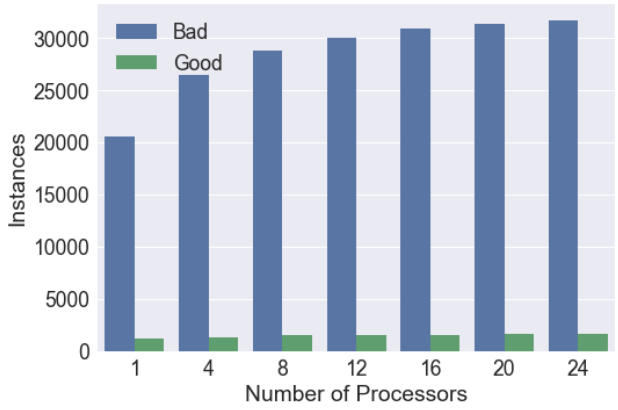


Figure 5.3: Comparison of all solver-preconditioner pairs considered to be "good" and "bad" at various CPU counts with all pairs within 25% of the optimal pair considered to be "good"

5.2 Results

The first experiment that the rest of the experiments depends on is choosing which of the many binary classifiers to use for the rest of the experiments. A comparison of the classifiers' success on the test data is depicted in Figure 5.4 and shows the Random Forest, Decision Tree, and K Nearest Neighbor classifiers performed the best overall. Ultimately, the Random Forest classifier was chosen as the best binary classifier to use in this case due to its relatively lower variance when compared to the other classifiers with high AUROC scores.

The remaining experiments and results for this chapter can be broken up into two major groupings:

(1) Determining how well classifiers, trained on data from a singular processor count, perform when tested on other singular processor counts.

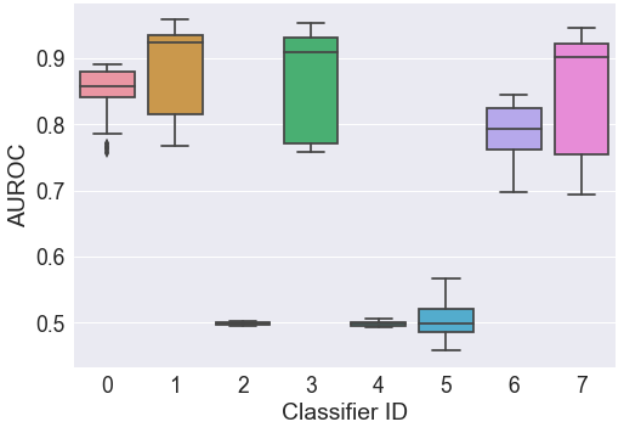


Figure 5.4: Classification performance comparison of the eight machine learning methods tested. Each of the ID numbers corresponds to the information provided in Table 5.3

(2) Determining how well classifiers, trained on data from multiple processor counts, perform when tested on singular processor counts.

The graphs of the ROC curves plot are named based on an X_Y numbering system where X represents the processor counts used in computing the training data and Y represents the processor counts used in computing the testing data.

5.2.1 Training on Single Core Count Performance

This section presents four ROC curve graphs depicting the prediction performance of the Random Forest classifier when trained on data from a single processor count of $np = 1, 4, 12, 24$ in Figures 5.5 to 5.8, respectively. These ROC graphs depict the individual performance of the classifier when trained on one core count and then used for predicting another core count. An ideal ROC curve is a perfectly horizontal line at $y = 1.0$, and having a perfect AUROC of 1.0. Therefore, the success of each classifier's performance is determined by which ROC curves have the largest area under their curves. Each of the figures in this section depicts the ROC itself as well as each curve's total area under the curve (AUROC).

Figure 5.5 depicts the most common example found in the current literature, a classifier trained solely on data obtained from solving linear systems using a single computational thread.

This graph illustrates how a serially trained classifier performs on various processor counts. The ROC curves in Figure 5.5 demonstrate that the classifier is highly successful when predicting solver-preconditioner pairs also running serially, but that the prediction performance suffers as the predictions are made on incrasingly larger numbers of cores. Here, the largest AUROC difference, 0.11, is found between $np = 1$ (AUROC=0.93) and $np = 20$ (AUROC=0.82).

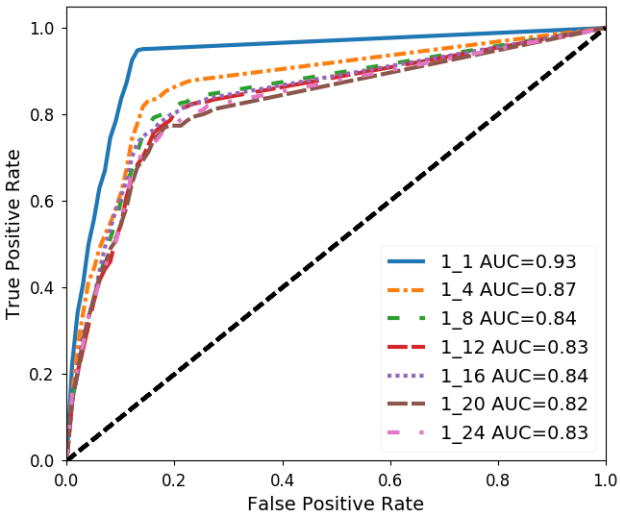


Figure 5.5: ROC curves obtained from training on data of $np = 1$ and testing the resulting classifier on runtime data generated for each of $np = 1, 4, 8, 12, 16, 20, 24$. The higher curve associated with 1_1 indicates that the prediction performance of the classifier is best when trained and tested on the same data and degrades when used on different numbers of cores.

Figure 5.6 is similar to Figure 5.5, but depicts a classifier trained on data from solving linear systems on 12 cores rather than 1. The prediction performance drop in this graph is more noticeable with a larger difference of 0.17 between $np = 12$ (AUROC=0.95) and $np = 1$ (AUROC=0.78). Here we can see that although there is fairly consistent prediction accuracy at the higher number of processors, there is a large performance loss when training on 12 processors and testing on data obtained from running serially.

Figure 5.7 is similar to the two previous figures, and depicts a classifier trained on data from solving linear systems on 24 cores. The prediction performance drop in Figure 5.7 is similar to that of Figure 5.6, with a difference of 0.17 occurring between $np = 24$ (AUROC=0.96) and $np = 1$ (AUROC=0.79). The similarities between Figures 5.6 and 5.7 indicate that the performance change

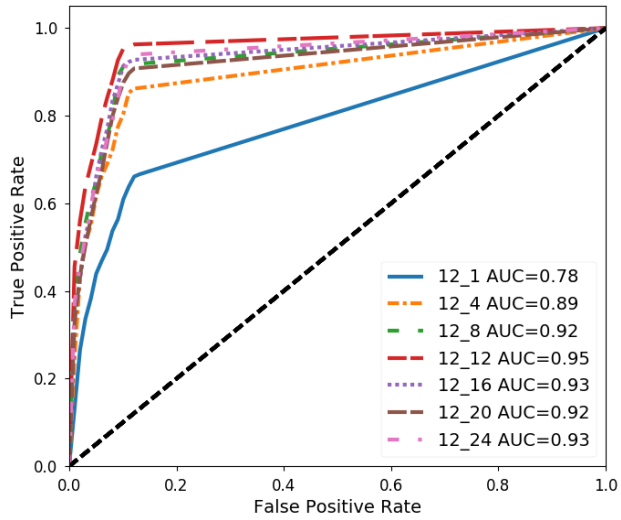


Figure 5.6: ROC curves obtained from training on data of $np = 12$ and testing the resulting classifier on runtime data generated for each of $np = 1, 4, 8, 12, 16, 20, 24$. The lower curve associated with 12_1 indicates that training the classifier on solving linear systems on 12 cores performs worse when trying to predict the best solver-preconditioner pairs for systems being solved serially.

between solver-preconditioner pairs is more noticeable when changing the training core count from 1 to 12 than from 12 to 24. When training on the training data generated at both 12 and 24 cores, the prediction performance remains high ≥ 0.89 in both cases for all data tested on more than one core.

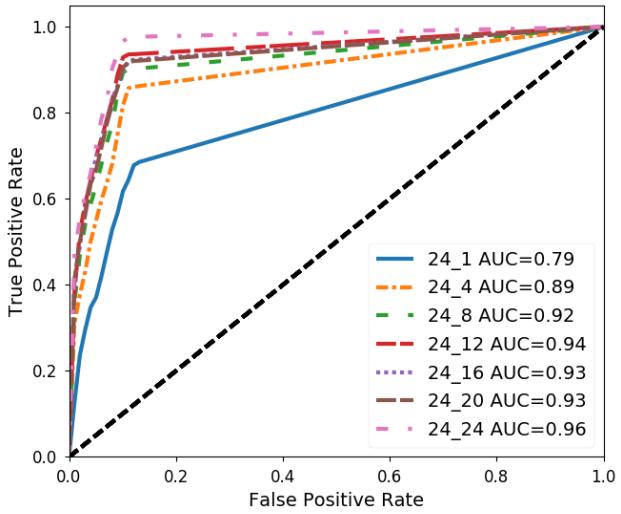


Figure 5.7: ROC curves obtained from training on data of $np = 24$ and testing the resulting classifier on each of $np = 1, 4, 8, 12, 16, 20, 24$.

Figure 5.8 is unique compared to Figures 5.5 to 5.7 because there are three fairly distinct

groupings of plots as opposed to two in the other figures. The figures with two plot groupings have one group containing the majority of plots and then another group containing a singular outlier plot. In Figure 5.5 the outlier (1_1) performs better than the rest of the plots, which are more closely grouped together. Figure 5.6, on the other hand, shows the outlier (12_1) having worse performance than the other graphs which are closely grouped together. In contrast, Figure 5.8 has both a higher (4_4) and lower (4_1) performing outlier on each side of the larger grouping of plots in the middle.

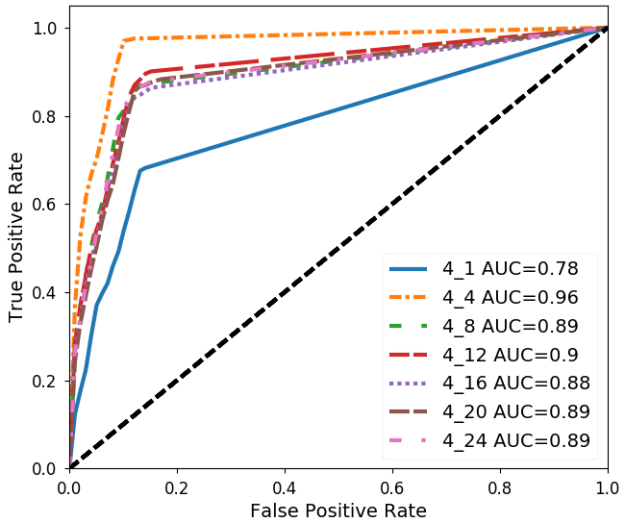


Figure 5.8: ROC curves obtained from training on data of $np = 4$ and testing the resulting classifier on runtime data generated for each of $np = 1, 4, 8, 12, 16, 20, 24$.

The results indicate that prediction performance decreases when training data taken from the linear solve times of one core count is used to predict the performance of another core count. The best prediction performance in all four graphs, Figures 5.5 to 5.8, occurs when the core count of the training data matches that of the testing data. All four figures show that the prediction performance is highest when training and testing on data from the same number of processors. Even at similar processor counts, the solver-preconditioner pairs that are "good" or "bad" for a given linear system can differ. The rest of the mismatched training and testing core counts do not perform as well in varying degrees, based on how distant the two core counts are from one another. There is a measurable loss in prediction performance, based on the AUROC, when the classifier is trained on runtime data generated using one processor count that is then tested on another processor count.

Therefore, we cannot simply extrapolate the results of one processor count to those of others and expect performance to remain the same. Figures 5.5 to 5.8 show that after changing to a parallel algorithm, the performance of the solvers and preconditioners become more similar as the number of cores increases.

Section 5.2.1 depicts all of the AUROC's obtained from the single-to-single core count experiments. Each of the best scores, unsurprisingly, uses the same dataset generated from the same core count. The table also clearly shows the largest difference in relative AUROC scores are those associated with either training or testing on data collected from one core.

	1	4	8	12	16	20	24
1	**0.93**	0.87	0.84	0.83	0.84	0.82	0.83
4	0.78	**0.96**	0.89	0.90	0.88	0.89	0.89
8	0.75	0.89	**0.96**	0.90	0.92	0.91	0.92
12	0.78	0.89	0.92	**0.95**	0.93	0.92	0.93
16	0.77	0.89	0.93	0.93	**0.95**	0.92	0.93
20	0.78	0.87	0.92	0.93	0.93	**0.95**	0.92
24	0.79	0.89	0.92	0.94	0.93	0.93	**0.96**

Table 5.4: The AUROC results obtained from creating a classifier by training on the data generated from a single core count and testing the classifier's performance on another single core count dataset. The left-most column indicates the number of cores used to generate the training information. The top-most row indicates the number of cores on which the classifier was tested. The entries in bold indicate the best AUROC score for each individual classifier.

5.2.2 Training on Multiple Core Counts Performance

Unlike the ROC graphs depicted in Section 5.2.1, Figures 5.9 to 5.12 depict the ROC curves obtained from training the classifier on data collected from two or more CPU cores, $np = 1, 4, 8, 12, 16, 20,$ and 24. The resulting multi-core classifier is then tested on data obtained from the single core counts.

Figure 5.9 shows the prediction results of training on data from all of the available cores and then testing the classifier on data for each core. The resulting ROC curves and resulting AUROCs are significantly higher than the previous graphs created from training on single core data. Because

the classifier is trained on each of the core counts, it produces results similar to those in Section 5.2.1 where the training core size and testing core size are equivalent. This result is to be expected since the classifier has been trained on all of the available cores individually The classifier correlates the input core count with the existing data at that core count.

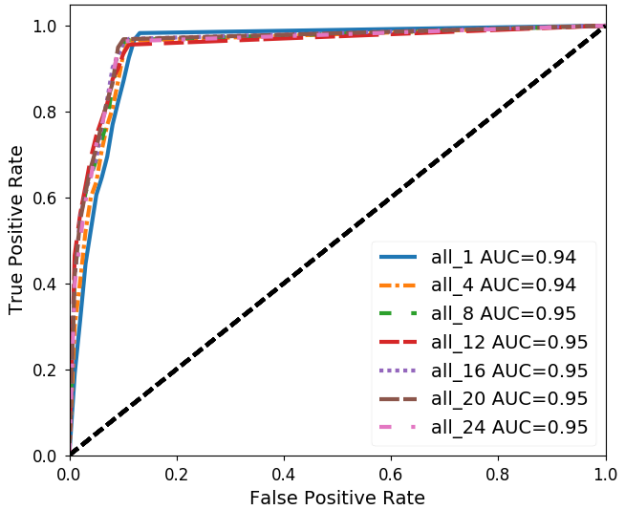


Figure 5.9: ROC curves obtained from training on data from all possible core counts, $np = \{1, 4, 8, 12, 16, 20, 24\}$, and testing the resulting classifier on each core count of $np = \{1, 4, 8, 12, 16, 20, 24.\}$

Figure 5.10 depicts the ROC curves obtained from training on all of the possible core counts except one, the resulting classifier is then tested on the left out core count. Overall, the AUROC scores are lower than those seen in Figure 5.9, especially at $np = 1$ and $np = 4$. This difference in performance shows that the absence of training data at a lower core count, especially 1, is more costly than data from a higher core count. Because this difference is much larger at small core counts, it implies that the performance of solvers and preconditioners has more uniqueness at smaller core counts than at higher core counts.

Because the results in Figure 5.10 suggest a split between the prediction performance associated with small core counts $np = 1, 4$ and larger core counts, $np = 8, 12, 16, 20, 24$, an experiment was performed to determine the prediction performance when training on only the two extremes of $np = 1$ and $np = 24$. The experimental results in Figure 5.11 shows the results of training at the

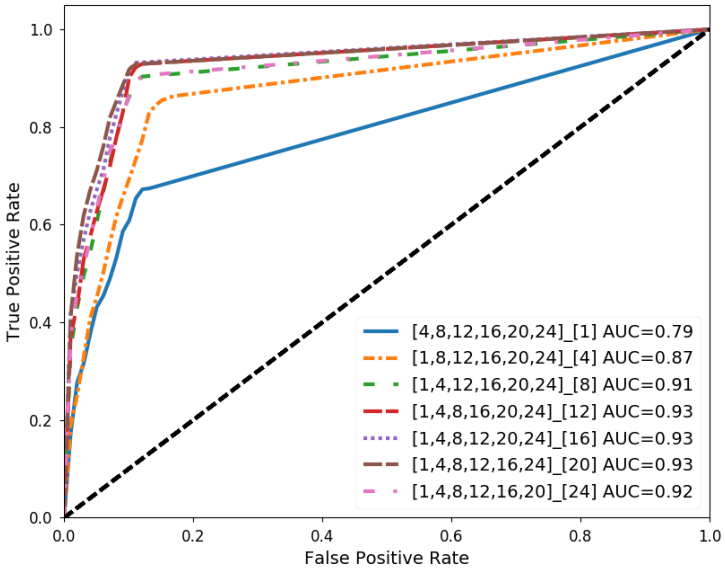


Figure 5.10: ROC curves obtained from training on data from all possible core counts, $np = \{1, 4, 8, 12, 16, 20, 24\}$, except the one core count it is being tested against.

two extremes. The final results show that the higher core counts tend to be high, 0.85+, and also increases the prediction performance at the previously weakest predictor $np = 1$ from 0.79 to 0.93. These results also show that it's possible to obtain ≥ 0.85 AUROC for each of the core counts, using only two of the possible seven data collection runs, resulting in using only 28.5% of all the data collected and used in Figure 5.9.

Figure 5.12 contains the results of the final multicore experiment, where the classifier has trained on the two extremes from the previous experiment, $np = 1, 24$, while adding in the middle core count of $np = 12$. Adding in $np = 12$ was done in an effort to examine the performance difference between the previous experiment and adding in more information in between the two extremes. This experiment shows that but with still significantly less required information than using all possible core counts.

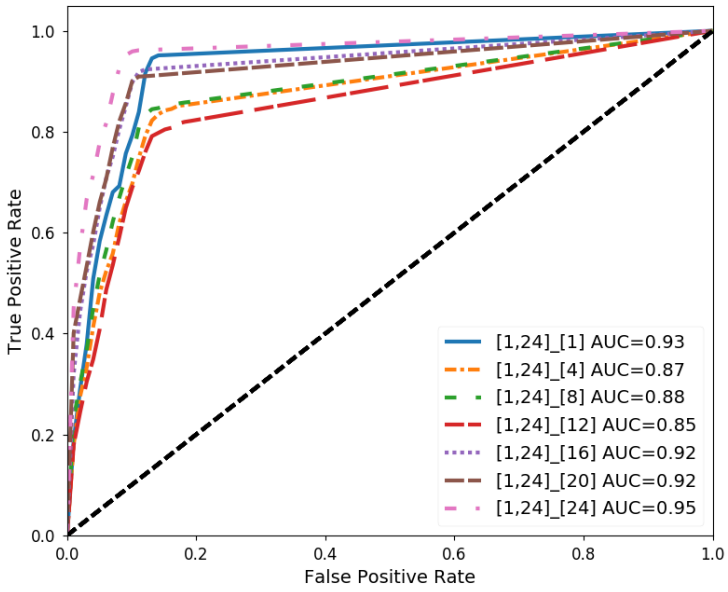


Figure 5.11: ROC curves obtained from training on data of $np = \{1, 24\}$ and testing the resulting classifier on each of $np = \{1, 4, 8, 12, 16, 20, 24.\}$

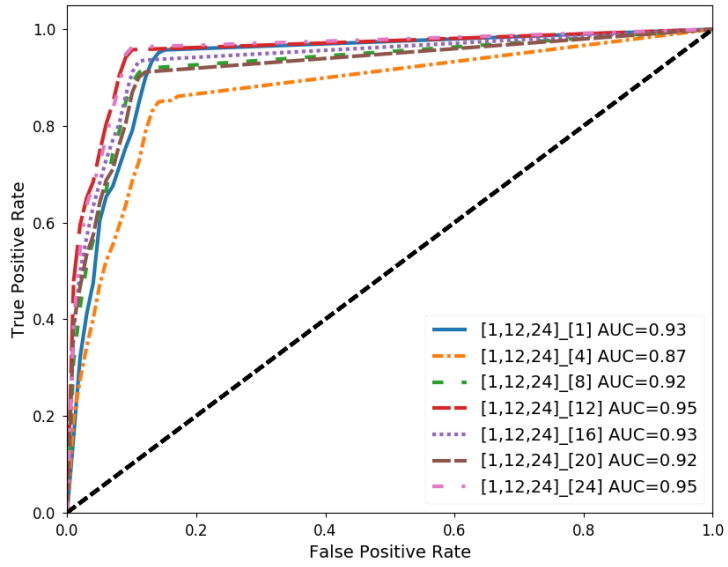


Figure 5.12: ROC curves obtained from training on data of $np = \{1, 12, 24\}$ and testing the resulting classifier on each of $np = \{1, 4, 8, 12, 16, 20, 24.\}$

5.2.3 Feature Selection

Table 5.5 shows the relative importance for each of the matrix-features and the solver and preconditioner. The feature rankings are created using the randomized lasso method [50], which uses a randomization method to select subsets of both the dataset and the features included. It then measures what percentage of the total number of tests selects the feature as an important feature in determining the final result of "good" or "bad." Therefore a score of 1.0 indicates a feature that was selected 100% of the time during testing and determined to be important to the overall classification.

Unsurprisingly, the solver and preconditioner being used to solve the linear system were selected in nearly every instance as being an important factor in the prediction. Column diagonal dominance was also selected in all cases, indicating that the value of the diagonal entries, compared to the other entries in the column, contribute a great deal to a solver-preconditioner pair being a "good" choice. Of the top eleven features (≥ 0.87): four relate to the amount of nonzero entries contained within the matrix and two relate to the rows containing only a single nonzero entry. The number of CPU cores used to solve the linear system was also selected in all test cases, indicating that there is enough of a measurable difference between the performance of the solver-preconditioner pairs at each of the np values. This shows that current prediction schemes result in high prediction performance when recommending solver-preconditioner pairs when the user is solving systems with the same number of cores used to generate the classifier's training data. However, if the user plans to use a different number of cores to solve the system than was used to generate the classifier's training data, the prediction performance measurably degrades at the single-node scale.

Multicore prediction is only one aspect of incorporating hardware information as part of the solver-preconditioner pair recommendation process. The next chapter tackles the problem of more than one type of computer hardware, while Chapter 7 combines these two ideas into a single problem. Finally, Chapter 8 tests the resulting classifiers' performances when tasked with predicting the best solver-preconditioner pairs for an unseen real-world problem.

Table 5.5: Features, ranked by their importance, using the Randomized Lasso algorithm for the features used in Chapter 5. More detailed descriptions of each feature can be found in Appendix A.1

.

Feature	Importance	Description
col_diag_dom	1	Comparison of diagonal entry with other entries in the column
np	1	Number of CPU cores used
solver_id	1	Choice of iterative solver
dummy_rows_kind	0.995	Value types of rows w/ one nonzero entry per row
prec_id	0.99	Choice of preconditioner
min_nnz_row	0.975	Minimum number of nonzeros in a single row
avg_nnz_row	0.965	Average number of nonzeros in a single row
lower_bw	0.96	Lower bandwidth
dummy_rows	0.955	Number of rows w/ only one nonzero entry
max_nnz_row	0.92	Maximum number of nonzeros in a single row
diag_nnz	0.87	Number of nonzero diagonal entries
col_log_val_spread	0.695	Max ratio between a column's minimum and maximum entries
antisymm_inf_norm	0.61	The infinity norm of $A - A^T/2$
num_value_symm_2	0.585	Percentage of nonzero entries a_{ij} with nonzero entries at a_{ij}
nnz_pattern_symm_2	0.49	Soft numeric value symmetry
upper_bw	0.49	Upper bandwidth
trace	0.435	Sum of all diagonal entries
row_log_val_spread	0.415	Max ratio between a row's minimum and maximum entries
col_var	0.37	Largest of each column's variance
num_value_symm_1	0.37	Hermitian property of matrix
abs_trace	0.34	Sum of the absolute values of the diagonal entries
symm_inf_norm	0.31	Infinity norm of $A + A^T/2$
diag_var	0.285	Variance of the diagonal entries
diag_sign	0.255	The types of values stored on the diagonal
nnz	0.215	Total number of nonzero entries
row_var	0.21	Largest of each row's variance
one_norm	0.205	One norm
inf_norm	0.15	Infinity norm
nnz_pattern_symm_1	0.14	If all nonzero entries a_{ij} have nonzeros at a_{ji}
row_diag_dom	0.13	Comparison of diagonal entry with other entries in the row
diag_avg	0.095	Average of the diagonal entries
antisymm_frob_norm	0.08	Frobenius norm of $A - A^T/2$
symm	0.075	If $A = A^T$
frob_norm	0.04	Frobenius norm
symm_frob_norm	0.04	Frobenius norm of $A + A^T/2$
rows	0.025	Number of rows
cols	0.02	Number of columns

Chapter 6

Multi-system Prediction

The goal of this dissertation is to examine the prediction performance of incorporating system hardware information into recommending the best performing iterative solvers and preconditioners for solving a given sparse linear system. This chapter examines how solving a linear system with different computer hardware affects the overall prediction performance of choosing the best solver-preconditioner pair. Existing solver-preconditioner recommendation methods depend on timing data generated by solving linear systems using a single computer system. However, it is ideal for the best solver-preconditioner pair to be aware of a user's hardware and adjust its recommendations accordingly. Therefore, the experiments in this chapter recommend solvers based not only on the features obtained from the coefficient matrix of a user's linear system, but also on the hardware being used to solve the linear system Section 6.1 of this chapter describes the methodology of designing and performing the experiments associated with multi-system prediction. Section 6.2 presents the results obtained from these multi-system experiments.

6.1 Problem Description

The work contained within this chapter focuses on recommending solver-preconditioner pairs for sparse linear systems, similar to the previous chapter, but attempts to solve a slightly different problem. Chapter 5 focuses on examining the prediction performance of using a variable number of cores within a single computer system to solve sparse linear systems. This chapter, instead, focuses on measuring how differences in computer hardware impact the overall prediction performance of

classifiers created from training data obtained only on one system. Current work on the prediction of iterative solvers and preconditioners for solving sparse linear systems is limited to creating classifiers based on training and testing data generated from solving linear systems on a single machine on a fixed number of cores. This chapter is focused on experiments that keep the number of cores used to solve the linear systems fixed, while changing the computer systems used to generate the training and testing timing data.

Every solver and preconditioner implementation has its own unique algorithm, therefore, each solver and preconditioner has its own unique memory movement, floating-point arithmetic, and message passing. Due to the differences in hardware architecture and their impact on overall performance, it is not expected that algorithms behave or perform identically across systems. Therefore, it is important to understand how prediction performance changes when using training data generated from multiple computer systems in order to give more tailored and accurate results to the end user. Since different computer systems have different performance capabilities based on their hardware components, it is possible that the best performing solver-preconditioner pairs differ between computer systems, when solving the same linear system. For instance, a system with a faster CPU may theoretically perform more floating-point operations per second (FLOPs) than a machine with a slower CPU. However, it is also possible for the slower CPU to be of a different architecture that allows for better vectorization or larger cache sizes, allowing it to perform a higher number of FLOPs than the faster CPU in certain situations. Many other aspects of the hardware, such as the latency and bandwidth between the CPU and main memory, are also important to the system's overall performance. The unique properties associated with each solver and preconditioner make their performance difficult to predict beforehand when solving a linear system. The best solver-preconditioner pair for solving a linear system on a personal laptop may not be the same as one being solved on a workstation or cluster due to the differences in clock speeds, bandwidth, cache sizes, etc.

Predicting whether or not a given solver-preconditioner pair will perform well on an arbitrary linear system, as well as an arbitrary computer system, requires four main data points:

(1) The wallclock time to solve the linear system using various solvers and preconditioners

(2) The numerical and structural features of the coefficient matrix

(3) What solvers and preconditioners are best at solving certain types of linear systems

(4) Hardware performance information from the computer systems solving the linear systems.

Existing works are based on recording the wallclock time of solving a sparse linear system with a given solver-preconditioner combination on a single computer system. The resulting timing data, associated with each matrix, is coupled with features derived from the coefficient matrix. These matrix features provide insight to its layout and numerical properties. By associating the numerical properties of each matrix with its corresponding runtime on different types of hardware, it is possible to form a basic statistical correlation that associates hardware specifications, coefficient matrix information, and each solver-preconditioner pair with the wallclock time taken to solve the linear system. Adding in this level of hardware awareness to a recommendation system allows for more specific recommendations, which are tailored not only to an end user's unique coefficient matrix, but also their unique computer system.

Aside from the hardware information, the experiments in this chapter largely follow the same methodology described in Chapters 4 and 5. Much of the large components as well as the code used to perform the experiments themselves remain the same. The coefficient matrices for the linear systems being solved are the same, as are the solvers and preconditioners. The main differences are the collection and introduction of hardware information, solving the linear systems at a fixed number of cores, and

Due to the modest performance of my personal laptop, only the 700 smallest matrices, by storage size, were used for solving linear systems on it. All other machines solved linear systems with the entire 1,586 matrix dataset obtained from the University of Florida's Sparse Matrix Collection.

The critical component of this chapter's research is exploring methods for determining how well current prediction methods perform when the hardware used for solving the linear systems

is not necessarily the same as the hardware used to originally solve the classifier's training data. Existing linear solver recommendation classifiers are created from data obtained from the total wallclock time it takes for each solver-preconditioner pair to converge when solving a given linear system on a single computer system using a fixed number of CPU cores. As in the last chapter, the "good" and "bad" monikers are given to each solver-preconditioner pair, for each system, for each matrix. In order to determine the possible benefits of training the classifier using the additional hardware information, the data must be trained on a single computer system and then the corresponding "good" and "bad" labels that go with it. Similar to the experiments in Chapter 5, a given solver-preconditioner-hardware-timing combination instance is considered "good" if it is within 25% of the fastest solve time for that matrix. The overall success of the predictions is determined by training classifiers on the timing data obtained from one or more computer systems and then testing the resulting classifier's predictions on the runtime data from other computer systems.

Training and testing the classifiers on timing data generated from different computer systems shows how well or poorly the classifier can utilize the hardware information from one or more computer systems and use it to predict the classification for other computer systems. For example, the classifier trains on data obtained from the performance of linear systems being solved using System X, and the resulting classifier can be tested on Systems Y and Z to determine the classifier's ability to generalize. Measuring the prediction performance using multiple testing and training datasets makes it possible to quantify how well a given training dataset extrapolates out to other systems. The experiments are divided into two portions. The first set of experiments examine the performance of predicting the best solver and preconditioner pairs for all of the available computer systems, when the classifier has only been trained on the timing data obtained from one of the systems. The second set of experiments is similar to the first, but is focused on training the classifiers with the timing data obtained from more than one of the computer systems.

6.1.1 Collecting System Information

Numerous metrics exist for computing the performance of a computer system. One open source project, the HPC Challenge [51], contains a suite of benchmarks that determine a variety of performance characteristics. Each of the HPC Challenge benchmarks is described in Section 4.1. The HPC Challenge benchmarks run tests that determine the capabilities of the system, but do not necessarily record the general hardware level description, sizes, or speeds of the actual hardware. Hardware specifications were collected using the **lscpu** Linux system command [83]; this command prints hardware information about the memory and CPU that uniquely identify each computer system. The complete list of system features collected from both the HPCC and lscpu, as well as brief descriptions of what they measure are described in Appendix A.2. Other applications and system commands exist that could possibly provide more insight to the computer's hardware, but these commands and programs typically require root access, which is not feasible on the majority of systems being tested. All the large computer clusters have user guides on their respective websites that describe the cluster's hardware and performance in varying levels of depth.

6.2 Results

Five computer systems were used for the multi-system experiments contained in this section. Four of these five systems are large computational clusters available to researchers, while the fifth is a personal laptop. Each of the systems are listed in Table 6.1, and their most important performance information are detailed in Table 6.2, while all the detailed performance information can be found in Appendix B. Three of the systems: Bridges, Comet, and Stampede, were used as part of the Extreme Science and Engineering Discovery Environment (XSEDE) project [84], Summit is a joint machinr operated by the University of Colorado and Colorado State University, and the Dell XPS13 laptop is my own personal machine.

The performance statistics of each system were obtained using the **lscpu** system command as well as the HPC Challenge suite of benchmarks. The HPC Challenge code was compiled using the

Table 6.1: Brief overview of the five computer systems used in the experiments of this dissertation. Each system was assigned an ID, which is referred to in most of the graphs and results. More detailed hardware information for each of these systems is available in detail in Appendix B.

System	Facility	Cores/Node	System ID
Bridges	Pittsburgh Supercomputing Center	28	1
Comet	San Diego Supercomputing Center	24	2
Summit	Univ. of Colorado / Colorado State Univ.	24	3
Stampede	Texas Advanced Computing Center	16	4
Dell XPS13 Laptop	N/A	4	5

same software stack used for all other experiments in this dissertation, as described in Table 4.1. Running the HPC Challenge program requires one of two inputs. The first input option requires the user to explicitly create the specific dimensions of the matrices and vectors used by the different benchmarks. The second input option takes a single user-based input that is equal to the amount of memory to use per core, thread, or in total; the code then creates the appropriate vectors and matrices to fit the specified size requirement. For the experiments in this dissertation, the HPC Challenge benchmarks were executed on each of the available computer system using all of the available CPU cores on a single node of each individual system. A memory variable of 1GB per core was used for all the benchmarks, thus each benchmark selected the optimal sizes of matrices and vectors for its individual test such that a total of 1GB of memory per core is used to store the data structures. For the computationally intensive benchmarks contained in the HPC Challenge, there are three different execution types, which depend on the number of cores used and the cores' communication with each other,

(1) Single: the benchmark is executed on only one of the available CPU cores

(2) Star: the benchmark is executed in an embarrassingly parallel fashion, with no communication between cores, across all available CPU cores with the average result being returned

(3) MPI: the benchmark benchmark is executed in parallel using all the available CPU cores in conjunction with each other.

It should be noted that the Single and Star methods allow for direct comparisons to be made between the performance of multiple systems, while the MPI method is based on the performance of an entire system node rather than on a single-core average. Table 6.2 shows a subset of the results obtained from the HPC Challenge, as well as additional system information obtained from **lscpu**, for each of the computer systems.

6.2.1 Prediction Performance when Training on a Single Computer System

This section presents the prediction performance of machine learning classifiers trained on timing information obtained from only one of the five available computer systems. The classifiers' performance are based on their ability to predict the best solver and preconditioner for solving a given linear system on only one of the available systems. Based on the results of the binary classifiers in Chapter 5, the Random Forest classifier with a three-fold stratified shuffle split was chosen for the experiments in this chapter.

Similar to the ROC graphs in Chapter 5, the graphs presented in this chapter are labeled using an X_Y format where X represents the system IDs of the timing data used to create the classifiers, while Y represents the system IDs of the timing data used to test the classifiers. Figure 6.1 shows the resulting ROC curves obtained from training on Bridges and testing the classifier's prediction on data on itself and the other four available test systems. Overall, the AUROC for each system is relatively high (≥ 0.8), indicating that there are overlapping successes between systems even with different hardware. The largest AUROC discrepancy when training on Bridges(1) can be found when testing on Comet(2). The most similar AUROC to Bridges predicting on its own data, is from the laptop, indicating that the hardware in those two machines are more similar compared to the others.

Figure 6.2 shows the ROC curves for the classifier trained on data from Bridges(1), and then tested on predicting the results for the other system. Here, the largest difference in AUROC scores, when compared to Summit(3), are found between Bridges(1) with 0.16 and Comet(2) at 0.13. These results, coupled with those found in Figure 6.1 indicate that not only is Bridges(1) different when

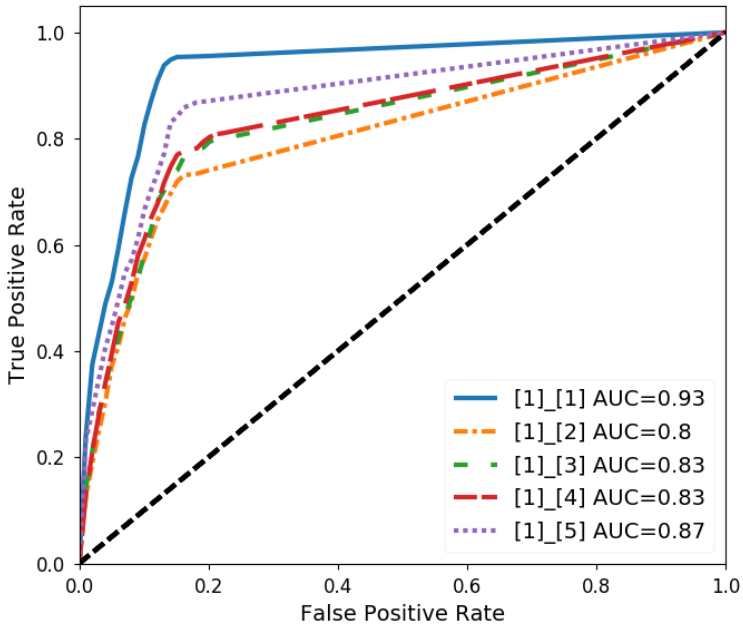


Figure 6.1: ROC Curves for a classifier trained on Bridges(1), and tested on each computer system individually at $np = 4$ for all systems.

compared to the group of Stampede(4), Summit(3), and the laptop(5), but also that it does not have much in common with Comet(2), either.

The ROC curves in Figure 6.3 again depict that there is more similarity among Stampede(4), Summit(3), and the laptop(5), than in Bridges(1) or Comet(2). These results suggest that at fairly low core count of $np = 4$, there are significant similarities between the performance of the laptop(5), Summit(3), and Stampede(4), when solving the tested linear systems.

The complete results for each one-on-one training and testing round are located in Table 6.3. Unsurprisingly, the best prediction performances occur when the same system is used for both the training and testing of the classifier.

6.2.2 Prediction Performance when Training on Multiple Computer System

Because of the similarities in the results for Summit(3), Stampede(4), and the laptop(5), it is possible that data obtained from only one of them is needed in order to correctly predict at a high

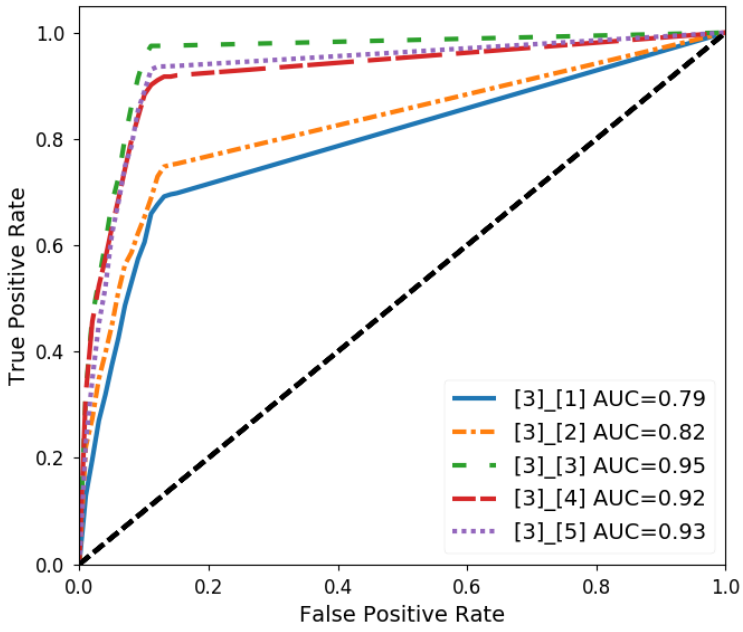


Figure 6.2: ROC Curves for a classifier trained on the Summit computer system, and tested on each computer system individually at $np = 4$ for all systems.

accuracy for the other two systems. Since Bridges(1) and Comet(2) seem to be more unique in their predictions, each of them is tested separately. Figure 6.4 depicts the ROC curves obtained from training on the combination of Bridges(1) and Summit(3), and then testing the resulting classifier on each of the individual systems.

The resulting AUROC for each tested system is very high, ≥ 0.93, except in the case of Comet(2), with an AUROC of 0.84. This discrepancy in scores indicates that there are similarities between Summit(3), Stampede(4), and the laptop(5), and that Bridges(1) and Comet(2) are relatively unique, and are also unique to each other. Similar results are found when the training is on data from Comet(2) and Summit(3), as seen in Figure 6.5. This figure shows a relatively low AUROC of 0.84 when tested on data from Bridges(1), and higher AUROCs for all the other systems ≥ 0.92.

By combining the results from the experiments in Figures 6.4 and 6.5, it is possible to obtain

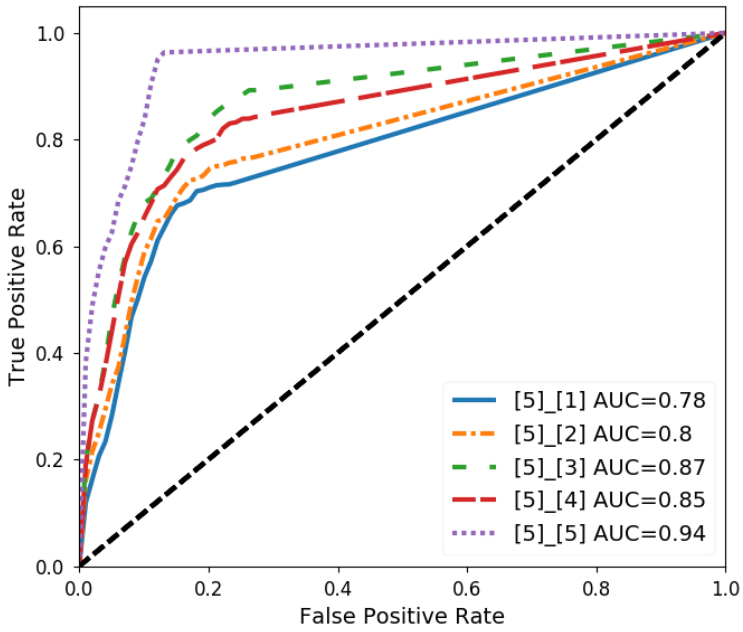


Figure 6.3: ROC Curves for a classifier trained on a Dell laptop(5), and tested on each computer system individually at $np = 4$.

very high prediction performance, all AUROCs ≥ 0.93, across all systems by training on data from only three of the five total systems: Bridges(1), Comet(2), and Summit(3) as depicted in Figure 6.6. The prediction performance using only the data from these three systems is nearly identical to the performance seen when all five systems are used as training data, as evidenced by Figure 6.7. In addition to the three-system combination shown in Figure 6.6, another three-system combination is also capable of achieving almost identical results: Bridges(1), Comet(2), and Stampede(4).

6.2.3 Feature Selection

Table 6.4 contains a list of all the available matrix and system features used in the training of the multi-system classifier. Each of the features is ranked based on its importance using the randomized lasso algorithm. A feature's importance is determined as the percentage of randomized training instances in which the feature was deemed to be important in determining the final classification. Similar to the feature rankings in Section 5.2, the most important features, selected

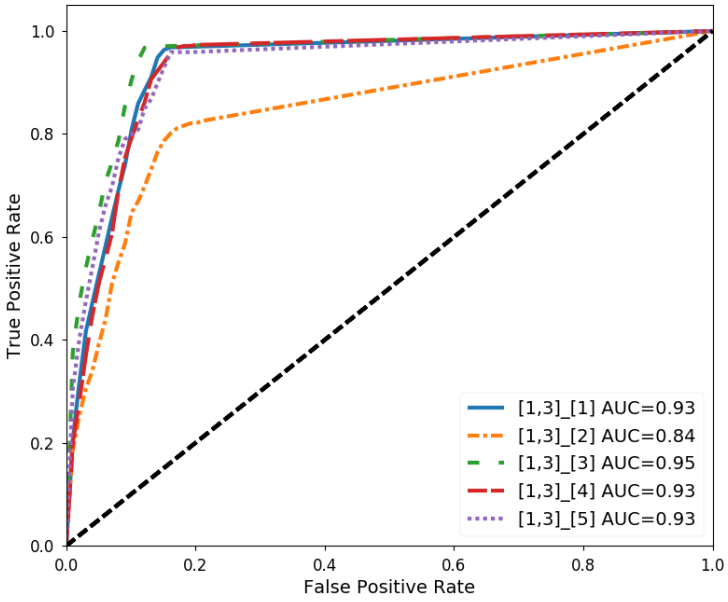


Figure 6.4: ROC Curves for a classifier trained on timing data from Bridges(1) and Summit(2), and then tested on each computer system individually at $np = 4$ for all systems.

$\geq$ 99% of the time, contain the solver, preconditioner, the "type" of dummy rows stored in the matrix, and diagonal dominance. Dummy rows are rows of the matrix A that contain only one nonzero entry. There are three possible "types" for the dummy row entries:

(1) Every dummy row of A contains a 1 along the diagonal of the matrix.

(2) Every dummy row has nonzero entries along the diagonal.

(3) At least one dummy row's entry, regardless of value, is not on the diagonal.

The highest ranked computer system features were selected as important factors in the classification outcome approximately half of the time.e related to cache size and FFT performance, indicating that the solver-preconditioner pairs were more limited by the CPU than memory bandwidth or latency. The majority of the hardware features scor The most impactful system features wered under a 10% selection rate, indicating that they either had a small effect on the algorithms at all, or more likely that they were overshadowed by the impact of one or more other features.

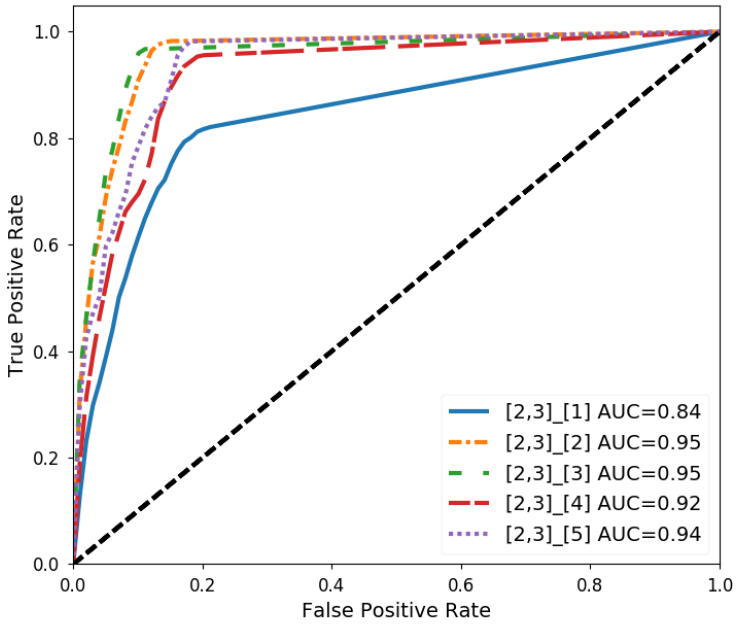


Figure 6.5: ROC Curves for a classifier trained on the Comet(2) and Summit(3) systems, and then tested on each computer system individually at $np = 4$ for all systems.

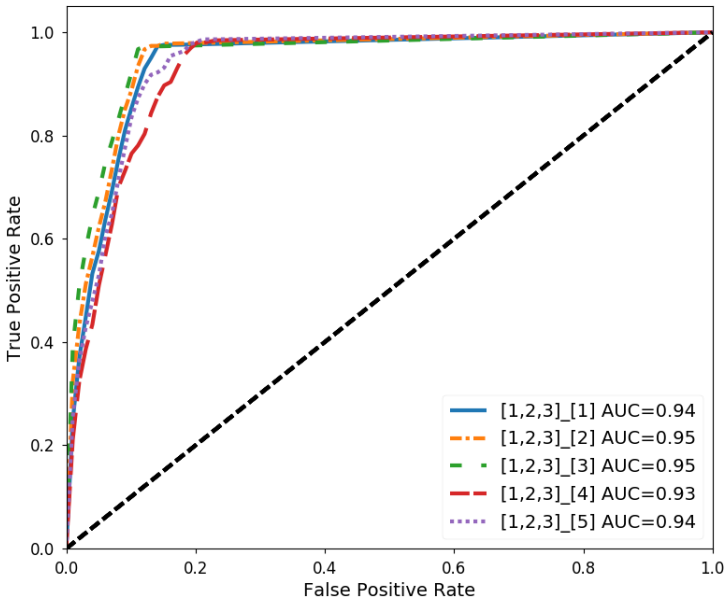


Figure 6.6: ROC Curves for a classifier trained on data from Bridges(1), Comet(2), and Summit(3), and then tested on each computer system individually at $np = 4$.

Table 6.2: The computationally relevant and comparable features and system information of the systems being tested throughout the dissertation. Each computer system's results were determined using the HPC Challenge benchmark and the **lscpu** system command. The complete output of each system's HPC Challenge benchmarks can be found in Appendix B.2

Feature	bridges	comet	summit	stampede	laptop
system_id	1	2	3	4	5
HPL_Tflops	0.22	0.38	0.53	0.29	0.04
StarDGEMM_Gflops	19.81	18.43	28.38	21.52	11.01
SingleDGEMM_Gflops	33.08	21.21	34.26	23.07	15.56
PTRANS_GBs	3.57	4.02	10.89	4.72	0.85
MPIRandomAccess_LCG_GUPs	0.07	0.06	0.10	0.05	0.00
MPIRandomAccess_GUPs	0.07	0.06	0.10	0.06	0.01
StarRandomAccess_LCG_GUPs	0.01	0.03	0.03	0.02	0.03
SingleRandomAccess_LCG_GUPs	0.06	0.06	0.07	0.04	0.06
StarRandomAccess_GUPs	0.01	0.03	0.03	0.02	0.02
SingleRandomAccess_GUPs	0.06	0.06	0.07	0.04	0.06
StarSTREAM_Copy	1.64	4.21	3.69	4.58	5.08
StarSTREAM_Scale	1.38	3.20	2.92	3.31	3.54
StarSTREAM_Add	1.53	3.60	3.34	3.76	3.86
StarSTREAM_Triad	1.59	3.62	3.34	3.73	3.86
SingleSTREAM_Copy	6.96	19.31	5.04	7.58	16.05
SingleSTREAM_Scale	5.02	12.03	5.81	13.16	9.42
SingleSTREAM_Add	5.56	13.13	5.92	14.19	9.93
SingleSTREAM_Triad	5.61	13.15	5.92	14.22	9.92
StarFFT_Gflops	1.05	1.93	1.84	1.77	1.10
SingleFFT_Gflops	1.34	2.74	2.46	2.69	1.95
MPIFFT_Gflops	9.00	17.93	14.20	10.20	2.07
MaxPingPongLatency_usec	0.98	0.58	0.49	0.48	0.70
RandomlyOrderedRingLatency_usec	1.64	0.57	0.70	0.53	0.68
MinPingPongBandwidth_GB	4.96	5.36	11.27	5.95	5.04
NaturallyOrderedRingBandwidth_GB	0.73	0.81	1.44	1.00	1.30
RandomlyOrderedRingBandwidth_GB	0.77	1.23	1.56	1.33	1.39
MinPingPongLatency_usec	0.37	0.39	0.31	0.28	0.48
AvgPingPongLatency_usec	0.57	0.47	0.38	0.38	0.59
MaxPingPongBandwidth_GB	12.09	9.54	15.45	9.83	8.39
AvgPingPongBandwidth_GB	9.51	7.26	14.71	7.75	6.76
NaturallyOrderedRingLatency_usec	1.51	0.62	0.66	0.58	0.68
MemPerProc	1024	512	1024	1024	1024
core_count	28	24	24	16	4
cpu_freq	2300	2500	2500	2700	2200
bogo_mips	4604.72	4988.09	4992.81	5399.28	4389.76
l1_cache	32	32	32	32	32
l2_cache	256	256	256	256	256
l3_cache	35840	30720	30720	20480	3072
memory_size	128	128	128	32	8
memory_freq_MHz	2133	2133	2133	1600	1600
memory_type_ddr	4	4	4	3	3

Table 6.3: The AUROC scores for each single-system classifier when tested on each of the available computer systems. The leftmost column indicates the system that generated the timing data used to create the classifier, while the topmost row specifies the timing data used to test the classifier. The results in bold indicate the best prediction performance for each classifier.

Training \Testing	Bridges(1)	Comet(2)	Summit(3)	Stampede(4)	Laptop(5)
Bridges(1)	**0.93**	0.8	0.83	0.83	0.87
Comet(2)	0.78	**0.95**	0.87	0.85	0.86
Summit(3)	0.79	0.82	**0.95**	0.92	0.93
Stampede(4)	0.8	0.81	0.93	**0.96**	0.91
Laptop(5)	0.78	0.8	0.87	0.85	**0.94**

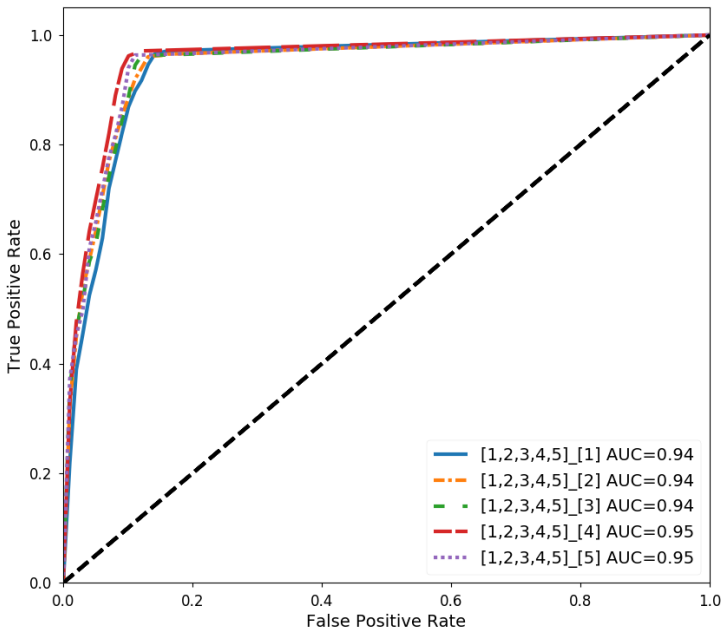


Figure 6.7: ROC Curves for a classifier trained on all possible computer systems and then tested on each computer system individually at $np = 4$.

Table 6.4: The ranking of all system and matrix features based on their importance.

Feature	Importance	Description
col_diag_dom	1	Diagonal entry compared w/ other column entries
dummy_rows_kind	1	Value types of rows w/ one nonzero entry per row
prec_id	1	Choice of preconditioner
solver_id	1	Choice of iterative solver
nnz	0.945	Total number of nonzero entries
lower_bw	0.915	Lower bandwidth
max_nnz_row	0.88	Maximum number of nonzeros in a single row
num_value_symm_2	0.74	Soft numeric value symmetry
row_log_val_spread	0.73	Max ratio between a row's min and max entries
dummy_rows	0.685	Number of rows w/ only one nonzero entry
num_value_symm_1	0.635	Hermitian property of matrix
bogo_mips	0.48	Busy-loop performance
l3_cache	0.48	Size of L3 cache
MPIFFT_Gflops	0.475	FFT performance in parallel
inf_norm	0.465	Infinity norm
antisymm_inf_norm	0.39	The infinity norm of $A - A^T/2$
min_nnz_row	0.39	Minimum number of nonzeros in a single row
trace	0.385	Sum of all diagonal entries
abs_trace	0.37	Sum of the absolute values of the diagonal entries
MinPingPongLatency_usec	0.365	Minimum pingpong latency
col_log_val_spread	0.34	Max ratio between a column's min and max entries
diag_sign	0.33	The types of values stored on the diagonal
nnz_pattern_symm_2	0.325	% of nonzero entries a_{ij} with nonzero entries at a_{ij}
cpu_freq	0.29	Frequency of the CPU
antisymm_frob_norm	0.285	Frobenius norm of $A - A^T/2$
StarFFT_Gflops	0.28	Average FFT performance per core
MPIRandomAccess_GUPs	0.275	Random access updates in parallel
symm_inf_norm	0.265	Infinity norm of $A + A^T/2$
HPL_Tflops	0.21	Dense matrix solve performance
nnz_pattern_symm_1	0.185	If all nonzero entries a_{ij} have nonzeros at a_{ji}
row_diag_dom	0.185	Diagonal entry compared w/ other row entries
symm	0.18	If $A = A^T$
one_norm	0.175	One norm
NaturallyOrderedRingBandwidth_GB	0.135	Ring bandwidth in sequential communication order
rows	0.13	Number of rows
upper_bw	0.13	Upper bandwidth
MPIRandomAccess_LCG_GUPs	0.1	Random access updates in parallel
cols	0.095	Number of columns
AvgPingPongLatency_usec	0.075	Average pingpong latency per core
MemProc	0.075	Amount of system memory per core
row_var	0.075	Largest of each row's variance
diag_avg	0.065	Average of the diagonal entries
SingleSTREAM_Triad	0.065	Triad stream performed on a single core
StarDGEMM_Gflops	0.06	DGEMM performance average per core
SingleRandomAccess_LCG_GUPs	0.055	Random access updates on a single core

SingleSTREAM_Add	0.045	Add stream performed on single core
memory_size	0.03	Total amount of memory
diag_nnz	0.025	Number of nonzero diagonal entries
PTRANS_GBs	0.02	Parallel matrix transpose performance
SingleFFT_Gflops	0.02	FFT performance on a single core
SingleRandomAccess_GUPs	0.02	Random access updates on a single core
avg_nnz_row	0.015	Average number of nonzeros in a single row
col_var	0.015	Largest of each column's variance
RandomlyOrderedRingBandwidth_GB	0.015	Ring bandwidth in random communication order
StarRandomAccess_GUPs	0.015	Average number of random updates per second
memory_freq	0.01	Memory frequency
SingleSTREAM_Scale	0.01	Scale stream performed on a single core
memory_type	0.005	DDR3 or DDR4
SingleDGEMM_Gflops	0.005	DGEMM single core performance
SingleSTREAM_Copy	0.005	Copy stream on single CPU core
StarRandomAccess_LCG_GUPs	0.005	Average number of random updates per second
StarSTREAM_Copy	0.005	Copy stream average per core
StarSTREAM_Triad	0.005	Triad stream average per core
AvgPingPongBandwidth_GB	0	Average pingpong bandwidth per core
diag_var	0	Variance of the diagonal entries
frob_norm	0	Frobenius norm
l1_cache	0	Size of L1 cache
l2_cache	0	Size of L2 cache
MaxPingPongBandwidth_GB	0	Maximum bandwidth using pingpong
MaxPingPongLatency_usec	0	Maximum latency between cores
MinPingPongBandwidth_GB	0	Minimum pingpong bandwidth
NaturallyOrderedRingLatency_usec	0	Ring latency in sequential communication order
RandomlyOrderedRingLatency_usec	0	Ring latency in random communication order
StarSTREAM_Add	0	Add stream average per core
StarSTREAM_Scale	0	Scaling stream average per core
symm_frob_norm	0	Frobenius norm of $A + A^T/2$

Together, all the results show that there are small, but measurable, differences that occur when training and testing classifiers on timing data obtained from solving linear systems using different hardware. This shows that that current methods can result in high prediction performance when recommending the best solver-preconditioner pairs when the user and classifier are both based on solving systems on the same hardware. However, if the user plans to use different hardware to solve the system than was used to generate the classifier's training data, then prediction performance degrades somewhat. Multi-system prediction is only one aspect of incorporating hardware information as part of the solver-preconditioner pair recommendation process. The previous chapter tackled the problem of multiple numbers of CPU cores, while Chapter 7 combines these two ideas into a single problem. Finally, Chapter 8 tests the resulting classifiers' performances when tasked with predicting the best solver-preconditioner pairs for an unseen real-world problem.

Chapter 7

Combining Multicore and Multi-system Prediction

The goal of this dissertation is to examine the prediction performance of incorporating system hardware information into recommending the best performing iterative solvers and preconditioners for solving a given sparse linear system. This chapter takes the two separate ideas from chapters 5 and 6 and combines them into a single problem. To create a truly hardware-aware recommendation system, the recommender must be aware of both the type of hardware being used to solve the linear system, as well as the scale of the hardware being used. This chapter's contribution to the overall goal is examining the overall prediction performance obtained by predicting how a given linear system would be best solved when supplied with multiple sets of timing data obtained from a variety of computer systems hardware and scales. Section 7.1 of this chapter describes the methodology of designing and performing the experiments associated with combining multicore and multi-system prediction methods discussed in Chapters 5 and 6. Section 7.2 presents the results obtained from these multicore and multi-system experiments.

7.1 Problem Description

The overall structure of the problem as well as the experiments discussed in this chapter are similar to those found in the previous two chapters. This chapter combines the two previous chapters' experiments and ideas regarding the separate multi-system and multicore problems and implements them to work in conjunction with each other. Combining these two different hardware aspects into a single classifier expands upon selecting the optimal solver and preconditioner for a

given linear system to include both the details of the available hardware and the number of available cores.

7.1.1 Data

The experiments in this chapter expand upon those in previous chapters by combining multiple sources of information to aid in the machine learning-based prediction of optimal solvers and preconditioners for solving sparse linear systems. Prediction, in this case, is based on training data obtained from four sources:

(1) Numerical and structural features of each testing/coefficient matrix

(2) Wallclock times taken to solve linear systems, using the aforementioned matrices, on different systems and number of processors

(3) Hardware specifications and information for each computer system

(4) Binary classification labels, used for training, indicating how well solvers and preconditioners perform on different numbers of processors, computer systems, and matrices.

The experiments described in this chapter use the same set of matrices obtained from the University of Florida's Sparse Matrix Collection [65] as described in Chapters 5 and 6. Features from each matrix were obtained using Anamod and contain multiple numerical and structural properties unique to each matrix. A full list of the matrix features, as well as their descriptions, can be found in Appendix A.1.

As described in Chapter 6, due to computational constraints, not every matrix was selected to be used in a linear solve on the laptop computer. The number of matrices tested on the laptop consisted of approximately half, 700, of the total number of matrices tested on the other systems. The matrices selected for the laptop-based solves were the 700 matrices requiring the least amount of memory for storing the matrix.

The computer systems used for the experiments are identical to those found in Chapter 6 and consist of PSC's Bridges(1), SDSC's Comet(2), TACC's Stampede(3), CU's Summit(4), and my

own personal Dell laptop(5). Each system has its own unique hardware information; the detailed specifications of each machine are fully described in Appendix B. The computer system specifications and information were collected in the same manner as described in Chapter 6, using the HPC Challenge collection of benchmarking tools, discussed in detail in section 4.1, and the **lscpu** system command. Each system's information was collected using the HPC Challenge's memory-based input option with 1GB of memory being dedicated for each available core in a single node. Rather than collecting timing data at each possible core count, each linear system was solved serially, as well as in parallel at every multiple of four cores thereafter. The laptop was the only exception to this rule, with solves also being performed at $np = 2$. The exact breakdown of each system's cores used for testing can be found in Table 7.1.

Table 7.1: The number of cores for each system used to solve the linear systems.

	Cores Tested
Bridges(1)	[1, 4, 8, 12, 16, 20, 24, 28]
Comet(2)	[1, 4, 8, 12, 16, 20, 24]
Summit(3)	[1, 4, 8, 12, 16, 20, 24]
Stampede(4)	[1, 4, 8, 12, 16]
Laptop(5)	[1, 2, 4]

7.1.2 Machine Learning

In the previous two chapters, each linear system solve was given only a single binary classification. Each solve instance was given a classification as either "good" or "bad." However, this system is not sufficient for combining the two sub-problems into a single problem, due to the new problem's two-dimensional nature. Each of the sub-problems inChapters 5 and 6 are inherently one-dimensional due to having only one independent variable: the number of CPU cores and the different system hardware, respectively. Therefore, how do we best classify solver-preconditioner pairs when multiple systems and core counts are available options? Solving this requires expanding the classification methods from binary to **multi-label**. In multi-label classification, each input vector can be classified as one or more of the available labels. Multi-label classification can be

represented as a series of binary classifications with each output label being true or false, depending on whether or not the current input vector belongs to that label.

For the experiments of this chapter, each matrix, solver-preconditioner, system, and core count permutation was assigned one or more of 18 possible classification labels. For these experiments, the 18 total possible binary labels were based on encountering a solving error, convergence, and relative timing performance compared to other input vector combinations. Solving error, the first binary label, is simply based on whether or not the given solver-preconditioner pair encountered a runtime error while executing the linear solve, with "true" indicating that it encountered an error. The second binary label given to each input, is based on the convergence status of the linear solve, with "true" indicating that is did converge. The remaining 16 binary labels are based on the performance of the specific solver and preconditioner pair on a given linear system with the given hardware and number of processors. These 16 labels are further broken down into four distinct groups of labels based on the wallclock time taken to solve the given linear system and to what other input vectors it should be compared against. These subgroups are based on comparing the input vector to

(1) Other linear solves having the same matrix, computer system, and number of processors (np_and_system)

(2) Only other linear solves having the same matrix and computer system, regardless of the number of processors (sys)

(3) Only other linear solves having the same matrix and number of processors, regardless of the computer system (np)

(4) All other linear solves having the same matrix, regardless of the computer system or number of processors used (overall).

These four groups of classification labels are then subdivided even further into four subgroups each. Each of these smaller groups is created based on the amount of thresholding to incorporate into the binary classification of each input. All the thresholding variables are percentage-based

and have values of 0, 25, 50, and 100%, which correspond to the amount of acceptable tolerance in the wallclock time. This tolerance scheme expands upon the simpler fixed 25% tolerance-based "good" and "bad" output of the experiments described in Chapters 5 and 6. For instance, in the 0% tolerance case of the group that is agnostic to both the system and number of processors, only the absolute fastest solver-preconditioner pair for each matrix is considered to be good and the binary classification for that specific label is labeled true. All the other numbers incorporate a tolerance such that the fastest solver-preconditioner pair as well as the system and number of processors allow for multiple possible "true" cases. For example, the 25% tolerance allows for any of the solver-preconditioner pairs combined with the appropriate grouping information to be accepted as true as long as the overall time is within 25% of the fastest successful solve at that matrix and within the given additional specifications. This allows for 16 additional groupings based on four percentage groups given for each of the four hardware groups. Coupling these 16 binary labels with the existing error and convergence binary labels results in 18 total possible labels for each input vector.

7.1.3 Additional Experiment Measurements

There are numerous methods for determining the success and failures of a given classifier. In Chapters 5 and 6, the receiver operating characteristic (ROC) curves, and their integrals, are the primary metric for determining a classifier's prediction performance. This section is dedicated to briefly describing other possible performance metrics.

7.1.3.1 Confusion Matrix Derivations

Every binary classifier's results can be depicted as a $2x2$ matrix with the predicted results along one axis and the actual results on the other axis. Each input instance is then categorized into one of the four resulting cells depending on its actual label and the label assigned to it by the classifier. "Positive" and "negative" describe the label assigned to the input vector by the classifier, and "true" and "false" describe the correctness of the predicted label. For instance, a true negative

instance describes a correctly assigned negative label from the classifier on an input vector. An example confusion matrix is depicted in Figure 7.1.

		Predicted Classification		
		Negative	Positive	Total
True Classification	Negative	True Negative (TN)	False Positive (FP)	$TN + FP$
	Positive	False Negative (FN)	True Positive (TP)	$FN + TP$
	Total	$TN + FN$	$FP + TP$	N

Figure 7.1: An example of a confusion matrix created from a binary classifier. The results are divided into the classifier's predicted results and the true results. The four entries in the table can be combined in many ways to create performance metrics.

There are many different possible combinations of the four confusion matrix entries that are useful for determining a classifier's success. Some of the most common and useful combinations are listed below along with their formulas.

(1) Accuracy: Proportion of all labels that are correct. Answers: How many of the predicted labels match their actual labels?

$$\frac{TN + FP}{N}$$

(2) True Positive Rate (TPR) / Sensitivity / Recall: Proportion of labels assigned as positive that are actually positive. Answers: Given that the actual input is positive, what is the probability of predicting a positive label?

$$\frac{TP}{TP + FN}$$

(3) True Negative Rate (TNR) / Specificity: Proportion of labels assigned as negative that are actually negative. Answers: Given that the actual input is negative, what is the probability of predicting a negative label?

$$\frac{TN}{TN + FP}$$

(4) Precision / Positive Predictive Value (PPV): Proportion of true positive labels out of all positive results. Answers: Given that the predicted label is positive, what is the probability

of the actual input being positive?

$$\frac{TP}{TP + FP}$$

(5) Negative Predictive Value (NPV): Proportion of true negative labels out of all negative results. Answers: Given that the predicted label is negative, what is the probability of the actual input being negative?

$$\frac{TN}{TN + FN}$$

(6) False Positive Rate (FPR): Proportion of labels assigned as positive that are actually negative. Answers: Given that the actual input is negative, what is the probability of predicting a positive label?

$$\frac{FP}{FP + TN}$$

(7) False Negative Rate (FNR): Proportion of labels assigned as negative that are actually positive. Answers: Given that the actual input is positive, what is the probability of predicting a negative label?

$$\frac{FN}{FN + TP}$$

(8) False Discovery Rate (FDR): Proportion of labels assigned incorrectly as positive compared to all outputs labeled as positive. Answers: What is the classifier's rate of false positives (type I errors)?

$$\frac{FP}{FP + TP}$$

(9) False Omission Rate (FOR): Proportion of labels assigned incorrectly as negative compared to all outputs labeled as negative. Answers: What is the classifier's rate of false negatives (type II errors)?

$$\frac{FN}{FN + TN}$$

(10) F1 Score: The harmonic mean of the precision (PPV) and recall (TPR). Answers: What is the weighted average of the classifier's precision and recall?

$$\frac{2TP}{2TP + FP + FN}$$

(11) Matthews Correlation Coefficient (MCC): The correlation coefficient between the actual input labels and the classifier's predicted labels. A balanced measure of a classifier since it uses both true and false positives and negatives. The only value not scaled from 0.0 to 1.0, but rather -1.0 to 1.0. A value of 1.0 indicates perfect classification, 0.0 indicates random classification, and -1.0 indicates incorrect classification for all input. Answers: How much correlation is there between the classifier's predicted labels and the actual labels?

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

7.2 Results

For the experiments in this chapter, a total of 18 binary classifiers are available for assignment to each input vector. All the classifier labels are based on each solver-preconditioner pair's performance when solving linear systems on multiple computer systems and numbers of cores. Compared to the single binary classification methods used in Chapters 5 and 6, the experiments in this chapter use multi-label classification. The collection of binary classifiers represents the 18 possible labels that can be assigned to each input vector. These labels were previously described in Section 7.1. Multiple labels can be assigned to each input vector, and the individual labels are represented as binary classifications with "true" indicating that the label is assigned. The rest of this section is divided into presenting the results obtained for each of the 18 classifiers using a variety of statistical metrics. Based on the results of the binary classifiers in Chapter 5, the Random Forest classifier with stratified shuffle split was chosen for the experiments in this chapter. However, due to the size of the dataset, the number of splits was reduced from 10 to 3 to reduce the total processing time.

Because the data associated with solvers and preconditioners are heavily imbalanced in favor of the null case, in which a given solver-preconditioner pair is not ideal for solving a given sparse linear system, some metrics are more interesting and important than others. For instance, the accuracy metric, while important, is not ideal in imbalanced scenarios due to its reliance on only true positive and true negative classifications. Because accuracy does not take into account false negatives or false

positives, it is possible to have an incredibly accurate classifier that always assigns the "false" label, regardless of the actual input. Since the recommendation system in this dissertation is ultimately concerned with correctly identifying the best solver-preconditioner pair for a given linear system, the true positive rate (TPR) is an important metric. The TPR of a label is the probability of correctly labelling an actual positive input as being positive. TPR for all 18 classification labels are depicted in Figure 7.2. The true positive rates of determining if a solver-preconditioner pair causes an error or converges to a solution is relatively high compared to the performance of the other classifier groups, at 85% and 77% respectively. Such a high TPR for these two classifiers indicates that predicting when an error will occur during a linear solve is the most easily identifiable label of a given input combination. Predicting if the combination will converge to a solution is the second most identifiable label. The rest of the TPRs for the four different label groups are

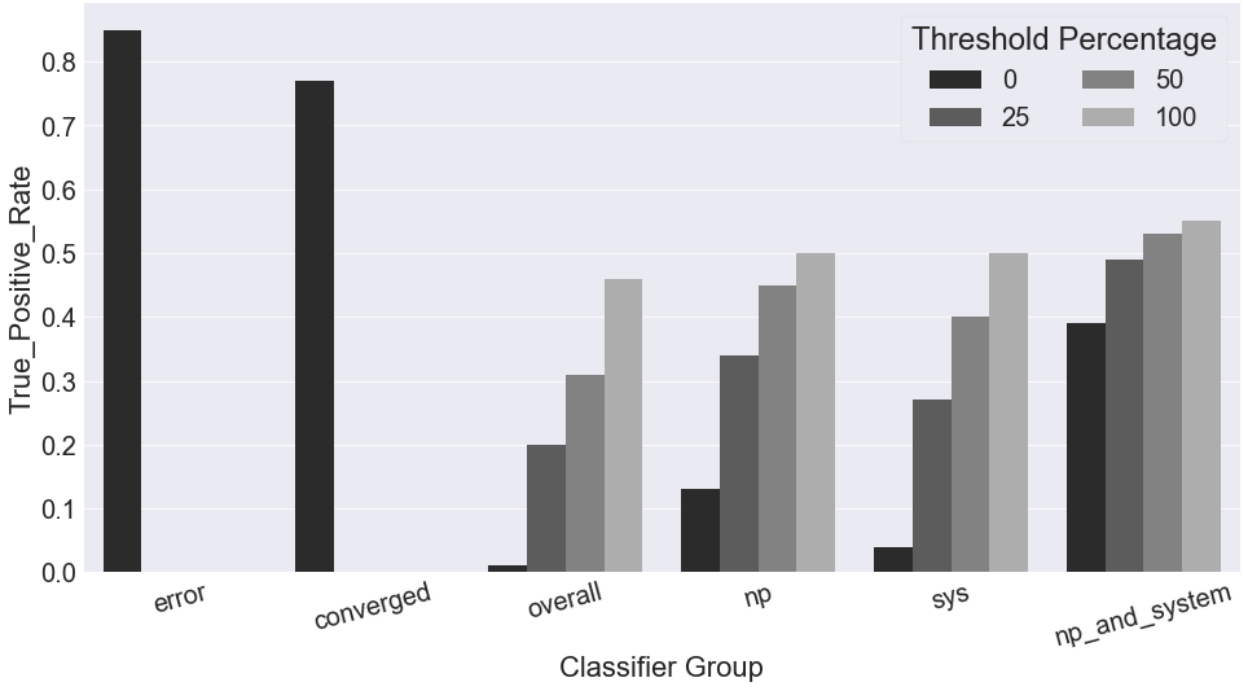


Figure 7.2: Comparison of true positive rates for each of the six groups and four threshold percentages.

significantly lower than the TPRs of the error and convergence labels. Such low percentages indicate that it is less difficult, in general, to correctly predict convergence and solver errors than the best

performing solver-preconditioner combination. The TPRs of the four groups at the 0% threshold are incredibly low, indicating that only using the fastest wallclock times in each group causes the prediction of true-positive results to be significantly lower than the 25, 50, and 100% thresholds. TPRs increase most noticeably for all groups when the threshold is increased from 0% from 25%. Increasing the threshold beyond 25% does improve the TPR, but with diminishing returns as the threshold percentage increases.

Another performance metric, covered in Section 5.1, is the receiver operating characteristic (ROC) curve. The ROC curve plots the true positive rate of the classifier against the false positive rate. A popular metric, which can be obtained from a given ROC curve, is the area under the receiver operating characteristic (AUROC), which is simply the integral of the ROC curve. The AUROC represents how well the classifier can accurately classify two input instances, each from the two binary instances of the individual output labels, into their respective outputs. Therefore an AUROC of 1.0 is equivalent to a perfect binary classification method that always selects the correct label for an input instance. Conversely, an AUROC of 0.5 is equivalent to randomly guessing when classifying any given input.

The AUROC results from each of the 18 classifiers are shown in Figure 7.3. Both the AUROC

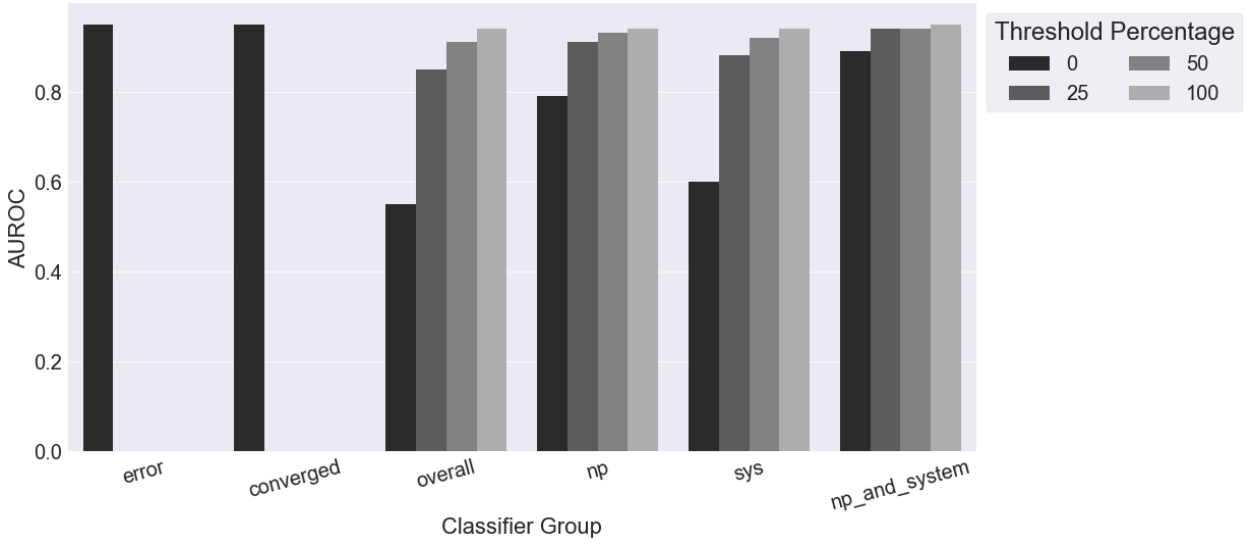


Figure 7.3: Comparison of the area under the receiver operating characteristic curve for each of the six groups and four threshold percentages.

of the error and convergence labels are very high at 0.95 each, indicating that the classifier is capable of correctly predicting the two labels when presented with two inputs: one each of the true and false cases per label. Again, the 0% threshold classifiers perform the worst for each of the timing-based classifiers, implying that the amount of information is too limited to correctly correlate features to the fastest times. For the "np_and_system" group of labels, the 0% threshold AUROC is only marginally lower ($\approx$ -0.05) than the rest of the group's classifiers at higher thresholds. This small difference between the group's AUROCs indicates that there is already sufficient training data from only looking at the fastest solver-preconditioner pairs for each input matrix-system-np combination. By its nature, the specific combination of the "np_and_system" group results in a higher total number of actual "true" values. The higher number of "true" instances occurs because the total number of fastest solver-preconditioner pairs per matrix is (num_np_values $\times$ num_systems) for the 0% threshold. The other classifier groups: "overall", "np", and "sys", contain a maximum of (1), (num_np_values), and (num_systems) "true" instances, respectively, for each of the training matrices at the 0% threshold. These other groups see improvements with larger thresholds due to the threshold allowing for more applicable training data to be included, which performs similarly to the fastest instance rather than only the absolute fastest instance within that group.

Similar in nature to the receiver operating characteristic (ROC) curves discussed in Section 5.1, a precision-recall curve is another common performance metric that plots the precision against the recall of the classifier at various thresholds. Ideally, a classifier has both high recall and high precision, which indicate that the classifier returns both accurate results (precision) and a high percentage of positive results (recall). Precision represents the measure of "result relevancy," a measure of how many truly relevant results are classified. Recall represents the measure of how many "relevant results" are classified. A classifier with high recall and low precision returns many results, but they are likely to be incorrectly identified. Classifiers with a low recall and high precision return very few results, but they are identified correctly. The precision-recall curve of a classifier with random performance is depicted by a horizontal line on the graph located at the precision value determined by the fraction of positive instances of the classifier. Classifiers with both perfect

precision and recall result in a graph represented as a horizontal line at Precision= 1 and a vertical line at Recall= 1, creating an area under the curve of 1.0. Similar to the ROC curves, it's possible to distill each precision-recall curve to a scalar value by measuring the area under the curve (AUPR).

The area under the precision-recall (AUPR) results for each of the created classifiers is shown in Figure 7.4. Again, these results show relatively low prediction performance at each of the 0% cases in the groups of four entries (0.02 - 0.51), while the "error" and "converged" labels have high AUPRs of 0.93 and 0.87 respectively. Drastic differences between the 0% threshold AUPRs indicate that it is more difficult to correctly predict the overall fastest solver-preconditioner pair for a linear system than for predicting at a given core count, system, or the combination of the two.

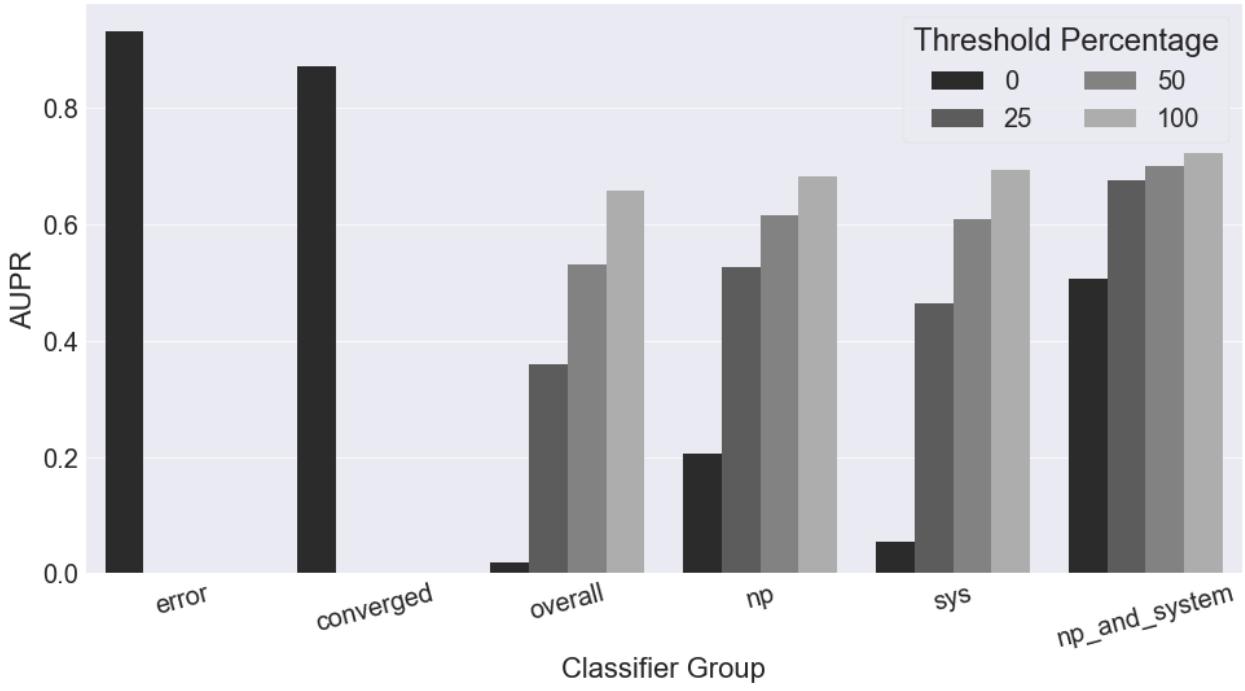


Figure 7.4: Comparison of the area under the precision-recall curve for each of the six groups and four threshold percentages.

The Matthews Correlation Coefficient (MCC) is considered to be one of the more balanced single-value metrics for determining the success of a binary classifier [85]. Its balance comes from combining the predicted and actual values of a classifier's confusion matrix: true/false and positive/negative values, into a single correlation value on the range of $[-1, +1]$. A score of $+1$

indicates a perfect positive correlation between predicted and actual labels, -1 indicates perfect negative correlation, and 0 indicates no correlation. The MCCs of the available labels are depicted in Figure 7.5. Again, the MCC shows similar results as the previous metrics, with the error and

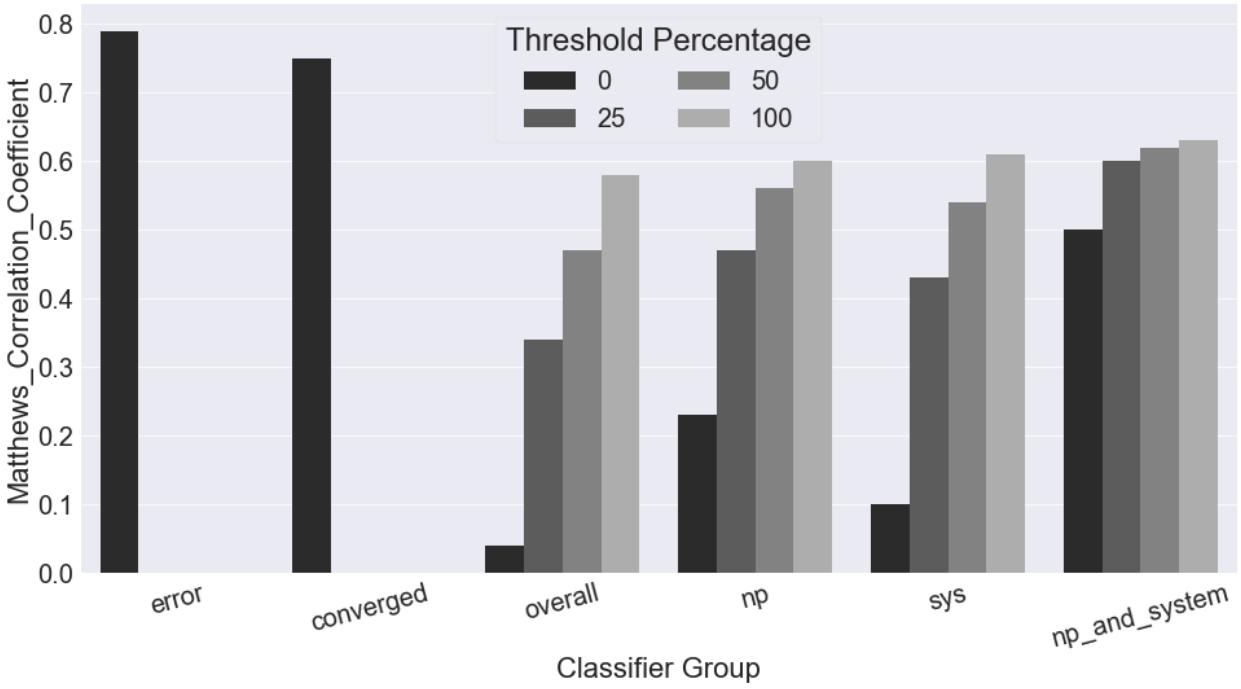


Figure 7.5: Comparison of the Matthews Correlation Coefficient (MCC) for each of the six groups and four threshold percentages.

convergence labels scoring significantly higher than the other 16 labels belonging to the individual classifier groups. Also similar to previous results, the improvement in the MCCs between the 0% threshold and 25% threshold are significantly larger than the gain obtained from the other increases to the threshold percentage.

Overall the results in all tests indicate that there is a significant advantage to including wallclock times that are close to the optimal wallclock time within a given group. All tests show a dramatic improvement when increasing from a threshold percentage of 0% to 25%, but only a minimal gain or even loss in performance when increasing the threshold percentage greater than 25%. This stagnation that occurs when increasing the threshold percentage indicates that the most important information and correlations within the data can be obtained from the additional timings

close to the best performing time, while the inclusion of times beyond that does not contribute much, if anything, to the classifier. The full results of each label's classification results can be viewed in their confusion matrix form in table C.1. All the label metrics derived from the confusion matrix results are shown in table C.2.

7.2.1 Feature Importance

The rankings of the system and matrix features, as well as the number of CPU cores used to solve the linear system, are depicted in Table 7.2. Overall, the ranking, especially those ranked towards the top of the table, are very similar to the results found in Table 6.4. This similarity between the two tables indicates that there is significant overlap of the most important features across individually different classification labels. It should be noted that the number of cores used for the computation, np, is selected as an important feature in about 83% of the random lasso-based tests, indicating the number of processors used was only important in approximately half of all predictions.

Table 7.2: The ordering of the importance of the number of processors based on the mean importance over the 18 individual classification labels. The full importance information for each individual label is available in Appendix C.

Feature	Mean	Std Dev	Description
solver_id	1.000	0.250	Choice of iterative solver
dummy_rows_kind	0.854	0.318	Value types of rows w/ one nonzero entry per row
np	0.834	0.337	Number of CPU cores
col_diag_dom	0.799	0.354	Diagonal entry compared w/ other column entries
prec_id	0.773	0.355	Choice of preconditioner
min_nnz_row	0.753	0.331	Minimum number of nonzeros in a single row
col_log_val_spread	0.583	0.303	Max ratio between a column's min and max entries
diag_nnz	0.556	0.323	Number of nonzero diagonal entries
num_value_symm_2	0.525	0.327	Soft numeric value symmetry
lower_bw	0.524	0.376	Lower bandwidth
diag_sign	0.475	0.303	The types of values stored on the diagonal
NaturallyOrderedRingBW	0.464	0.308	Ring bandwidth in sequential communication order
dummy_rows	0.438	0.329	Number of rows w/ only one nonzero entry
num_value_symm_1	0.427	0.324	Hermitian property of matrix
max_nnz_row	0.384	0.381	Maximum number of nonzeros in a single row
row_diag_dom	0.368	0.366	Diagonal entry compared w/ other row entries

antisymm_inf_norm	0.364	0.377	The infinity norm of $A - A^T/2$
upper_bw	0.329	0.302	Upper bandwidth
row_log_val_spread	0.320	0.179	Max ratio between a row's min and max entries
l3_cache	0.302	0.298	Size of L3 cache
avg_nnz_row	0.296	0.290	Average number of nonzeros in a single row
RandomlyOrderedRingBW	0.263	0.154	Ring bandwidth in random communication order
nnz_pattern_symm_2	0.258	0.171	% of nonzero entries a_{ij} with nonzero entries at a_{ij}
symm_inf_norm	0.240	0.166	Infinity norm of $A + A^T/2$
MemProc	0.231	0.241	Amount of system memory per core
nnz	0.225	0.251	Total number of nonzero entries
inf_norm	0.201	0.206	Infinity norm
rows	0.199	0.184	Number of rows
nnz_pattern_symm_1	0.190	0.126	If all nonzero entries a_{ij} have nonzeros at a_{ji}
MinPingPongLatency	0.175	0.146	Minimum pingpong latency
memory_freq	0.175	0.135	Memory frequency
memory_size	0.166	0.106	Total amount of memory
MinPingPongBW	0.156	0.103	Minimum pingpong bandwidth
memory_type	0.150	0.136	DDR3 or DDR4
AvgPingPongLatency	0.147	0.152	Average pingpong latency per core
SingleSTREAM_Copy	0.124	0.195	Copy stream on single CPU core
diag_var	0.121	0.171	Variance of the diagonal entries
one_norm	0.120	0.123	One norm
bogo_mips	0.117	0.143	Busy-loop performance
MPIFFT_Gflops	0.114	0.098	FFT performance in parallel
symm	0.098	0.086	If $A = A^T$
cols	0.093	0.095	Number of columns
diag_avg	0.093	0.144	Average of the diagonal entries
antisymm_frob_norm	0.092	0.144	Frobenius norm of $A - A^T/2$
MaxPingPongLatency	0.089	0.073	Maximum latency between cores
StarSTREAM_Copy	0.066	0.076	Copy stream average per core
col_var	0.064	0.086	Largest of each column's variance
cpu_freq	0.060	0.084	Frequency of the CPU
StarSTREAM_Scale	0.056	0.066	Scaling stream average per core
AvgPingPongBW	0.055	0.077	Average pingpong bandwidth per core
SingleRandomAccess_LCG_GUPs	0.053	0.108	Random access updates on a single core
StarRandomAccess_LCG_GUPs	0.051	0.086	Average number of random updates per second
PTRANS_GBs	0.047	0.037	Parallel matrix transpose performance
row_var	0.047	0.054	Largest of each row's variance
symm_frob_norm	0.045	0.073	Frobenius norm of $A + A^T/2$
SingleSTREAM_Triad	0.043	0.062	Triad stream performed on a single core
StarSTREAM_Add	0.039	0.074	Add stream average per core
StarRandomAccess_GUPs	0.038	0.075	Average number of random updates per second
SingleRandomAccess_GUPs	0.035	0.059	Random access updates on a single core
trace	0.032	0.062	Sum of all diagonal entries
frob_norm	0.031	0.050	Frobenius norm
StarSTREAM_Triad	0.031	0.053	Triad stream average per core
abs_trace	0.030	0.043	Sum of the absolute values of the diagonal entries
NaturallyOrderedRingLatency	0.030	0.042	Ring latency in sequential communication order
MPIRandomAccess_GUPs	0.026	0.058	Random access updates in parallel

SingleFFT_Gflops	0.024	0.062	FFT performance on a single core
SingleDGEMM_Gflops	0.023	0.051	DGEMM single core performance
SingleSTREAM_Add	0.022	0.031	Add stream performed on single core
StarDGEMM_Gflops	0.022	0.020	DGEMM performance average per core
MaxPingPongBW	0.018	0.033	Maximum bandwidth using pingpong
RandomlyOrderedRingLatency	0.018	0.042	Ring latency in random communication order
StarFFT_Gflops	0.017	0.034	Average FFT performance per core
SingleSTREAM_Scale	0.013	0.023	Scale stream performed on a single core
MPIRandomAccess_LCG_GUPs	0.011	0.017	Random access updates in parallel
HPL_Tflops	0.006	0.009	Dense matrix solve performance
l1_cache	0.000	0.000	Size of L1 cache
l2_cache	0.000	0.000	Size of L2 cache

Chapter 8

Prediction Applied to a Real-World Flow Problem

The goal of this dissertation is to examine the prediction performance of incorporating system hardware information into recommending the best performing iterative solvers and preconditioners for solving a given sparse linear system. This chapter's contribution to the overall goal is testing the prediction performance of the combined multicore and multi-system classifier, created in Chapter 7, on a real-world problem. The combined classifiers are tested on a small collection of lid-driven cavity flow problems generated from example code from the PETSc library. Section 8.1 describes the cavity flow problem as well as the experiments used to examine the prediction performance of the previously created classifier from Chapter 7. Section 8.2 shows the prediction results and subsequent analysis of applying the combined classifier to the sparse matrices created from the lid-driven cavity flow problem.

8.1 Problem Description

Thus far, the machine learning classifiers created throughout this dissertation have been trained and tested on matrices collected from the University of Florida's Sparse Matrix Collection [65]. The applications that generate the UF collection matrices come from a variety of fields and use cases. Because of the time and computational resources required to solve the various linear systems for each matrix, the UF matrices used throughout these experiments are limited in their size, as briefly described in Section 4.5. Therefore, this chapter is devoted to testing the classifiers' performance on a real-world problem being executed in parallel.

The lid-driven cavity flow problem is a common testing example in the field of computational fluid dynamics [86, 87]. The problem can simply be thought of as a square container filled with liquid. Three sides of the square remain fixed in place, while the lid of the container is capable of moving laterally in the X-direction. An illustrated example of the problem can be seen in Figure 8.1.

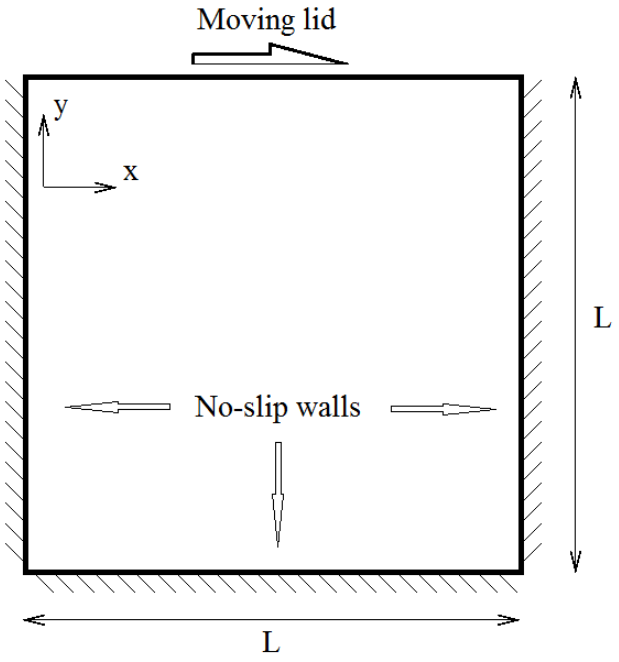


Figure 8.1: Depiction of the general lid-driven cavity flow problem [88].

Like most 2D fluid problems, the lid-driven cavity flow problem is divided into a two-dimensional Cartesian mesh, which then undergoes some level of refinement. Refinement occurs when the existing mesh is subdivided into more cells than were in the original mesh. A more refined mesh contains more data points and therefore more details about the system, but subsequently requires more computational power to solve. The PETSc example problem uses the finite difference method with a five-point stencil on a uniformly refined two-dimensional Cartesian mesh. Each point of the mesh contains four unknowns that are determined at each iteration:

- Velocity in the X-direction

- Velocity in the Y-direction
- Temperature
- Vorticity.

The example code contains multiple input parameters that vary across the runs and impact the size of the resulting sparse linear systems and their values,

- Mesh Size – the total number of cells that exist after refinement on the 2D mesh
- Grashof number (GR) – a dimensionless variable that is equal to the ratio of buoyancy to the viscous force of a fluid
- Prandtl number (PR) – a dimensionless variable that is equal to the ratio of momentum diffusivity to thermal diffusivity
- Lid velocity – a dimensionless variable that represents the speed of the moving lid.

It should be noted that the Reynolds number, a common feature in fluid dynamics, is simply the product of the Grashof and Prandtl numbers.

Rather than attempting to completely solve the cavity flow problem outright with Trilinos, the experiments contained in this chapter are solving the sparse linear systems that are created from the lid-driven cavity flow problems. Therefore, the sparse matrices that lie at the heart of this chapter's experiments are extracted from an iteration of the finite difference methods used to solve the lid-driven cavity flow problem. Each of the executions used different parameters in order to create a small suite of differing test matrices. The parameters used to create the matrices from the example PETSc code executions have the following values,

- Mesh size $\in \{(10^2 \times 10^2), (10^3 \times 10^3)\}$
- Grashof number $\in \{10^2, 10^3, 10^4\}$
- Prandtl number $= 1$
- Lid velocity $= 10^{-6}$

The matrices created from the $(10^2 \times 10^2)$ and $(10^3 \times 10^3)$ mesh sizes are square matrices of dimension 4×10^4 and 4×10^6, respectively.

All possible matrix permutations are included in the testing suite; each solver-preconditioner combination is then used to solve the linear system created from the individual test suite matrices. The test suite solves are performed on multiple nodes of two of the aforementioned systems, Comet(2) and Summit(3), as described in Section 6.1 and appendix B. Each node of both Comet(2) and Summit(3) contain 24 total CPU cores. Linear solves for all the test suite linear systems on both systems, for both mesh sizes, take place at $np \in \{96, 192, 384\}$. All the permutations of problem parameters and CPU core counts results in 1260 datapoints per system, a total of 2520.

8.2 Results

The classifiers created for this chapter's experiments are trained and tested on data obtained from both the existing linear systems created from the University of Florida Sparse Matrix Collection, used previously in Chapters 5 to 7, and linear systems created from the lid-driven cavity flow problem introduced in this chapter. For these experiments, two multi-label classifiers were trained and tested:

(1) a classifier trained on the wallclock times, matrix features, and system features associated with the linear systems from the UF Collection, and tested on the same information obtained from the cavity flow problem

(2) a classifier trained on the wallclock times, matrix features, and system features associated with the linear systems of the UF Collection and half of the available cavity flow linear systems, and tested on the same information obtained from the remaining half of the cavity flow problem.

In the context of the cavity flow problems, the data is split in half based on the system on which it was executed. Therefore, the half used in the training of the classifier came from Comet(2), while the other half used for testing came from Summit(3). Creating two multi-label classifiers allows for

a more direct observation of how well the prediction performance of a classifier based solely the UF collection compares to one with the addition of domain specific linear systems.

Following the same basic methodology described in Chapter 7, the experiments of this section are based on the same set of 18 labels in a multi-label classification environment, where any subset of the labels can be assigned to a given input. In practice, these labels can be represented by 18 unique binary classifiers, with each classifier producing a single "true" or "false" output that indicates whether or not the given label is applied to the input vector. Two of the 18 classifiers are simplistic: one determines if the given linear system encountered an error or not, and the other determines if the linear system managed to converge to a solution or not. The remaining 16 binary labels are based on the performance of each specific solver and preconditioner pair on a given linear system with the given hardware and number of processors. These 16 labels are broken down into four distinct groups of labels based on both the wallclock time taken to solve the given linear system, and the hardware used for the solve. The subgroups created based on the hardware are based on comparing the input vector to

(1) other linear solves having the same matrix, computer system, and number of processors (np_and_system)

(2) only other linear solves having the same matrix and computer system, regardless of the number of processors (sys)

(3) only other linear solves having the same matrix and number of processors, regardless of the computer system (np)

(4) all other linear solves having the same matrix, regardless of the computer system or number of processors used (overall).

In general, training a machine learning classifier on one dataset, and then using it to classify input from another dataset, can result in poor prediction performance if there is not enough similarity between the two datasets. Differences in the training and prediction input data can result

in classifiers that are overfit to the training input data such that it does not generalize to any new input data outside of the training set. It is also possible for the prediction input data to be so different from the training data that it's impossible for the classifier to correctly identify it using its existing methods.

Overall, the performance of predicting the best solver-preconditioner pairs for the lid-driven cavity flow problems resulted in significantly poorer results than those in Chapters 5 to 7. In the case of the first classifier, where no cavity flow systems were used as part of the training process, multiple classification techniques from the Scikit-Learn library failed to correctly classify any of the available input vectors as true positive for the 16 timing-based labels. The classification methods that predicted no true positive values include random forest, multilayer perceptron, k-nearest neighbor, gradient boosting, and AdaBoost. The results presented throughout the rest of the section are based on those obtained solely using the random forest classification method. Table 8.1 shows the prediction results of the two classifiers on input vectors where the actual value of the data is true.

Table 8.1: Comparison of the confusion matrix entries predicted by the two multi-label classifiers. The first classifier was trained on data collected solely from matrices of the UF Collection and tested on the cavity flow matrices. The second classifier was trained on combined input matrices collected from both the UF Collection as well as half the available cavity flow matrices. The second classifier was tested on the remaining half of the cavity flow matrices.

	Trained w/ UF				Trained w/ UF+Cavity			
Test	TP	FP	TN	FN	TP	FP	TN	FN
error	292	371	1638	219	123	13	988	136
converged	63	66	1831	560	183	50	853	174
overall	0	0	2508	12	0	0	1249	11
overall_25	0	0	2468	52	0	0	1213	47
overall_50	0	0	2411	109	0	0	1185	47
overall_100	0	0	2311	209	41	6	1135	78
np_0	0	0	2484	36	0	0	1236	24
np_25	0	0	2421	99	0	3	1192	65
np_50	0	0	2328	192	34	4	1145	77
np_100	0	15	2201	304	40	6	1095	119
sys_0	0	0	2496	24	0	0	1248	12
sys_25	0	0	2423	97	28	8	1204	20
sys_50	0	0	2362	158	33	15	1170	42
sys_100	0	0	2252	268	79	24	1116	41
np_and_system_0	0	3	2445	72	1	1	1223	35
np_and_system_25	0	6	2350	164	0	0	1176	84
np_and_system_50	0	0	2267	253	0	0	1130	130
np_and_system_100	0	12	2165	343	73	2	1084	101

Figures 8.2 and 8.3 depict the AUROCs obtained from all 18 labels of both multi-label classifiers. Despite the very poor performance in predicting true positives instances, the AUROC in both classifiers is still relatively high (0.8+) for all labels except the four timing-based labels at the 0% threshold. This suggest that although the general prediction performance is poor, the classifier is able to identify when given two input vectors, one that is actually "true" and one that is actually "false." The scores of AUROCs again follow the general trend of becoming higher with increased threshold percentages. There is also a measurable increase in each label's AUROC when the linear systems from the cavity flow problems are included as part of the data for training the classifier. This suggests that although different computer systems are used for the training and testing cavity flow data, the information is still valuable as an addition to the training information. The AUROC graphs as well as the confusion matrix tables, show how adding specific types of problem domains to the training data increases the prediction performance of those types of datasets. Although not feasible in the frame of this dissertation, it is ideal for the code to add linear systems it has not seen before into its repository, and use it for subsequent training.

Due to the lack of true positive and false positive predictions, multiple metrics, such as the true positive rate, Matthews Correlation Coefficient, and F1 score, are impossible to compute for some of the labels. While both the true negative rate and the accuracy of the classifier are very good for these predictions, they can be misleading due to being completely dominated by their reliance on true negatives as part of the formula.

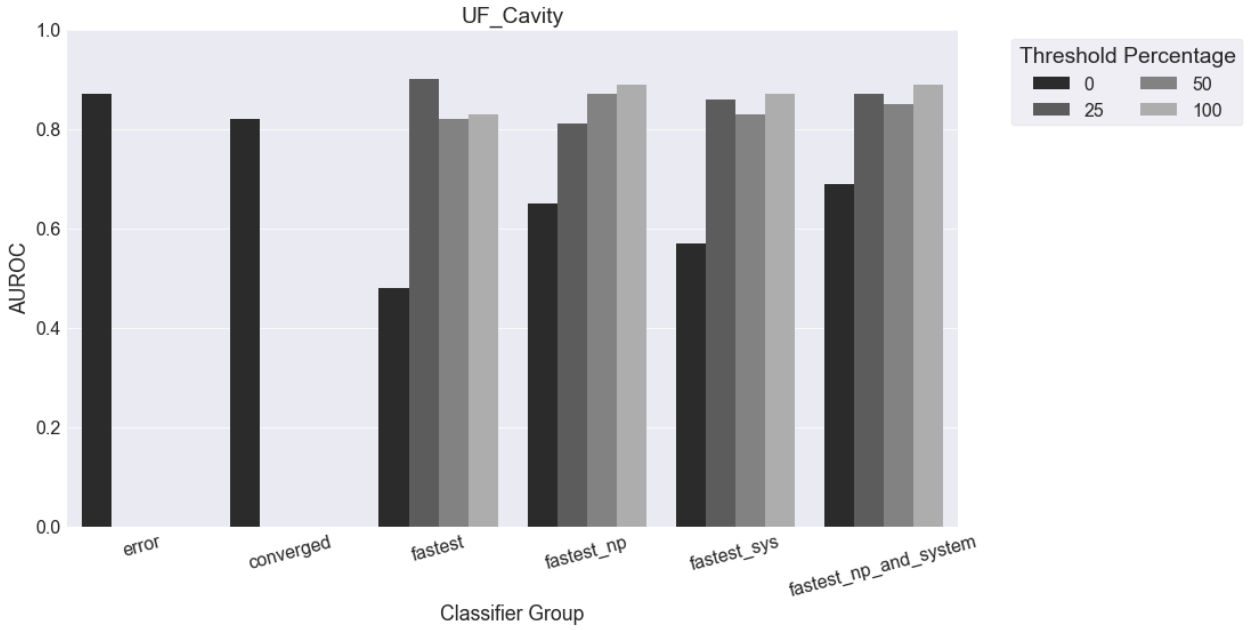


Figure 8.2: Comparison of the area under the receiver operating characteristic curves from classifiers trained on data solely from the UF collection and tested on linear systems from the lid-driven cavity flow experiments, for each of the six groups and four threshold percentages.

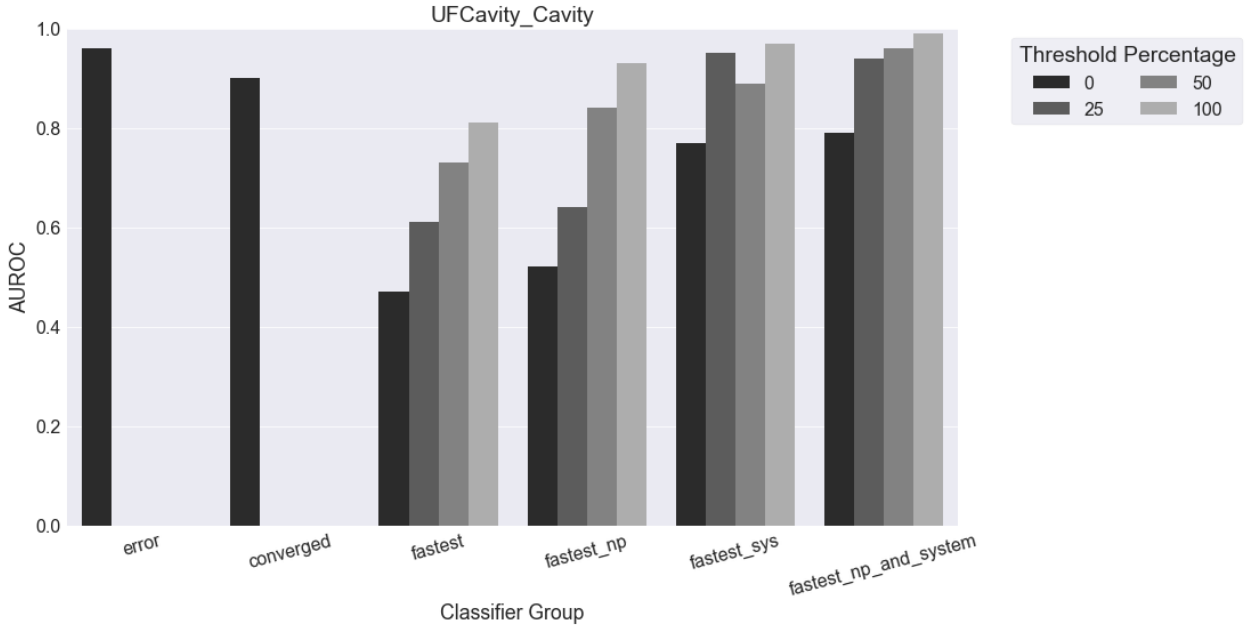


Figure 8.3: Comparison of the area under the receiver operating characteristic curves from classifiers trained on data from both the UF collection and half of the lid-driven cavity flow systems, and tested on the remaining half of the linear systems from the lid-driven cavity flow systems, for each of the six groups and four threshold percentages.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Solving sparse systems of linear equations is a commonly encountered computation in scientific and high-performance computing applications. Selecting the best iterative solver and preconditioner for solving a given sparse linear system, especially for novice users, is not a simple task. This dissertation expanded upon existing work by introducing new techniques that incorporate hardware information into the prediction of ideal iterative linear solver and preconditioners for sparse linear systems. By accounting for hardware, it is possible to create more specially tailored solver-preconditioner suggestions for a novice user. In addition to incorporating hardware information, this dissertation also examined the limitations of generalizing the solver-preconditioner predictions to multiple computer systems and CPU core counts.

Chapter 5 examines the prediction performance degradation observed when using wallclock timing information obtained at one single core count and then attempting to predict the best solver-preconditioner pairs at other core counts. The results show that each core count does not generalize equally to other core counts. The core counts with higher numbers had more similarities with each other than when compared to a serially solved system, as can be seen in Figure 5.10. Having such a discrepancy in similarity also means that not every CPU core count is required in order to maintain a high level of prediction performance. Overall, the timing information recorded for one specific computer system, using a fixed number of CPU cores, results in a loss of prediction performance when extrapolated to other systems and core counts. The randomized lasso algorithm

was used to rank the most important aspects of the linear solves including matrix properties, number of cores used, and the solver-preconditioner pair. The top-ranking features include the column diagonal dominance (100%), solver (100%), number of cores (100%), and the preconditioner (99%).

Chapter 6 examined the prediction performance degradation observed when using wallclock timings obtained on one computer system and then attempting to predict the best solver-preconditioner pairs on other computer systems. Similar to the results of Chapter 5, there is an observable difference in the prediction performance when training on timing information from one computer system and using it to predict the best solver-preconditioner pairs for another. Figures 6.6 and 6.7 show that three of the systems, Summit(3), Stampede(4), and the Laptop(5), can generalize their predictions to each other, while the other two computer systems, Bridges(1) and Comet(2), have little generalization overlap with each other or any other systems. Both of these observations are best depicted in the similarities between Figures 6.6 and 6.7, as well as Table 6.3. The randomized lasso algorithm was used to rank the importance of each of the matrix features as well as the individual hardware features. The top scoring features are almost exactly the same as those found in Chapter 5. However, the four most important hardware features include the bogo_mips measurement (48%), the L3 cache size (48%), the number of GFLOPS achieved when performing a parallel FFT (47.5%), and the minimum latency required to perform ping-pong between two cores (36.5%).

Chapter 7 combines the ideas of Chapters 5 and 6 into a single problem where we predict the best solver-preconditioner pairs for multiple computer systems, each with multiple CPU core counts available. Rather than using the binary classification used in the previous chapters, the problem was expanded into an a multi-label classification problem with 18 possible labels. These labels include error status, convergence status, and multiple timing comparisons with the best-performing solver-preconditioner pairs available for a given system, CPU core count, specific system and CPU count, and overall. Due to the multi-label design, more performance metrics were explored to validate the findings.

Chapter 8 used the ideas and results of Chapter 7 and applied them to a real-world lid-driven cavity flow problem. The prediction performance was much lower than that of the previous chapters,

indicating that the problem type was not similar enough to existing training data to find correlations between the UF-based training data and the cavity flow-based testing data. Incorporating half of the cavity flow data into the training set and then testing the classifier on the remaining half produced improved prediction results.

In general, the 0% threshold proved to be too restrictive, while the 25% threshold provided the single biggest gain in performance. Predicting whether or not a given combination of computer system, number of CPU cores, solver, preconditioner, and sparse matrix, will result in error or converge to a solution are both easier to predict than determining the best solver and preconditioner for a given input combination. The randomized lasso algorithm was used to rank the importance of each of the matrix features, the individual hardware features, and the number of CPU cores used in the solve. Overall, the rankings of the matrix features are somewhat similar to those found in Chapter 6, however there is a large difference in the hardware feature rankings. The number of CPU cores used was determined to be useful in the final classification of input 85.4% of the time, ranking higher than the choice of preconditioner, at 77.3%

In short, this dissertation shows that it is possible to predict the best solver-preconditioner combinations for a given sparse linear system dependent on the hardware used for executing the solve. Additionally, this dissertation demonstrates fundamental problems with existing prediction methods, including the measures of success and the ability to generalize performance. The study's results address these issues by expanding on the measures of classifier prediction success, as well as broadening the applicability of generalized performance results. Overall, this dissertation contributes to the successful continuation of research into improving effectiveness and usability of computer science in these areas.

9.2 Future Work

Going forward, there are many possible avenues for work and research that expands upon the findings of this dissertation.

One of the limiting factors of this work was solving many linear systems in various permutations

in a somewhat reasonable amount of time on the available hardware. While most of the linear systems created from the University of Florida's Sparse Matrix Collection are large enough to be solved on a single node, many others, especially from larger real-world problems, are not. Therefore, two future paths for expanding upon this work are the inclusion of larger training matrices and the computational resources and compute time necessary for solving the resulting linear systems.

While the computer systems used were all unique in some ways, they were all x86-based Intel machines with similar designs. Future experiments might explore additional computer systems based on hardware such as ARM, IBM's BlueGene, Nvidia GPUs, or Intel Xeon Phis.

Although this work begins the exploration of parallelism and how it impacts linear system solves and their prediction, it only takes into account distributed-memory parallelism available through MPI. However, many solvers and preconditioners support shared-memory parallelism using OpenMP, POSIX threads, or similar. Therefore, it is important to explore how shared-memory parallelism changes the performance of solvers and preconditioners on the training set of linear systems, and the subsequent predictions.

Because of time and computational restraints, this work was performed almost exclusively on single nodes of the available computer systems. Being limited to a single node affects how much computation can take place in a given time, therefore limiting the total number and size of the training matrices.

The lid-driven cavity flow problem represents a small percentage of the many possible computational fluid dynamics problems, and an even smaller percentage of all the possible problems using sparse linear systems. Although the UF Collection contains matrices from various problem types and domains, there are always going to be additional matrices that can be added. One possible option is to allow for a submission form of sorts that allows users to upload their own matrices

Machine learning is an incredibly fast-paced field with constantly evolving methods. As techniques are created, it is possible for them to surpass the methods used in this work in terms of prediction performance. Neural networks were briefly explored as part of this dissertation, but the preliminary results were very poor. However, it might be worth re-examining this idea in the future

as the methods continue to grow.

Future work in this area should incorporate multigrid methods [1] as possible choices for solving the sparse linear systems. Methods such as algebraic multigrid can be used as a preconditioning step for other solvers. Due to their performance and ability to help solve sparse linear systems more quickly than other preconditioning methods [89], multigrid options should be included in future testing.

Bibliography

[1] Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia, PA, USA, 2nd edition, 2003.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst.
Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia, PA, 2nd edition, 1994.

[3] Noel M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen. How Fast are Nonsymmetric Matrix Iterations? SIAM Journal on Matrix Analysis and Applications, 13(3):778–795, 1992.

[4] V. Eijkhout and E. Fuentes. Multi-stage Learning of Linear Algebra Algorithms. In International Conference on Machine Learning and Applications, pages 402–407, Dec 2008.

[5] 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.

[6] Erika Fuentes. Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems. PhD thesis, University of Tennessee - Knoxville, December 2007.

[7] J. S. Yeom, J. J. Thiagarajan, A. Bhatele, G. Bronevetsky, and T. Kolev. Data-driven performance modeling of linear solvers for sparse matrices. In 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pages 32–42, Nov 2016.

[8] Sanjiva Weerawarana, Elias N. Houstis, John R. Rice, Anupam Joshi, and Catherine E. Houstis. PYTHIA: A Knowledge-based System to Select Scientific Algorithms. ACM Trans. Math. Softw., 22(4):447–468, December 1996.

[9] John R. Rice and Ronald F. Boisvert. Solving Elliptic Problems Using ELLPACK. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[10] Jack Dongarra and Victor Eijkhout. Self-Adapting Numerical Software for Next Generation Applications. The International Journal of High Performance Computing Applications, 17(2):125–131, 2003.

[11] R Clint Whaley and Antoine Petitet. Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS. Software: Practice and Experience, 35(2):101–121, Feb 2005.

[12] L.S. Blackford et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). ACM Transactions on Mathematical Software, 28(2):135–151, June 2002.

[13] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of Machine Learning to the Selection of Sparse Linear Solvers. The International Journal of High Performance Computing Applications, 2006.

[14] Sanjukta Bhowmick, Brice Toth, and Padma Raghavan. Towards Low-Cost, High-Accuracy Classifiers for Linear Solver Selection. Computational Science–ICCS 2009, pages 463–472, 2009.

[15] Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes, and David Keyes. Application of alternating decision trees in selecting sparse linear solvers. In Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors, Software Automatic Tuning: From Concepts to State-of-the-Art Results, pages 153–173. Springer New York, New York, NY, 2010.

[16] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[17] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems. ACM Trans. Math. Software, 31(3):351–362, 2005.

[18] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. SIAM, Philadelphia, PA, 3rd edition, 1999.

[19] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An Overview of the Trilinos Project. ACM Transactions on Mathematical Software, 31(3):397–423, 2005.

[20] Ronan Collobert and Jason Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In Proceedings of the 25th International Conference on Machine Learning, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. CoRR, abs/1301.3781, 2013.

[22] Achim Basermann, Uwe Jaekel, M Nordhausen, and Koutaro Hachiya. Parallel Iterative Solvers for Sparse Linear Systems in Circuit Simulation. Future Generation Computer Systems, 21(8):1275–1284, Oct 2005.

[23] Michael Schacht Hansen and Thomas Sangild Sørensen. Gadgetron: An Open Source Framework for Medical Image Reconstruction. Magnetic Resonance in Medicine, 69(6):1768–1776, 2013.

[24] S H Langer, I Karlin, V A Dobrev, M L Stowell, and M E Kumbera. Performance analysis and optimization for blast, a high order finite element hydro code. In NECDC, Jan 2015.

[25] F. Tracy, T. Oppe, and S. Gavali. Testing parallel linear iterative solvers for finite element groundwater flow problems. In Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference, HPCMP-UGC '07, pages 474–481, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Lloyd N. Trefethen and David Bau. Numerical Linear Algebra. SIAM, 1997.

[27] Gene H. Golub and Charles F. Van Loan. Matrix Computations. The Johns Hopkins University Press, Baltimore, 4th edition, 2013.

[28] Ilse C. F. Ipsen and Carl D. Meyer. The idea behind krylov methods. The American Mathematical Monthly, 105(11):889–899, 1998.

[29] Christopher C. Paige and Michael A. Saunders. Lsqr: An algorithm for sparse linear equations and sparse least squares. ACM Trans. Math. Softw., 8(1):43–71, March 1982.

[30] M. A. Pai and H. Dag. Iterative solver techniques in large scale power system computation. In Proceedings of the 36th IEEE Conference on Decision and Control, volume 4, pages 3861–3866 vol.4, Dec 1997.

[31] A. Gil, J. Segura, and N. Temme. Numerical Methods for Special Functions. Society for Industrial and Applied Mathematics, 2007.

[32] Trilinos Home Page. HTTPS://TRILINOS.ORG, 2016.

[33] Xiaoye S. Li. An Overview of SuperLU: Algorithms, Implementation, and User Interface. ACM Transactions on Mathematical Software Software, 31(3):302–325, September 2005.

[34] MUltifrontal Massively Parallel Solver (MUMPS 5.0.2) Users' Guide. HTTP://MUMPS.ENSEEIHT.FR/DOC/USERGUIDE_5.0.2.PDF, 2016.

[35] Mark Hoemmen. Tpetra Project Overview. High Performance Computing Operational Review (HPCOR) Workshop, 2015.

[36] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[37] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos. Journal of Parallel and Distributed Computing, 74(12):3202–3216, December 2014.

[38] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering, 5(1):46–55, January 1998.

[39] David R. Butenhof. Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[40] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. Queue, 6(2):40–53, March 2008.

[41] Belos. HTTPS://TRILINOS.ORG/PACKAGES/BELOS/, 2016.

[42] Eric Bavier, Mark Hoemmen, Sivasankaran Rajamanickam, and Heidi Thornquist. Amesos2 and Belos: Direct and Iterative Solvers for Large Sparse Linear Systems. Scientific Programming, 20(3):241–255, July 2012.

[43] Ifpack2. HTTPS://TRILINOS.ORG/PACKAGES/IFPACK2/, 2016.

[44] Andrey Prokopenko, Christopher M. Siefert, Jonathan J. Hu, Mark Hoemmen, and Alicia Klinvex. Ifpack2 User's Guide 1.0. Technical Report SAND2016-5338, Sandia National Labs, 2016.

[45] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The Elements of Statistical Learning. Springer-Verlag New York, Inc., New York, NY, USA, 2nd edition, 2009.

[46] Peter Flach. Machine Learning: The Art and Science of Algorithms That Make Sense of Data. Cambridge University Press, New York, NY, USA, 2012.

[47] Kevin P. Murphy. Machine Learning: A Probabilistic Perspective. Massachusetts Institute of Technology, Cambridge, MA, USA, 2012.

[48] Ben Gorman. Gradient Boosting Explained. HTTPS://GORMANALYSIS.COM/GRADIENT-BOOSTING-EXPLAINED/, Jan 2017.

[49] Robert Tibshirani. Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society. Series B (Methodological), 58(1):267–288, 1996.

[50] Sijian Wang, Bin Nan, Saharon Rosset, and Ji Zhu. Random lasso. Ann. Appl. Stat., 5(1):468–485, 03 2011.

[51] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, New York, NY, USA, 2006. ACM.

[52] A. Petitet, R.C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. HTTP://WWW.NETLIB.ORG/BENCHMARK/HPL/, 2016.

[53] LINPACK. HTTP://WWW.NETLIB.ORG/LINPACK/, 2016.

[54] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK Users' Guide. SIAM, Jan 1997.

[55] Univ. of Tennessee, Univ. of California Berkeley, Univ. of Colorado Denver, and NAG Ltd. DGEMM. HTTP://WWW.NETLIB.ORG/LAPACK/EXPLORE-HTML/D7/D2B/DGEMM_8F.HTML, 2015.

[56] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[57] STREAM FAQs. http://www.cs.virginia.edu/stream/ref.html#size, 2016.

[58] PARKBENCH (PARallel Kernels and BENCHmarks). http://www.netlib.org/parkbench/, 2016.

[59] RandomAccess. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/, 2016.

[60] M. Heideman, D. Johnson, and C. Burrus. Gauss and the History of the Fast Fourier Transform. IEEE ASSP Magazine, 1(4):14–21, October 1984.

[61] Daisuke Takahashi. FFTE: A Fast Fourier Transform Package. http://www.ffte.jp/, 2016.

[62] Rolf Rabenseifner and Gerrit Schulz. Effective bandwidth (b_{eff}) benchmark. https://fs.hlrs.de/projects/par/mpi//b_eff/, 2016.

[63] Victor Eijkhout and Erika Fuentes. A Standard and Software for Numerical Metadata. ACM Trans. Math. Softw., 35(4):25:1–25:20, February 2009.

[64] Lighthouse Github. https://github.com/LighthouseHPC/lighthouse/, 2017.

[65] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. ACM Trans. Math. Softw., 38(1):1:1–1:25, December 2011.

[66] GCC, the gnu compiler collection. https://gcc.gnu.org/, 2017.

[67] Boost C++ Libraries. http://www.boost.org/, 2017.

[68] GCC v6.2. https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/, 2017.

[69] Open MPI: Open source high performance computing. https://www.open-mpi.org/software/ompi/v1.10/, 2017.

[70] OpenBLAS: An optimized blas library. http://www.openblas.net/, 2017.

[71] Boost v1.60. http://www.boost.org/doc/libs/1_60_0/, 2017.

[72] Trilinos v12.8. trilinos.org/download/previous-releases/download-12-8/, 2017.

[73] W. McKinney. Data Structures for Statistical Computing in Python. In Proceedings of the 9^{th} Python in Science Conference, pages 51–56, 2010.

[74] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The Best of Both Worlds. Computing in Science Engineering, 13(2):31 –39, 2011.

[75] University of Colorado. Summit. https://www.rc.colorado.edu/resources/compute/summit, 2017.

[76] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[77] Ethem Alpaydin. Introduction to Machine Learning. The MIT Press, 2014.

[78] Charles E. Metz. Basic Principles of ROC Analysis. Seminars in Nuclear Medicine, 8(4):283 – 298, 1978.

[79] Alex Rogozhnikov. ROC Curve Demonstration. http://arogozhnikov.github.io/2015/10/05/roc-curve.html, Oct 2015.

[80] J A Hanley and B J McNeil. The Meaning and Use of the Area Under a Receiver Operating Characteristic (ROC) Curve. Radiology, 143(1):29–36, 1982. PMID: 7063747.

[81] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. SIGKDD Explor. Newsl., 6(1):20–29, June 2004.

[82] Scikit-Learn Developers. Stratified Shuffle Split. http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html, 2016.

[83] Sam Varshavchik. lscpu. http://manpages.courier-mta.org/htmlman1/lscpu.1.html, 2016.

[84] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. XSEDE: Accelerating Scientific Discovery. Computing in Science & Engineering, 16(5):62–74, 2014.

[85] David Martin Powers. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. Journal of Machine Learning Technologies, 2(1):37 – 63, 2011.

[86] U Ghia, K.N Ghia, and C.T Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. Journal of Computational Physics, 48(3):387 – 411, 1982.

[87] Carlos Henrique Marchi, Roberta Suero, and Luciano Kiyoshi Araki. The lid-driven square cavity flow: numerical solution with a 1024 x 1024 grid. Journal of the Brazilian Society of Mechanical Sciences and Engineering, 31:186 – 198, 09 2009.

[88] CFD Education Center. Lid-driven cavity flow. http://cfdblogvienna.blogspot.com/p/computational-fluid-dynamics_12.html.

[89] Aditi Ghai, Cao Lu, and Xiangmin Jiao. A comparison of preconditioned krylov subspace methods for large-scale nonsymmetric linear systems, 2016.

[90] Bridge's System Configuration. https://www.psc.edu/bridges/user-guide/system-configuration, 2016.

[91] Comet User Guide. http://www.sdsc.edu/support/user_guides/comet.html, 2017.

[92] Stampede User Guide. https://portal.tacc.utexas.edu/user-guides/stampede, 2017.

Appendix A

Feature Definitions

A.1 Matrix Features

(1) rows – Number of rows m in the coefficient matrix A.

(2) cols – Number of columns n in the coefficient matrix A.

(3) min_nnz_row – Minimum number of nonzero entries found in a single row of A.

(4) row_var – Largest variance computed from each row of A.

$$max(\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}).$$

(5) col_var – Largest variance computed for each column of A.

(6) diag_var – Variance computed on the vector of diagonal entries in A.

(7) nnz – Total number of nonzero entries in A.

(8) frob_norm – The Frobenius norm of the matrix A.

$$\|A\|_F = \sqrt{\sum_{j=1}^{\infty}\sum_{i=1}^{\infty}|{a_{ij}}^2|}$$

(9) symm_frob_norm – The Frobenius norm of the symmetric matrix.

$$S = \frac{A + A^T}{2}$$

$$\|A\|_F = \sqrt{\sum_{j=1}^{\infty}\sum_{i=1}^{\infty}|{s_{ij}}^2|}$$

(10) antisymm_frob_norm – The Frobenius norm of the symmetric matrix

$$S_A = \frac{A - A^T}{2}$$

.

$$\|A\|_F = \sqrt{\sum_{j=1}^{\infty}\sum_{i=1}^{\infty} |s_A ij^2|}$$

(11) one_norm – The one-norm of the matrix A.

$$\|A\|_1 = \max_{1 \le j \le n} \sum i = 1^n \|a_{ij}\|$$

(12) inf_norm – The infinity-norm of the matrix A.

$$\|A\|_\infty = \max_{1 \le i \le n} \sum j = 1^n \|a_{ij}\|$$

(13) symm_inf_norm – The infinity-norm of the symmetric matrix.

$$S = \frac{A + A^T}{2}$$

$$\|S\|_\infty = \max_{1 \le i \le n} \sum j = 1^n \|S_{ij}\|$$

(14) antisymm_inf_norm – The infinity-norm of the symmetric matrix.

$$S_A = \frac{A - A^T}{2}$$

$$\|S_A\|_\infty = \max_{1 \le i \le n} \sum j = 1^n \|S_A ij\|$$

(15) max_nnz_row – Maximum number of nonzero entries contained in a single row of the matrix A.

(16) trace – Sum of the diagonal entries contained in A.

(17) abs_trace – Sum of the absolute values of the diagonal entries contained in A.

(18) min_nnz_row – Minimum number of nonzero entries contained in a single row of the matrix A.

(19) avg_nnz_row – The average number of nonzero entries in a given row of the matrix A.

(20) dummy_rows – The number of rows in A containing only one nonzero entry.

(21) dummy_rows_kind – The dummy rows kind of A can be one of the following three outputs:

(a) Every dummy row of A contains a 1 along the diagonal of the matrix.

(b) Every dummy row has nonzero entries along the diagonal.

(c) At least one dummy row's entry is not on the diagonal.

(22) num_value_symm_1 – Whether or not $A = A^T$.

(23) nnz_pattern_symm_1 – Whether or not all nonzero positions, a_{ij}, in A also contain a nonzero position in the matrix A^T.

(24) num_value_symm_2 – The soft numeric value symmetry of A. S is the symmetric portion of A.

$$S = \frac{A + A^T}{2}$$

$$1 - \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} |s_{ij}|}{2 \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|}$$

(25) nnz_pattern_symm_2 – The percentage of nonzero entries in A at a_{ij} that have corresponding nonzero entries in A^T.

$$f(a_{ij}) = \begin{cases} 1, & \text{if } a_{ij}, a_{ji} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{m} \sum_{j=1}^{n} f(a_{ij})}{nnz}$$

(26) row_diag_dom – A property related to the absolute values of the diagonal entries in a

matrix compared to other entries in the row.

$$row\_diag\_dom = \begin{cases} 0, & \text{if for any row } i, a_{ii} < \sum_{j \neq i} |a_{ij}| \\ 1, & \text{if for all rows } i, a_{ii} \geq \sum_{j \neq i} |a_{ij}| \\ 2, & \text{if for all rows } i, a_{ii} > \sum_{j \neq i} |a_{ij}| \end{cases}$$

(27) col_diag_dom – A property related to the absolute values of the diagonal entries in a matrix compared to other entries in the column.

$$row\_diag\_dom = \begin{cases} 0, & \text{if for any column } j, a_{jj} < \sum_{i \neq j} |a_{ij}| \\ 1, & \text{if for all columns } j, a_{jj} \geq \sum_{i \neq j} |a_{ij}| \\ 2, & \text{if for all columns } j, a_{jj} > \sum_{i \neq j} |a_{ij}| \end{cases}$$

(28) diag_avg – The arithmetic mean of the diagonal entries of A.

(29) diag_sign –This property indicates the diagonal sign pattern.

$$diag\_sign = \begin{cases} -2, & \text{all diagonal entries}, a_{ii} < 0 \\ -1, & \text{all diagonal entries}, a_{ii} \leq 0 \\ 0, & \text{all diagonal entries}, a_{ii} = 0 \\ 1, & \text{all diagonal entries}, a_{ii} \geq 0 \\ 2, & \text{all diagonal entries}, a_{ii} > 0 \\ 3, & \text{otherwise} \end{cases}$$

(30) diag_nnz – The number of nonzero diagonal entries in A.

(31) lower_bw – The smallest number c such that $a_{ij} = 0$ when $i > j + c$.

(32) upper_bw – The smallest number c such that $a_{ij} = 0$ when $i > j - c$.

(33) row_log_val_spread – The maximum ratio between a row's minimum and maximum entries.

$$\max_i \log_{10} \frac{\max_j |a_{ij}|}{\min_j |a_{ij}|}$$

(34) col_log_val_spread – The maximum ratio between a column's minimum and maximum entries.

$$\max_j \log_{10} \frac{\max_i |a_{ij}|}{\min_i |a_{ij}|}$$

A.2 System Information Features

(1) HPL_Tflops – Performance of solving a randomly generated dense linear system of double floating-point precision using MPI, measured in teraflops per second. The solving method is LU factorization with partial row pivoting.

(2) StarDGEMM_Gflops – The floating-point execution rate, in gigaflops per second, of double precision real matrix-matrix multiply using the DGEMM BLAS routine on multiple cores. The final result is the arithmetic mean of the performance of each core.

(3) SingleDGEMM_Gflops – The floating-point execution rate, in gigaflops per second, of double precision real matrix-matrix multiply using the DGEMM BLAS routine on a single core.

(4) PTRANS_GBs – Performs a parallel matrix transpose in a block-cyclic manner between two cores. The result is measured in gigabytes per second.

(5) MPIRandomAccess_LCG_GUPs – Number of memory locations that are updated per second with randomized integers generated using the LCG algorithm. The communication is performed in an all-to-all manner. Measured in "giga-updates" per second.

(6) MPIRandomAccess_GUPs – Number of memory locations that are updated per second with randomized integers. The communication is performed in an all-to-all manner. Measured in "giga-updates" per second.

(7) StarRandomAccess_LCG_GUPs – Number of memory locations that are updated per second with randomized integers generated using the LCG algorithm. No communication

occurs and each core is running in an embarrassingly parallel fashion. Measured in average number of "giga-updates" per second.

(8) StarRandomAccess_GUPs – Number of memory locations that are updated per second with randomized integers. No communication occurs and each core is running in an embarrassingly parallel fashion. Measured in average number of "giga-updates" per second.

(9) SingleRandomAccess_LCG_GUPs – Number of memory locations that are updated per second with randomized integers generated using the LCG algorithm. No communication occurs and the code is executed on a single core. Measured in number of "giga-updates" per second.

(10) SingleRandomAccess_GUPs – Number of memory locations that are updated per second with randomized integers. No communication occurs and the code is executed on a single core. Measured in number of "giga-updates" per second.

(11) StarSTREAM_Copy – Sustainable memory bandwidth measured in gigabytes per second based on copying vectors in an embarrassingly parallel manner.

(12) StarSTREAM_Scale – Sustainable memory bandwidth measured in gigabytes per second based on scaling vectors in an embarrassingly parallel manner.

(13) StarSTREAM_Add – Sustainable memory bandwidth measured in gigabytes per second based on adding vectors in an embarrassingly parallel manner.

(14) StarSTREAM_Triad – Sustainable memory bandwidth measured in gigabytes per second based on adding and scaling vectors in an embarrassingly parallel manner.

(15) SingleSTREAM_Copy – Sustainable memory bandwidth measured in gigabytes per second based on copying vectors on a single core.

(16) SingleSTREAM_Scale – Sustainable memory bandwidth measured in gigabytes per second based on scaling vectors on a single core.

(17) SingleSTREAM_Add – Sustainable memory bandwidth measured in gigabytes per second based on adding two vectors on a single core.

(18) SingleSTREAM_Triad – Sustainable memory bandwidth measured in gigabytes per second based on adding and scaling vectors on a single core.

(19) StarFFT_Gflops – Performs double precision complex one-dimensional Discrete Fourier Transform measured in gigaflops per second. Performed in an embarassingly parallel manner across the available cores and the average performance is returned.

(20) SingleFFT_Gflops – Performs double precision complex one-dimensional Discrete Fourier Transform measured in gigaflops per second.

(21) MPIFFT_Gflops – Performs distributed double precision complex one-dimensional Discrete Fourier Transform across all available cores. The overall performance is measured in gigaflops per second.

(22) MaxPingPongLatency_usec – Maximum latency for a number of non-simultaneous ping-pong tests. The ping-pongs are performed between as many as possible distinct pairs of processors using MPI send and receive routines. The results are measured in micro-seconds.

(23) MinPingPongLatency_usec – Minimum latency for a number of non-simultaneous ping-pong tests. The ping-pongs are performed between as many as possible distinct pairs of processors using MPI send and receive routines. The results are measured in micro-seconds.

(24) AvgPingPongLatency_usec – Average latency for a number of non-simultaneous ping-pong tests. The ping-pongs are performed between as many as possible distinct pairs of processors using MPI send and receive routines. The results are measured in micro-seconds.

(25) RandomlyOrderedRingLatency_usec – Latency in the ring communication pattern using randomly ordered processes. The result is averaged over various iterations and measured in

micro-seconds.

(26) MinPingPongBandwidth_GBytes – Minimum bandwidth for a number of non-simultaneous ping-pong tests. The ping-pongs are performed between as many as possible distinct pairs of processors using MPI send and receive routines. The results are measured in gigabytes per second.

(27) NaturallyOrderedRingBandwidth_GBytes – Bandwidth achieved in the ring communication pattern. The ring is formed with consecutive processes in the MPI_COMM_WORLD and measured in gigabytes per second.

(28) RandomlyOrderedRingBandwidth_GBytes – Bandwidth achieved in the ring communication pattern. The ring is formed with randomly ordered processes in the MPI_COMM_WORLD and measured in gigabytes per second.

(29) MaxPingPongBandwidth_GBytes – Maximum bandwidth for a number of non-simultaneous ping-pong tests. The ping-pongs are performed between as many as possible distinct pairs of processors using MPI send and receive routines. The results are measured in gigabytes per second.

(30) AvgPingPongBandwidth_GBytes – Average latency for a number of non-simultaneous ping-pong tests. The ping-pongs are performed between as many as possible distinct pairs of processors using MPI send and receive routines. The results are measured in gigabytes per second.

(31) NaturallyOrderedRingLatency_usec – Latency in the ring communication pattern using consecutively ordered processes. The result is averaged over various iterations and measured in micro-seconds.

(32) MemProc – Amount of memory available per core.

(33) core_count – The number of physical CPU cores available.

(34) cpu_freq – Clock speed of the CPU.

(35) bogo_mips – Unscientific measure of a CPU's ability to perform a "busy loop."

(36) l1_cache – Size of each processor's L1 cache.

(37) l2_cache – Size of each processor's L2 cache.

(38) l3_cache – Size of each processor's L3 cache.

(39) memory_size – Size, in gigabytes, of the available RAM.

(40) memory_freq – Clock speed, in gigahertz, of the system's RAM.

(41) memory_type – The type of memory available in the system, DDR3 or DDR4.

Appendix B

Hardware Information

B.1 Computer Hardware

Five different clusters were primarily used for the experiments described in this dissertation. The details of each computer system are depicted below. The complete HPC Challenge benchmark results for each computer system are shown in Appendix B.2.

Table B.1: Hardware Information for each node in PSC's Bridges [90].

Hardware	Specifications
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	28
On-line CPU(s) list:	0-27
Thread(s) per core:	1
Core(s) per socket:	14
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Model name:	Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
Stepping:	2
CPU MHz:	2300.000
BogoMIPS:	4600.89
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	35840K
NUMA node0 CPU(s):	0-6,14-20
NUMA node1 CPU(s):	7-13,21-27
RAM:	128GB DDR4

Table B.2: Hardware information for each node in SDSC's Comet [91].

Hardware	Specifications
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	24
On-line CPU(s) list	0-23
Thread(s) per core	1
Core(s) per socket	12
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	63
Stepping	2
CPU MHz	2501.000
BogoMIPS	4988.09
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	30720K
NUMA node0 CPU(s)	0-11
NUMA node1 CPU(s)	12-23
RAM:	128GB DDR4

Table B.3: Hardware information for each node in TACC's Stampede [92].

Hardware	Specifications
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	16
On-line CPU(s) list	0-15
Thread(s) per core	1
Core(s) per socket	8
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	45
Model name	Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz
Stepping	7
CPU MHz	2700.249
BogoMIPS	5399.28
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	20480K
NUMA node0 CPU(s)	0,2,4,6,8,10,12,14
NUMA node1 CPU(s)	1,3,5,7,9,11,13,15
RAM:	32GB DDR3

Table B.4: Hardware information for each node in the University of Colorado's Summit [75].

Hardware	Specifications
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	24
On-line CPU(s) list	0-23
Thread(s) per core	1
Core(s) per socket	12
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	63
Model name	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
Stepping	2
CPU MHz	2494.329
BogoMIPS	4992.81
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	30720K
NUMA node0 CPU(s)	0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s)	1,3,5,7,9,11,13,15,17,19,21,23
RAM:	128GB DDR4

Table B.5: Hardware information for my personal Dell XPS13 Laptop.

Hardware	Specifications
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	4
On-line CPU(s) list	0-3
Thread(s) per core	2
Core(s) per socket	2
Socket(s)	1
NUMA node(s)	1
Vendor ID	GenuineIntel
CPU family	6
Model	61
Model name	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
Stepping	4
CPU MHz	2200.000
CPU max MHz	2700.0000
CPU min MHz	500.0000
BogoMIPS	4389.76
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	3072K
NUMA node0 CPU(s)	0-3
RAM:	8GB DDR3

B.2 HPC Challenge Results

Table B.6: All the available output from the HPC Challenge benchmarks for the five systems used in this work.

	Bridges	Comet	Stampede	Summit	Laptop
VersionMajor	1	1	1	1	1
VersionMinor	5	5	5	5	5
VersionMicro	0	0	0	0	0
VersionRelease	f	f	f	f	f
LANG	C	C	C	C	C
Success	1	0	1	1	1
sizeof_char	1	1	1	1	1
sizeof_short	2	2	2	2	2
sizeof_int	4	4	4	4	4
sizeof_long	8	8	8	8	8
sizeof_void_ptr	8	8	8	8	8
sizeof_size_t	8	8	8	8	8
sizeof_float	4	4	4	4	4
sizeof_double	8	8	8	8	8
sizeof_s64Int	8	8	8	8	8
sizeof_u64Int	8	8	8	8	8
sizeof_struct_double_double	16	16	16	16	16
CommWorldProcs	28	24	16	24	4
MPI_Wtick	6.54E-10	4.00E-10	3.70E-10	4.01E-10	5.44E-10
HPL_Tflops	0.215951	0.376918	0.294586	0.53264	0.035
HPL_time	505.835	81.4316	161.057	163.317	163.726
HPL_eps	1.11E-16	1.11E-16	1.11E-16	1.11E-16	1.11E-16
HPL_RnormI	8.11E-10	NaN	5.80E-10	1.76E-10	9.614E-11
HPL_Anorm1	13832.6	9069.59	10478.2	12816.5	5241.06
HPL_AnormI	13822.6	9071.88	10482.8	12815	5242.6
HPL_Xnorm1	69854.4	NaN	31254.8	16722	13351.5
HPL_XnormI	7.91617	NaN 4.33342	1.84384	3.57249	
HPL_BnormI	0.499999	0.499992	0.499986	0.499997	0.499999
HPL_N	54720	35840	41440	50720	20640
HPL_NB	80	80	80	80	80
HPL_nprow	4	4	4	4	2
HPL_npcol	7	6	4	6	2
HPL_depth	1	1	1	1	1
HPL_nbdiv	2	2	2	2	2
HPL_nbmin	4	4	4	4	4
HPL_cpfact	R	R	R	R	R
HPL_crfact	C	C	C	C	C
HPL_ctop	1	1	1	1	1
HPL_order	C	C	C	C	C
HPL_dMACH_EPS	1.11E-16	1.11E-16	1.11E-16	1.11E-16	1.11-16
HPL_dMACH_SFMIN	0.00E+00	0.00E+00	0.00E+00	0.00E+00	2.22-308
HPL_dMACH_BASE	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2
HPL_dMACH_PREC	2.22E-16	2.22E-16	2.22E-16	2.22E-16	2.22-16
HPL_dMACH_MLEN	5.30E+01	5.30E+01	5.30E+01	5.30E+01	53
HPL_dMACH_RND	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1

HPL_dMACH_EMIN	-1.02E+03	-1.02E+03	-1.02E+03	-1.02E+03	-1021
HPL_dMACH_RMIN	0.00E+00	0.00E+00	0.00E+00	0.00E+00	2.22E-308
HPL_dMACH_EMAX	1.02E+03	1.02E+03	1.02E+03	1.02E+03	1024
HPL_dMACH_RMAX	1.79E+308	1.79E+308	1.79E+308	1.79E+308	1.79E+308
HPL_sMACH_EPS	5.96E-08	5.96E-08	5.96E-08	5.96E-08	5.96E-08
HPL_sMACH_SFMIN	1.18E-38	1.18E-38	1.18E-38	1.18E-38	1.175E-38
HPL_sMACH_BASE	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2
HPL_sMACH_PREC	1.19E-07	1.19E-07	1.19E-07	1.19E-07	1.19E-07
HPL_sMACH_MLEN	2.40E+01	2.40E+01	2.40E+01	2.40E+01	24
HPL_sMACH_RND	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1
HPL_sMACH_EMIN	-1.25E+02	-1.25E+02	-1.25E+02	-1.25E+02	-125
HPL_sMACH_RMIN	1.18E-38	1.18E-38	1.18E-38	1.18E-38	1.175E-38
HPL_sMACH_EMAX	1.28E+02	1.28E+02	1.28E+02	1.28E+02	128
HPL_sMACH_RMAX	3.40E+38	3.40E+38	3.40E+38	3.40E+38	3.40E+038
dweps	1.11E-16	1.11E-16	1.11E-16	1.11E-16	1.11E-16
sweps	5.96E-08	5.96E-08	5.96E-08	5.96E-08	5.96E-08
HPLMaxProcs	28	24	16	24	4
HPLMinProcs	28	24	16	24	4
DGEMM_N	5969	4223	5980	5976	5957
StarDGEMM_Gflops	19.807	18.4259	21.5235	28.3834	11.0092
SingleDGEMM_Gflops	33.0807	21.2062	23.0727	34.2565	15.563
PTRANS_GBs	3.57253	4.02147	4.71523	10.8949	0.850288
PTRANS_time	1.67628	0.260291	0.696458	0.472242	0.894743
PTRANS_residual	0	0	0	0	0
PTRANS_n	27360	17920	20720	25360	10320
PTRANS_nb	80	80	80	80	80
PTRANS_nprow	4	4	4	4	2
PTRANS_npcol	7	6	4	6	2
MPIRandomAccess_LCG_N	2147483648	1073741824	1073741824	2147483648	268435456
MPIRandomAccess_LCG_time	39.4521	60.6271	67.9955	59.352	64.641
MPIRandomAccess_LCG_CheckTime	12.955	8.847	43.5388	13.1635	5.02296
MPIRandomAccess_LCG_Errors	0	0	387	0	0
MPIRandomAccess_LCG_ErrorsFraction	0	0	3.60E-07	0	0
MPIRandomAccess_LCG_ExeUpdates	2714570936	3903927360	3362514544	5851117992	323141296
MPIRandomAccess_LCG_GUPs	0.0688067	0.0643925	0.049452	0.0985833	0.00499901
MPIRandomAccess_LCG_TimeBound	60	60	60	60	60
MPIRandomAccess_LCG_Algorithm	0	0	0	0	0
MPIRandomAccess_N	2147483648	1073741824	1073741824	2147483648	268435456
MPIRandomAccess_time	39.2777	57.6682	50.9653	59.4248	59.4625
MPIRandomAccess_CheckTime	13.1772	8.01242	10.2463	14.132	4.56017
MPIRandomAccess_Errors	0	0	0	0	0
MPIRandomAccess_ErrorsFraction	0	0	0	0	0
MPIRandomAccess_ExeUpdates	2750951700	3674209032	2912731856	5785879512	320918972
MPIRandomAccess_GUPs	0.0700385	0.0637129	0.0571513	0.0973646	0.005397
MPIRandomAccess_TimeBound	60	60	60	60	60
MPIRandomAccess_Algorithm	0	0	0	0	0
RandomAccess_LCG_N	67108864	33554432	67108864	67108864	67108864
StarRandomAccess_LCG_GUPs	0.012859	0.033345	0.0228886	0.0334192	0.0266994
SingleRandomAccess_LCG_GUPs	0.0626308	0.0648788	0.0372958	0.0717821	0.0604318
RandomAccess_N	67108864	33554432	67108864	67108864	67108864
StarRandomAccess_GUPs	0.0128778	0.0334041	0.0230922	0.0335016	0.0228788
SingleRandomAccess_GUPs	0.0612444	0.0648515	0.0374511	0.0716084	0.056846
STREAM_VectorSize	35646171	17840355	35776533	35729422	35500800
STREAM_Threads	1	1	1	1	1

StarSTREAM_Copy	1.63992	4.21272	4.57726	3.69016	5.08263
StarSTREAM_Scale	1.38328	3.19541	3.30669	2.92074	3.53951
StarSTREAM_Add	1.5253	3.60064	3.76346	3.33642	3.85717
StarSTREAM_Triad	1.5853	3.61897	3.73212	3.34267	3.85697
SingleSTREAM_Copy	6.95636	19.3064	7.58094	5.03965	16.0505
SingleSTREAM_Scale	5.0235	12.0275	13.1603	5.81145	9.4218
SingleSTREAM_Add	5.56274	13.1324	14.1851	5.91912	9.93453
SingleSTREAM_Triad	5.60783	13.1487	14.2154	5.92306	9.91911
FFT_N	16777216	8388608	16777216	16777216	16777216
StarFFT_Gflops	1.04898	1.93068	1.7685	1.84404	1.1024
SingleFFT_Gflops	1.33642	2.73676	2.69306	2.4645	1.94941
MPIFFT_N	134217728	67108864	134217728	134217728	33554432
MPIFFT_Gflops	8.99739	17.9299	10.1954	14.1991	2.06849
MPIFFT_maxErr	2.43E-15	2.16E-15	2.31E-15	2.43E-15	2.24E-15
MPIFFT_Procs	16	16	16	16	4
MaxPingPongLatency_usec	0.977552	0.580369	0.479631	0.488717	0.700395
RandomlyOrderedRingLatency_usec	1.64323	0.572225	0.532336	0.701083	0.679061
MinPingPongBandwidth_GBytes	4.96149	5.36232	5.95163	11.2696	5.03606
NaturallyOrderedRingBandwidth_GBytes	0.726378	0.812646	0.996609	1.44221	1.3016
RandomlyOrderedRingBandwidth_GBytes	0.768784	1.23318	1.32789	1.56437	1.38813
MinPingPongLatency_usec	0.372814	0.388025	0.283691	0.312606	0.477783
AvgPingPongLatency_usec	0.565746	0.467398	0.382727	0.383007	0.592444
MaxPingPongBandwidth_GBytes	12.0903	9.54204	9.83478	15.4459	8.39117
AvgPingPongBandwidth_GBytes	9.50937	7.2649	7.75271	14.7139	6.76401
NaturallyOrderedRingLatency_usec	1.50814	0.624638	0.579283	0.664996	0.679178
FFTEnblk	16	16	16	16	16
FFTEnp	8	8	8	8	8
FFTEl2size	1048576	1048576	1048576	1048576	1048576
M_OPENMP	201511	201511	201511	201511	201511
omp_get_num_threads	1	1	1	1	1
omp_get_max_threads	1	1	1	1	1
omp_get_num_procs	14	12	8	12	4
MemProc	1024	512	1024	1024	1024
MemSpec	3	3	3	3	3
MemVal	1024	512	1024	1024	1024
MPIFFT_time0	1.39E-06	1.80E-06	7.54E-07	8.33E-07	2.05E-06
MPIFFT_time1	0.24311	0.0808808	0.203111	0.199713	0.209868
MPIFFT_time2	0.295238	0.0836043	0.138058	0.162992	0.347712
MPIFFT_time3	0.132825	0.0447779	0.115376	0.108817	0.101813
MPIFFT_time4	0.779895	0.173629	1.03221	0.541918	1.03963
MPIFFT_time5	0.432601	0.0754626	0.209878	0.194478	0.2239
MPIFFT_time6	2.19E-06	8.74E-07	1.21E-06	3.34E-07	1.30E-06
CPS_HPCC_FFT_235	0	0	0	0	0
CPS_HPCC_FFTW_ESTIMATE	0	0	0	0	0
CPS_HPCC_MEMALLCTR	0	0	0	0	0
CPS_HPL_USE_GETPROCESSTIMES	0	0	0	0	0
CPS_RA_SANDIA_NOPT	0	0	0	0	0
CPS_RA_SANDIA_OPT2	0	0	0	0	0
cps_using_fftw	0	0	0	0	0

Appendix C

Detailed Results from Chapter 7

C.1 Confusion Matrix of Chapter 7 Combined Classification

Table C.1: The resulting confusion matrix entries for each of the 18 classifiers in Chapter 7.

ID	Group	%	TP	FP	TN	FN
0	error	0	304913	37440	476369	52813
1	converged	0	136911	27375	665535	41714
2	overall	0	13	117	870536	869
3	overall	25	1122	742	865070	4601
4	overall	50	3738	1513	858028	8256
5	overall	100	10909	3451	844127	13048
6	np	0	922	1409	863222	5982
7	np	25	7200	3627	847004	13704
8	np	50	13751	5111	835723	16950
9	np	100	22438	7113	819845	22139
10	sys	0	157	464	867456	3458
11	sys	25	3762	1727	856098	9948
12	sys	50	8994	3106	845950	13485
13	sys	100	18136	5332	830213	17854
14	np_and_system	0	8250	3982	846455	12848
15	np_and_system	25	18085	5671	828924	18855
16	np_and_system	50	24002	7052	819047	21434
17	np_and_system	100	33235	9699	801718	26883

Table C.2: Derived metrics for the 18 labels discussed in Chapter 7.

group	%	TPR	TNR	PPV	NPV	FNR	FPR	FDR	FOR	ACC	F1	MCC	AUROC	AUPR
error	0	0.85	0.93	0.89	0.90	0.15	0.07	0.11	0.10	0.90	0.44	0.79	0.95	0.93
converged	0	0.77	0.96	0.83	0.94	0.23	0.04	0.17	0.06	0.92	0.40	0.75	0.95	0.87
overall	0	0.01	1.00	0.10	1.00	0.99	0.00	0.90	0.00	1.00	0.01	0.04	0.55	0.02
overall	25	0.20	1.00	0.60	0.99	0.80	0.00	0.40	0.01	0.99	0.15	0.34	0.85	0.36
overall	50	0.31	1.00	0.71	0.99	0.69	0.00	0.29	0.01	0.99	0.22	0.47	0.91	0.53
overall	100	0.46	1.00	0.76	0.98	0.54	0.00	0.24	0.02	0.98	0.28	0.58	0.94	0.66
np	0	0.13	1.00	0.40	0.99	0.87	0.00	0.60	0.01	0.99	0.10	0.23	0.79	0.21
np	25	0.34	1.00	0.67	0.98	0.66	0.00	0.33	0.02	0.98	0.23	0.47	0.91	0.53
np	50	0.45	0.99	0.73	0.98	0.55	0.01	0.27	0.02	0.97	0.28	0.56	0.93	0.62
np	100	0.50	0.99	0.76	0.97	0.50	0.01	0.24	0.03	0.97	0.30	0.60	0.94	0.68
sys	0	0.04	1.00	0.25	1.00	0.96	0.00	0.75	0.00	1.00	0.04	0.10	0.60	0.06
sys	25	0.27	1.00	0.69	0.99	0.73	0.00	0.31	0.01	0.99	0.20	0.43	0.88	0.47
sys	50	0.40	1.00	0.74	0.98	0.60	0.00	0.26	0.02	0.98	0.26	0.54	0.92	0.61
sys	100	0.50	0.99	0.77	0.98	0.50	0.01	0.23	0.02	0.97	0.31	0.61	0.94	0.69
np_and_system	0	0.39	1.00	0.67	0.99	0.61	0.00	0.33	0.01	0.98	0.25	0.50	0.89	0.51
np_and_system	25	0.49	0.99	0.76	0.98	0.51	0.01	0.24	0.02	0.97	0.30	0.60	0.94	0.68
np_and_system	50	0.53	0.99	0.77	0.97	0.47	0.01	0.23	0.03	0.97	0.31	0.62	0.94	0.70
np_and_system	100	0.55	0.99	0.77	0.97	0.45	0.01	0.23	0.03	0.96	0.32	0.63	0.95	0.72

C.2 Overall Feature Importance

Table C.3: The percentage of times that each matrix and system feature was selected as an important factor in assigning the classification label to the fastest solver-preconditioners regardless of core count or system.

Feature	overall_0	overall_25	overall_50	overall_100
HPL_Tflops	0	0.01	0.02	0
StarDGEMM_Gflops	0	0.015	0	0.12
SingleDGEMM_Gflops	0	0.005	0	0
PTRANS_GBs	0	0.04	0.105	0.135
MPIRandomAccess_LCG_GUPs	0.01	0.02	0	0
MPIRandomAccess_GUPs	0.005	0.02	0	0.005
StarRandomAccess_LCG_GUPs	0	0.025	0.095	0.005
SingleRandomAccess_LCG_GUPs	0.17	0.395	0.01	0
StarRandomAccess_GUPs	0	0.02	0.085	0.005
SingleRandomAccess_GUPs	0.02	0.21	0.005	0
StarSTREAM_Copy	0	0.005	0.02	0.01
StarSTREAM_Scale	0	0	0.025	0.015
StarSTREAM_Add	0	0	0.025	0.04
StarSTREAM_Triad	0	0	0.01	0.01
SingleSTREAM_Copy	0	0	0	0.13
SingleSTREAM_Scale	0.015	0.035	0	0
SingleSTREAM_Add	0.045	0.09	0.015	0
SingleSTREAM_Triad	0.125	0.225	0.01	0
StarFFT_Gflops	0.005	0	0	0
SingleFFT_Gflops	0	0	0	0
MPIFFT_Gflops	0.135	0.06	0.01	0.01
MaxPingPongLatency_usec	0	0	0.13	0.11
RandomlyOrderedRingLatency_usec	0	0	0	0
MinPingPongBandwidth_GBytes	0.17	0.235	0.285	0.305
NaturallyOrderedRingBandwidth_GBytes	0.435	0.57	0.565	0.505
RandomlyOrderedRingBandwidth_GBytes	0.04	0.185	0.505	0.38
MinPingPongLatency_usec	0.15	0.035	0	0.375
AvgPingPongLatency_usec	0.01	0	0.04	0.36
MaxPingPongBandwidth_GBytes	0	0.04	0.04	0.02
AvgPingPongBandwidth_GBytes	0.07	0.125	0.12	0.075
NaturallyOrderedRingLatency_usec	0	0.005	0.025	0.025
MemProc	0.035	0.045	0.075	0.24
core_count	0.27	0.1	0.12	0.15
cpu_freq	0.255	0.195	0	0.06
bogo_mips	0.495	0.43	0	0.07
l1_cache	0	0	0	0
l2_cache	0	0	0	0
l3_cache	0.525	0.215	0.24	0.055
memory_size	0.015	0.025	0.01	0.08
memory_freq	0	0.005	0	0.06
memory_type	0	0.005	0.01	0.07
rows	0.47	0.225	0.185	0

cols	0.245	0.14	0.075	0
min_nnz_row	0	0.99	1	1
row_var	0	0.095	0.025	0.02
col_var	0	0.02	0.02	0.01
diag_var	0	0	0	0
nnz	0	0	0.08	0.98
frob_norm	0	0.005	0	0
symm_frob_norm	0	0.01	0.01	0.025
antisymm_frob_norm	0	0	0	0.165
one_norm	0	0.255	0.2	0.16
inf_norm	0	0.49	0.405	0.465
symm_inf_norm	0	0.13	0.29	0.255
antisymm_inf_norm	0	0	0	0.465
max_nnz_row	0.005	0.66	0.545	0.015
trace	0	0	0	0.01
abs_trace	0	0	0	0
avg_nnz_row	0	0.11	0.005	0.005
dummy_rows	0	0	0	0.305
dummy_rows_kind	0	1	1	1
num_value_symm_1	0	0.51	0.625	0.675
nnz_pattern_symm_1	0	0.255	0.275	0.235
num_value_symm_2	0	0	0.575	0.71
nnz_pattern_symm_2	0	0	0.31	0.345
row_diag_dom	0	0.21	0.22	0
col_diag_dom	0	0.99	1	1
diag_avg	0	0.1	0.1	0.115
diag_sign	0	0.47	0.47	0.715
diag_nnz	0.24	0.635	0.475	0.36
lower_bw	0.07	0.245	0.355	0.985
upper_bw	0.025	0	0.305	0.995
row_log_val_spread	0	0.73	0.675	0.25
col_log_val_spread	0	0.3	0.545	0.76
symm	0	0.11	0.1	0.09
np	0.55	1	1	1
solver_id	1	1	1	1
prec_id	0.49	1	1	0.5

C.3 NP Feature Importance

Table C.4: The percentage of times that each matrix and system feature was selected as an important factor in assigning the classification label to the fastest solver-preconditioners at a given number of cores, regardless of system.

Feature	np_0	np_25	np_50	np_100
HPL_Tflops	0	0	0	0
StarDGEMM_Gflops	0	0.005	0.01	0.02
SingleDGEMM_Gflops	0	0	0	0.005
PTRANS_GBs	0	0.08	0.025	0.06
MPIRandomAccess_LCG_GUPs	0	0	0	0
MPIRandomAccess_GUPs	0	0	0	0
StarRandomAccess_LCG_GUPs	0	0	0	0
SingleRandomAccess_LCG_GUPs	0.06	0	0.005	0.015
StarRandomAccess_GUPs	0	0	0	0
SingleRandomAccess_GUPs	0.01	0.005	0.01	0.01
StarSTREAM_Copy	0	0.04	0.015	0.025
StarSTREAM_Scale	0	0.03	0.005	0
StarSTREAM_Add	0	0.01	0.03	0.015
StarSTREAM_Triad	0	0.005	0	0
SingleSTREAM_Copy	0	0.065	0.07	0.19
SingleSTREAM_Scale	0.035	0	0	0
SingleSTREAM_Add	0.145	0.005	0	0
SingleSTREAM_Triad	0.28	0	0	0
StarFFT_Gflops	0.035	0	0	0
SingleFFT_Gflops	0	0	0	0
MPIFFT_Gflops	0.355	0.07	0	0.025
MaxPingPongLatency_usec	0	0.055	0.14	0.03
RandomlyOrderedRingLatency_usec	0	0	0.005	0
MinPingPongBandwidth_GBytes	0.075	0.21	0.155	0.18
NaturallyOrderedRingBandwidth_GBytes	0.505	0.525	0.515	0.515
RandomlyOrderedRingBandwidth_GBytes	0.01	0.215	0.45	0.23
MinPingPongLatency_usec	0.135	0	0.18	0.28
AvgPingPongLatency_usec	0.005	0.005	0.35	0.105
MaxPingPongBandwidth_GBytes	0	0.01	0	0
AvgPingPongBandwidth_GBytes	0.045	0.12	0.035	0.06
NaturallyOrderedRingLatency_usec	0	0.005	0.04	0.005
MemProc	0.035	0.325	0.215	0.445
core_count	0.155	0.37	0.24	0.215
cpu_freq	0.275	0	0.025	0.01
bogo_mips	0.495	0	0.05	0.015
l1_cache	0	0	0	0
l2_cache	0	0	0	0
l3_cache	0.295	0.4	0.05	0.075
memory_size	0	0.25	0.14	0.205
memory_freq	0	0.115	0.375	0.355
memory_type	0	0.055	0.285	0.295
rows	0.52	0.12	0	0

cols	0.245	0.09	0	0
min_nnz_row	0	0.46	0.505	0.995
row_var	0	0.04	0.04	0.05
col_var	0	0.02	0.02	0.02
diag_var	0	0.015	0.005	0
nnz	0.005	0.005	0.06	0.485
frob_norm	0	0.005	0.01	0
symm_frob_norm	0	0.005	0	0.005
antisymm_frob_norm	0	0.015	0	0.04
one_norm	0	0.125	0.18	0.18
inf_norm	0	0.34	0.32	0.28
symm_inf_norm	0	0.465	0.44	0.45
antisymm_inf_norm	0	0.055	0.03	0.515
max_nnz_row	0	0.05	0	0.33
trace	0	0.005	0	0
abs_trace	0	0.005	0	0
avg_nnz_row	0	0.005	0	0
dummy_rows	0.005	0.445	0	0.025
dummy_rows_kind	0	1	1	1
num_value_symm_1	0	0.625	0.635	0.62
nnz_pattern_symm_1	0	0.235	0.23	0.22
num_value_symm_2	0	0.71	0.765	0.76
nnz_pattern_symm_2	0	0.27	0.275	0.315
row_diag_dom	0	0.275	0.24	0.715
col_diag_dom	0	0.815	1	1
diag_avg	0	0.095	0.08	0.1
diag_sign	0	0.4	0	0.08
diag_nnz	0.145	0.575	0	0.275
lower_bw	0.075	0.24	0.395	0.94
upper_bw	0.015	0.005	0	0.7
row_log_val_spread	0	0.29	0.225	0.26
col_log_val_spread	0	0.75	0.8	0.745
symm	0	0.125	0.13	0.155
np	0.535	0.02	0.82	1
solver_id	1	1	1	1
prec_id	1	1	0.79	0

C.4 Sys Feature Importance

Table C.5: The percentage of times that each matrix and system feature was selected as an important factor in assigning the classification label to the fastest solver-preconditioners for each system regardless of np.

Feature	np_0	np_25	np_50	np_100
HPL_Tflops	0	0	0	0
StarDGEMM_Gflops	0	0.005	0.01	0.02
SingleDGEMM_Gflops	0	0	0	0.005
PTRANS_GBs	0	0.08	0.025	0.06
MPIRandomAccess_LCG_GUPs	0	0	0	0
MPIRandomAccess_GUPs	0	0	0	0
StarRandomAccess_LCG_GUPs	0	0	0	0
SingleRandomAccess_LCG_GUPs	0.06	0	0.005	0.015
StarRandomAccess_GUPs	0	0	0	0
SingleRandomAccess_GUPs	0.01	0.005	0.01	0.01
StarSTREAM_Copy	0	0.04	0.015	0.025
StarSTREAM_Scale	0	0.03	0.005	0
StarSTREAM_Add	0	0.01	0.03	0.015
StarSTREAM_Triad	0	0.005	0	0
SingleSTREAM_Copy	0	0.065	0.07	0.19
SingleSTREAM_Scale	0.035	0	0	0
SingleSTREAM_Add	0.145	0.005	0	0
SingleSTREAM_Triad	0.28	0	0	0
StarFFT_Gflops	0.035	0	0	0
SingleFFT_Gflops	0	0	0	0
MPIFFT_Gflops	0.355	0.07	0	0.025
MaxPingPongLatency_usec	0	0.055	0.14	0.03
RandomlyOrderedRingLatency_usec	0	0	0.005	0
MinPingPongBandwidth_GBytes	0.075	0.21	0.155	0.18
NaturallyOrderedRingBandwidth_GBytes	0.505	0.525	0.515	0.515
RandomlyOrderedRingBandwidth_GBytes	0.01	0.215	0.45	0.23
MinPingPongLatency_usec	0.135	0	0.18	0.28
AvgPingPongLatency_usec	0.005	0.005	0.35	0.105
MaxPingPongBandwidth_GBytes	0	0.01	0	0
AvgPingPongBandwidth_GBytes	0.045	0.12	0.035	0.06
NaturallyOrderedRingLatency_usec	0	0.005	0.04	0.005
MemProc	0.035	0.325	0.215	0.445
core_count	0.155	0.37	0.24	0.215
cpu_freq	0.275	0	0.025	0.01
bogo_mips	0.495	0	0.05	0.015
l1_cache	0	0	0	0
l2_cache	0	0	0	0
l3_cache	0.295	0.4	0.05	0.075
memory_size	0	0.25	0.14	0.205
memory_freq	0	0.115	0.375	0.355
memory_type	0	0.055	0.285	0.295
rows	0.52	0.12	0	0

cols	0.245	0.09	0	0
min_nnz_row	0	0.46	0.505	0.995
row_var	0	0.04	0.04	0.05
col_var	0	0.02	0.02	0.02
diag_var	0	0.015	0.005	0
nnz	0.005	0.005	0.06	0.485
frob_norm	0	0.005	0.01	0
symm_frob_norm	0	0.005	0	0.005
antisymm_frob_norm	0	0.015	0	0.04
one_norm	0	0.125	0.18	0.18
inf_norm	0	0.34	0.32	0.28
symm_inf_norm	0	0.465	0.44	0.45
antisymm_inf_norm	0	0.055	0.03	0.515
max_nnz_row	0	0.05	0	0.33
trace	0	0.005	0	0
abs_trace	0	0.005	0	0
avg_nnz_row	0	0.005	0	0
dummy_rows	0.005	0.445	0	0.025
dummy_rows_kind	0	1	1	1
num_value_symm_1	0	0.625	0.635	0.62
nnz_pattern_symm_1	0	0.235	0.23	0.22
num_value_symm_2	0	0.71	0.765	0.76
nnz_pattern_symm_2	0	0.27	0.275	0.315
row_diag_dom	0	0.275	0.24	0.715
col_diag_dom	0	0.815	1	1
diag_avg	0	0.095	0.08	0.1
diag_sign	0	0.4	0	0.08
diag_nnz	0.145	0.575	0	0.275
lower_bw	0.075	0.24	0.395	0.94
upper_bw	0.015	0.005	0	0.7
row_log_val_spread	0	0.29	0.225	0.26
col_log_val_spread	0	0.75	0.8	0.745
symm	0	0.125	0.13	0.155
np	0.535	0.02	0.82	1
solver_id	1	1	1	1
prec_id	1	1	0.79	0

C.5 NP and System Feature Importance

Table C.6: The percentage of times that each matrix and system feature was selected as an important factor in assigning the classification label to the fastest solver-preconditioners specific to each np and system.

Feature	np_and_sys_0	np_and_sys_25	np_and_sys_50	np_and_sys_100
HPL_Tflops	0	0.005	0	0
StarDGEMM_Gflops	0.035	0.035	0.025	0.09
SingleDGEMM_Gflops	0	0	0	0
PTRANS_GBs	0.06	0.1	0.06	0.065
MPIRandomAccess_LCG_GUPs	0	0	0	0
MPIRandomAccess_GUPs	0	0	0	0
StarRandomAccess_LCG_GUPs	0	0	0	0
SingleRandomAccess_LCG_GUPs	0.005	0.025	0.02	0.015
StarRandomAccess_GUPs	0	0	0	0
SingleRandomAccess_GUPs	0.02	0.03	0.01	0.005
StarSTREAM_Copy	0.02	0.05	0.015	0.02
StarSTREAM_Scale	0.01	0.065	0	0.005
StarSTREAM_Add	0.01	0.055	0.03	0.025
StarSTREAM_Triad	0.015	0.085	0	0.025
SingleSTREAM_Copy	0.19	0.085	0.115	0.29
SingleSTREAM_Scale	0	0	0	0
SingleSTREAM_Add	0	0	0	0
SingleSTREAM_Triad	0.005	0	0	0
StarFFT_Gflops	0	0.005	0	0
SingleFFT_Gflops	0	0.005	0	0
MPIFFT_Gflops	0.025	0.015	0.02	0.015
MaxPingPongLatency_usec	0.04	0.235	0.135	0.085
RandomlyOrderedRingLatency_usec	0	0	0.005	0
MinPingPongBandwidth_GBytes	0.14	0.11	0.21	0.18
NaturallyOrderedRingBandwidth_GBytes	0.5	0.5	0.425	0.48
RandomlyOrderedRingBandwidth_GBytes	0.33	0.39	0.355	0.245
MinPingPongLatency_usec	0.26	0.265	0.33	0.38
AvgPingPongLatency_usec	0.22	0.365	0.445	0.385
MaxPingPongBandwidth_GBytes	0.01	0.01	0.01	0
AvgPingPongBandwidth_GBytes	0.085	0.045	0.04	0.045
NaturallyOrderedRingLatency_usec	0.02	0.075	0.04	0.005
MemProc	0.305	0.175	0.25	0.36
core_count	0.295	0.215	0.21	0.105
cpu_freq	0.04	0.03	0.015	0.05
bogo_mips	0.045	0.095	0.085	0.095
l1_cache	0	0	0	0
l2_cache	0	0	0	0
l3_cache	0.13	0.14	0.085	0.075
memory_size	0.185	0.125	0.135	0.145
memory_freq	0.25	0.275	0.285	0.36
memory_type	0.28	0.31	0.33	0.27
rows	0.005	0.015	0	0.05
cols	0	0.015	0	0.015

min_nnz_row	0.77	0.975	0.985	1
row_var	0.005	0.165	0.05	0.085
col_var	0.035	0.205	0.015	0.24
diag_var	0.07	0.145	0	0.245
nnz	0.025	0.34	0	0.285
frob_norm	0	0.09	0.11	0.09
symm_frob_norm	0.01	0.12	0.24	0.065
antisymm_frob_norm	0.075	0	0	0.005
one_norm	0.005	0.175	0.125	0.085
inf_norm	0	0.195	0.175	0.115
symm_inf_norm	0	0.255	0.285	0.265
antisymm_inf_norm	0.325	0.615	0.455	1
max_nnz_row	0.08	0.915	0.47	0.99
trace	0.005	0.05	0.04	0.015
abs_trace	0.005	0.005	0.04	0.025
avg_nnz_row	0.68	0.325	0	0.03
dummy_rows	0.545	0.96	0.37	0.305
dummy_rows_kind	1	1	1	1
num_value_symm_1	0.165	0.58	0.6	0.59
nnz_pattern_symm_1	0.085	0.26	0.265	0.285
num_value_symm_2	0.455	0.74	0.76	0.73
nnz_pattern_symm_2	0.225	0.395	0.305	0.33
row_diag_dom	0.265	0.77	0.805	1
col_diag_dom	0.61	0.975	0.99	1
diag_avg	0.005	0.195	0.385	0.205
diag_sign	0.995	0.22	0	0.89
diag_nnz	0	0.915	0.675	0.985
lower_bw	0.625	0.615	0.1	0.965
upper_bw	0.22	0.02	0.005	0.995
row_log_val_spread	0.495	0.28	0.25	0.315
col_log_val_spread	0.54	0.76	0.77	0.715
symm	0.045	0.16	0.135	0.125
np	1	0.985	0.985	1
solver_id	1	1	1	1
prec_id	1	1	0.445	0.49