MODEL-BASED ALGORITHM SELECTION TECHNIQUES

by

SUDHARSHAN SRINIVASAN

A THESIS

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

June 2019

THESIS APPROVAL PAGE

Student: Sudharshan Srinivasan

Title: Model-Based Algorithm Selection Techniques

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

Boyana Norris	Chair
Thien Huu Nguyen	Core Member

and

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

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

Degree awarded June 2019

© 2019 Sudharshan Srinivasan

THESIS ABSTRACT

Sudharshan Srinivasan

Master of Science

Department of Computer and Information Science

June 2019

Title: Model-Based Algorithm Selection Techniques

With significant research going into the development of scientific software over the years, there exist a plethora of toolkits using different algorithms to solve the same problem. But the performance of these toolkits are very much problem specific and depend on multiple factors, including experimental setup and hardware configurations. This makes it very difficult to choose a suitable software beforehand without testing them for specific problems while the wrong choice of software contributes to severe performance downgrade. In this thesis, we address this selection challenge by proposing a faster and reliable model-based approach instead of empirically running time-consuming experiments to select suitable software toolkits. In specific, we would be looking at selection for two classes of algorithms that solve parallel graph processing applications and systems of linear equations. Appropriate metrics have also been introduced to evaluate the quality of selection techniques

CURRICULUM VITAE

NAME OF AUTHOR: Sudharshan Srinivasan

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene
SRM Institute of Science and Technology, Chennai

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2019, University of Oregon
Bachelor of Technology, Computer Science, 2017, SRM Institute of Science and Technology & Science

AREAS OF SPECIAL INTEREST:

Machine Learning
Parallel Computing

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, University of Oregon, 2018 – 2019

Graduate Teaching Fellow, University of Oregon, 2017 – 2018

Summer Intern, Tata Consultencey Services(TCS), Chennai, Summer 2015

PUBLICATIONS:

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

Pollard, S. & **Srinivasan, S.** & Norris, B. A performance and recommendation system for parallel graph processing implementations *Work-inprogress. In Companion of the 10th ACM/SPEC International Conference on Performance Engineering, ICPE âĂŹ19.*

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. RELATED WORK 3

III. BACKGROUND AND TOOLS 5

3.1. Graph processing systems 5

3.1.1. The Graph 500 5

3.1.2. The Graph Algorithm Platform (GAP) and GraphBIG benchmark suite 5

3.1.3. GraphMat 6

3.1.4. Galois 6

3.1.5. PowerGraph 6

3.2. Algorithms 6

3.2.1. Breadth-first search (BFS) 6

3.2.2. Single-source shortest path (SSSP) 6

3.2.3. Page rank 7

3.3. Machine learning models and validations 7

3.3.1. Linear regression 7

3.3.2. Logistic regression 7

3.3.3. Random Forest 7

3.4. Miscellaneous 8

3.4.1. Portable, Extensible Toolkit for Scientific Computation (PETSc 8

Chapter Page

3.4.2. Multi-physics object oriented simulation environment (MOOSE) 8

3.4.3. SuiteSparse Matrix Collection 8

3.4.4. Beautiful soup 8

IV. METHODOLOGY 10

4.1. Selection of parallel graph processing packages 10

4.1.1. Execution time prediction of graph processing experiments 11

4.1.1.1. Learning set 12

4.1.1.2. Feature selection 13

4.1.1.3. Prepossessing 17

4.1.1.4. Fitting the model 19

4.1.1.5. Experiments and Validation 21

4.1.2. Classification of graph processing experiments 21

4.1.2.1. Preprocessing 21

4.1.2.2. Classification model 22

4.1.2.3. Experiment and validation 22

4.2. Selection of solvers for a system of linear equations 23

4.2.1. Ranking framework 23

4.2.1.1. Learning set 23

4.2.1.2. Two stage ranking algorithm 23

4.2.1.3. Experiment and validation 26

V. RESULTS AND ANALYSIS 28

5.1. Parallel graph processing experiments 28

Chapter Page

5.1.1. Execution time prediction 28

5.1.2. Classification of experiments 30

5.2. Experiments on solving linear systems 32

5.2.1. Predicting execution times for experiments 33

5.2.2. Ranking experiments for linear systems 33

VI. CONCLUSION AND FUTURE WORK 38

REFERENCES CITED 40

LIST OF FIGURES

Figure Page

1. Multiple levels of abstractions in the learning set 12
2. First 20 rows of learning set 13
3. Average clustering coefficient 14
4. Connected components 14
5. Example of 1-hot encoding 19
6. Overview of the ranking algorithm 24
7. Validation setup for testing 26
8. Linear regression with and without ridging and normalization 29
9. Confusion matrices for all algorithms 30
10. TEPS for each algorithm 32
11. spread of speedups across testing sets 34
12. Top 10 ranked experiments 36

LIST OF TABLES

Table Page

1. Hyperparamets and its final value 26
2. Statistics of speedups across all testing sets 35
3. Statistics of speedups across all testing sets(MOOSE) 36

CHAPTER I

INTRODUCTION

The research behind developing software toolkits for scientific problems has significantly improved along the years with a variety of tools available for efficiently solving high volumes of mathematical computations. Several different optimizations in algorithms have spawned a vast array of toolkits with each offering a unique advantage to specific problem characteristics, but a silver bullet that efficiently performs well across all unique problems and experimental setups remain elusive. This brings about a new selection concern where the wrong choice of an algorithm could lead to massive performance downgrades that adds up with multiple iterations of the experiment performed across an application.

This is compounded by the fact that the selection is usually made by researchers who are specialists in that application but have limited knowledge when it comes to identifying the minor intricacies in experimental setups that could lead to significant performance improvements.

In this thesis, we provide a model-based approach towards solving this selection concern for two classes of algorithms, namely, selection of parallel graph processing packages and solvers for linear systems. A web-based framework using the proposed approaches was also developed for these applications. It is essential for researchers to choose reliable algorithms and toolkits, and we aim to provide this solution. Reliability refers to not only having consistent performance improvements but also never to have experiments that fail, which in reality is a common concern.

One way of ensuring reliability is to choose algorithms from the empirical analysis of exhaustively running all combinations of experiment setups. But this

scales up pretty quickly making it unfeasible to run experiments just for selection. For example, there are over 120 solvers available for linear systems, and each can be used on many different experimental setups. It would take weeks or months to figure out which experimental setup and algorithm work best for all their linear systems in an application. A model-based approach circumvents all this by using machine learning techniques to predict the quality of solvers for specific systems rather than explicitly running those experiments to determine quality.

The modeling is done on the metadata of the problem set, which tries to capture the relation between the problem and the criterion variable. The criterion variable for both the classes of algorithms is execution time while the metadata acts as features of the learning models. The fundamental motivation behind our approach towards selection is the fact that given a problem set, the time taken to perform feature extraction and modeling for criterion variable is still less than actually running that experiment.

Chapters two and three deal with background information and related works, while chapter four presents our framework and its workings. In specific, we provide both a predictive and classification modeling based approach towards algorithm selection for the two mentioned classes of algorithm. For the latter class, we have also provided a ranking framework that gives ranked list of solvers for specific linear systems. Chapter four discusses and analyses the results we obtained for all the experiments performed, followed by the conclusion chapter, which gives us a holistic view of the framework and the final models selected for use. It also offers details of the current and future work conducted by us along the same lines and identifies where improvements could be made to the existing methodologies.

CHAPTER II

RELATED WORK

Improving quality and performance of software toolkits is a really mature field with a lot of active research going towards development of new algorithms and techniques for efficiently solving various problems. But a lot of these solutions are still application dependent and works with different degrees of performance for each new application. The end performance is also a factor of the experimental setup and the hardware configurations which are not extensively taken into account when developing these toolkits. The combination of these facts have resulted in a plethora of software toolkits available for most problems. But with that said, not a lot of research has gone into selecting the best toolkits for specific problems with the characteristics of application and experimental setup in mind.

In specific, the two classes of algorithms we would be looking at, namely, algorithms for parallel graph processing and solving linear systems are further unexplored when it comes to choosing the right software for varying experimental setups. An extensive survey on large scale graph processing packages is done by [2], analysing the performance metrics of various graph packages. Surveys of iterative solvers and preconditioners are relatively more explored with [4] giving us performance evaluations for solvers focusing on large linear systems. Likewise, [5], [7], [6] also gives us a comprehensive survey of solvers and preconditioners.

But it is to be noted that all of the mentioned works are surveys on existing solutions that runs empirical experiments to analyze performance. They provide inferences on best performing solvers based on empirical evidence but none of them suggest solvers for specific problems, let alone experimental setups. Papers [24] and

[16] takes one step closer by providing a portfolio based approach for selecting SAT solvers. But these are once again done through empirical evidences.

The closest work to us that uses machine learning based models for software solver selections are [14] and [17]. The former uses K-nearest neighbor based approach while the the latter is the closest to our research which uses a similar feature selection and classification based approach for selection. But none of them to the best of our knowledge, perform predictive modelling. In this thesis, we look at selection of software toolkits with a machine learning model based approaches that uses both predictive modeling and classification for fast and accurate selection.

CHAPTER III

BACKGROUND AND TOOLS

This thesis is at the intersection of HPC and data science, using a variety of tools and other supporting material that readers need to be familiar with before using our proposed framework. We can broadly classify these tools as:

3.1 Graph processing systems

For our first class of algorithms, we performed numerous experiments on graph datasets using six parallel graph processing packages [18]. The first five packages were implemented using shared memory parallelism while we included one package that was implemented using distributed memory parallelism.

3.1.1 The Graph 500.

This package provides reference OpenMP implementations for benchmarking graph algorithms using shared memory parallelism. They also provide distributed memory and cloud/MapReduce implementations but we focus only on shared memory for this package. It accepts graphs in compressed sparse row (CSR) formats.

CSR is used to represent sparse matrices by using three vectors for storing the non-zero elements along with their count and indices, thereby removing the redundancy of storing zero elements.

3.1.2 The Graph Algorithm Platform (GAP) and GraphBIG benchmark suite.

Both are graph benchmarking suites developed that aims at providing standardized graph processing evaluations and optimized reference implementations for shared memory parallelism [3] [?]. They also use a CSR representation to store input graphs.

3.1.3 GraphMat.

GraphMat is a library that provides OpenMP reference implementations for graph algorithms similar to GAP but it uses a doubly compressed sparse row representation for input graphs [23].

3.1.4 Galois.

Galois is a multilevel programming model that also uses shared memory parallelism for it's reference implementation. It provides its own execution model for implementing algorithms on parallel setups.

3.1.5 PowerGraph.

Unlike the other five chosen packages, PowerGraph uses distributed memory parallelism for reference implementations of the mention algorithms [9]. It also provides shared memory implementations but we focus only on the former variant in this framework. For storage, it uses its own novel schema on top of CSR.

3.2 Algorithms

For the graph processing applications, we look at four different graph theoretic algorithms that the framework incorporates while for linear systems, we consider around 70 different solvers, which are in turn 70 different algorithms out of which, we will discuss the two baseline solvers used across our thesis. The four graph theoretic algorithms are briefly described in the remainder of this section.

3.2.1 Breadth-first search (BFS).

It is a search algorithm that iteratively traverses a tree or graph data structure for any specific query node[25]. Starting at the root vertex, all the neighbouring vertices within the same depth level are explored before going deeper into a specific branch unlike depth-first search which does the opposite. It has a computational complexity of $O(V + E)$ where V and E denote the number of vertices and edges respectively.

3.2.2 Single-source shortest path (SSSP).

It is a graph theoretic algorithm that identifies a path between any two vertices such that the sum of

weight of edges between the two vertices are minimized [13]. It has a computational complexity of $O(V + E)logV$.

3.2.3 Page rank. It is another graph theoretic algorithm that identifies and ranks nodes based on their impotence which is defined by the degree of the node [20]. The computation complexity of page rank is $O(V + E)$.

3.3 Machine learning models and validations

In this thesis we concentrate on a machine learning based approach towards selection so we will look at multiple learning models. In specific, we will discuss two classification models and one predictive model.

3.3.1 Linear regression. It is a modeling approach for capturing the relationship between dependent and independent features linearly [22]. By capturing this relationship, a model can now predict the independent feature given the dependent features. We implement it using SKlearn's python based library [21].

3.3.2 Logistic regression. Similar to linear regression, this model also finds the linear relation between dependent and independent features but it fits as a classification problem rather than predictive modeling [11]. This is also implemented using SKlearn [21].

3.3.3 Random Forest. Is a supervised learning algorithm that can be used for both predictive modeling and classification. It models its application by creating an ensemble of decision trees or namely, a 'forest' which are aggregated to give a more stable prediction than normal decision trees [15]. The general idea behind this is the fact that a combination of learning models improve accuracy as different splits in decisions can cause massive difference in predictions and one would ideally want the mean across all possible trees generated.

3.4 Miscellaneous

Some general tools that does not belong to any of the previous classifications were also used across this thesis.

3.4.1 Portable, Extensible Toolkit for Scientific Computation (PETSc. PETSc is a framework that provides data structures and routines for solving partial differential equations used by scientific applications [1]. The framework contains scalable routines that can run in a parallel environment using MPI standard for all inter-process communications. The routines we use for solving the linear systems are from PETSc.

3.4.2 Multi-physics object oriented simulation environment (MOOSE). It is a multi-physics framework that provides high level interface and real-time applications for mathematical models given by the PETSc library [12]. We use the various linear systems offered by MOOSE as one of learning set for the class of algorithms that solves system of linear equations. We use the MOOSE learning set for serially execution experiments.

3.4.3 SuiteSparse Matrix Collection. It is a growing set of sparse matrices from real-life applications that is used to evaluate and benchmark matrix algorithms by the linear algebra community [8]. It consists of applications ranging from structural engineering, computational fluid dynamics, model reduction, electromagnetics, semiconductor devices, thermodynamics, materials, acoustics, computer graphics/vision, robotics/kinematics, and other discretizations. We use the SuiteSparse collection for creating the training set of parallel experiments.

3.4.4 Beautiful soup. The BeautifulSoup library is used for web scraping in python. We use beautiful soup to extract the metadata of real world graph datasets from SNAP and KONECT. A regular HTTP request is made to the

server which returns HTML page as the response. The response is then parsed to BeautifulSoup format so we can use BeautifulSoup to work on it. Using functions on the BeautifulSoup object like “find all” and “get” can help in getting the desired object.

CHAPTER IV

METHODOLOGY

In the previous chapters, we gave a short synopsis of what the framework tries to achieve and why we need it. We also briefly discussed the supporting concepts required for understanding the framework. In this chapter, we describe the in-depth concepts behind the framework along with explaining why we implemented a particular technique and all the other alternative techniques that we tried or contemplated trying. Although this chapter covers details about all implementations and techniques we tried, information on why some of the methods failed or problems with implementations are discussed in the results and analysis section.

We have previously mentioned that our framework provides selection techniques for two broad classes of problems, namely, parallel graph processing and system of linear equations. To maintain the specified structure, this chapter will be sub-categorized into each of those problems.

4.1 Selection of parallel graph processing packages

It has already been established that with a plethora of available parallel graph processing packages, selection based on hard empirical evidence of various experiments are time and resource consuming to obtain and as a result, we are introducing machine learning based package selection.

In terms of what we can achieve with a machine learning based framework for package selection, we could either perform execution time prediction or a binary classification of packages for a particular graph and hardware features. Hence, this section is further categorized into two subsections, with each detailing one of the aspects.

4.1.1 Execution time prediction of graph processing experiments.

The objective here is to efficiently and accurately predict the execution time of a particular experiment without explicitly running those experiments. An experiment in this context is one specific package running one specific algorithm over one specific graph dataset across given hardware configurations.

Machine learning-based projects differ from each other in various ways, but the two universally common requirements is a dataset and a model. To distinguish between a machine learning dataset and graph dataset, throughout this thesis, we would be referring to the former as a learning set and the actual graph dataset as a dataset.

The learning set in this case is a collection of individual runs of experiments which consists of three components, i.e.) dataset, algorithm, and package along with its features and ground truth. The dataset is any graph dataset in edge-view format. Datasets come in two flavors. One is a synthetic dataset which we auto-generated. We use the Graph500 synthetic graph generator which creates a Kronecker graph with initial parameters of A = 0.57, B = 0.19, C = 0.19, and $D = 1(A + B + C) = 0.05$ and set the average degree of a vertex as 16. Hence, a Kronecker graph with scale S has 2^S vertices and approximately $16 * 2^S$ edges. Kronecker graphs are a generalization of RMAT graphs. The RMAT takes the scale as a parameter. Most of our experiments have a scale of 22, which is 4,194,304 vertices and an average of 16 edges per vertex.

Another flavor of the dataset is real-world dataset. These are the graph datasets that that are modeled from real-world applications like connectivity map of Facebook and Google. These real-world datasets are obtained from scraping

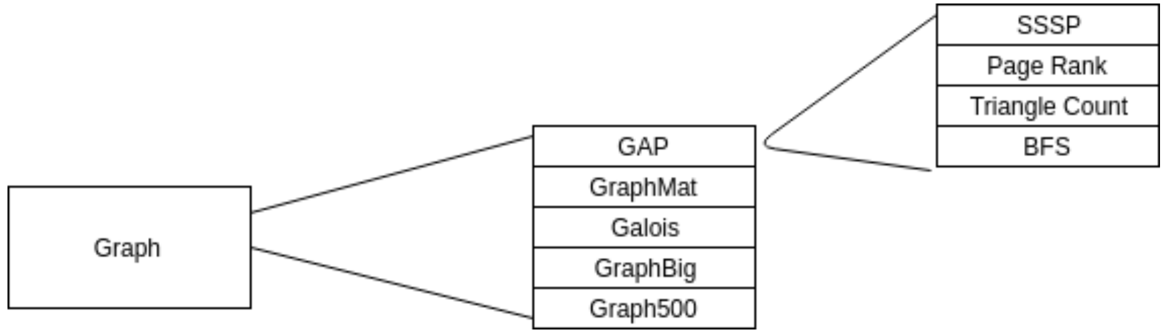


Figure 1. Multiple levels of abstractions in the learning set

the web sources using Beautiful soup. The web sources include SNAP and Konect databases. Overall, we have managed to get 450 different graph datasets that are a combination of real-world and synthetic graphs.

The next component is the algorithm. In our framework, a list of four algorithms which includes breadth-first search (BFS), triangle count (TC), single source shortest path (SSSP), and page rank (PR) is implemented on each dataset. More details on these algorithms are explained in the background section. Each of these algorithms is performed by six different parallel graph processing packages that include GAP, GraphMat, Graph500, GraphBig, Galois, and Powergraph, which are again explained in detail in the background section.

4.1.1.1 Learning set. Figure 1 shows us the multilevel abstraction in our learning set, where each graph dataset has four possible algorithms, and six different graph processing packages can implement each algorithm. As a result, 24 different experiments need to be conducted for each graph dataset bringing our total experiment count to 4000, which is the size of our learning set. Each experiment in the learning set is accompanied by the features of the graph and its ground truth value, which in this case is the execution time of that particular experiment.

dataset	package	algorithm	nvertices	nedges	nthreads	Nodes.in	Edges.i	Nodes.in	Edges.in.la	Average.clust	Number.of.	Fraction.of	Diameter..long	X90.percentile	runtime
parsed-e	Galois	SSSP	36692	183831	2	0.918	0.984	0.918	0.984	0.497	727044	0.03015	11	4.8	0.01046875
parsed-e	GAP	SSSP	36692	183831	2	0.918	0.984	0.918	0.984	0.497	727044	0.03015	11	4.8	0.00613375
parsed-e	GraphBIG	SSSP	36692	183831	2	0.918	0.984	0.918	0.984	0.497	727044	0.03015	11	4.8	3.21E-05
parsed-e	GraphMat	SSSP	36692	183831	2	0.918	0.984	0.918	0.984	0.497	727044	0.03015	11	4.8	0.002122281
parsed-e	PowerGraph	SSSP	36692	183831	2	0.918	0.984	0.918	0.984	0.497	727044	0.03015	11	4.8	0.00625
parsed-f	Galois	SSSP	4039	88234	1	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.0081875
parsed-f	GAP	SSSP	4039	88234	1	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.0004775
parsed-f	GraphBIG	SSSP	4039	88234	1	1	1	1	1	0.6055	1612010	0.2647	8	4.7	4.69E-07
parsed-f	GraphMat	SSSP	4039	88234	1	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.000560906
parsed-f	PowerGraph	SSSP	4039	88234	1	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.09375
parsed-f	GraphBIG	SSSP	4039	88234	2	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.004153514
parsed-f	GAP	SSSP	4039	88234	4	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.002128125
parsed-f	GraphBIG	SSSP	4039	88234	4	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.004151681
parsed-f	GraphMat	SSSP	4039	88234	4	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.000495188
parsed-f	PowerGraph	SSSP	4039	88234	4	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.0375
parsed-f	GAP	SSSP	4039	88234	8	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.003196875
parsed-f	GraphBIG	SSSP	4039	88234	8	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.00429238
parsed-f	GraphMat	SSSP	4039	88234	8	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.000551
parsed-f	PowerGraph	SSSP	4039	88234	8	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.0625
parsed-f	GAP	SSSP	4039	88234	16	1	1	1	1	0.6055	1612010	0.2647	8	4.7	0.005983125

Figure 2. First 20 rows of learning set

Since different algorithms have different execution times, it won't really be fair if we trained our model across all algorithms. For this purpose we also filter the datapoints with respect to each algorithm. The learning set with the first 10 entries can be seen in figure 2

4.1.1.2 Feature selection. The metadata of each experiment acts as the features upon which a prediction model is implemented. The features are a mix of both metadata of the graph by itself and hardware configurations. Based on regression analysis, the 12 features that had coefficients above 0.2 were finally chosen in the learning set. Features with high coefficients signify high correlation with execution time and are more likely that they would contribute more towards an accurate prediction by the model. The first three features with the highest coefficients are physical properties of the graph and hardware:

1. Number of vertices
2. Number of edges
3. Number of Threads

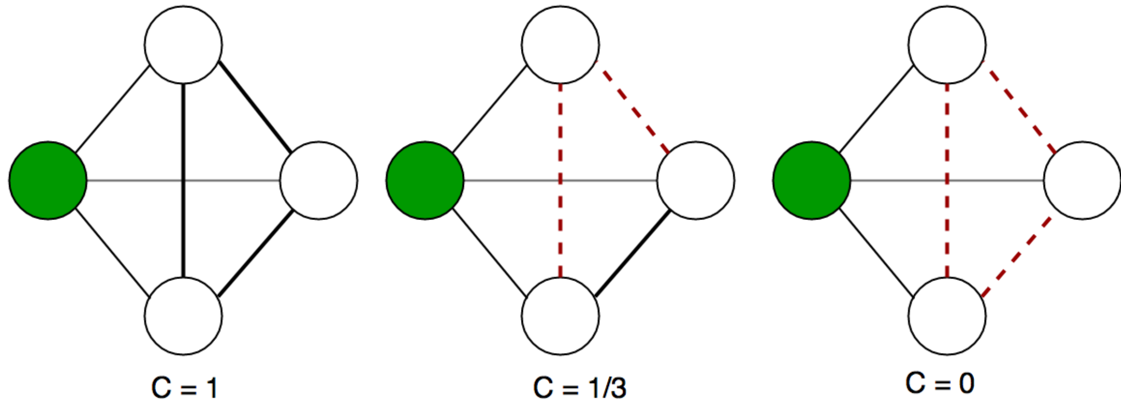


Figure 3. Average clustering coefficient

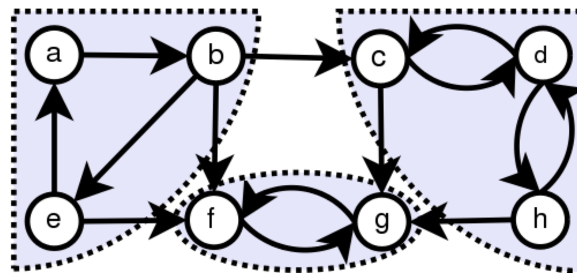


Figure 4. Connected components

The number of vertices has a correlation coefficient of 0.86 with respect to execution time. This is expected since execution time is largely dependent on the size of the graph.

Similarly, the number of edges is correlated with execution time with a coefficient of 0.81. The number of threads follows behind with a coefficient of 0.77 as it determines the communication complexity and time lost in transferring data. The next set of features relates to the density and connectivity of the graph:

Average clustering coefficient

The clustering coefficient is a measure of how well the neighbors of a node are connected. 3 depicts an example calculation of the clustering coefficient. The local clustering coefficient of the green node is computed as the proportion of

connections among its, which are realized compared with the number of all possible connections. In the figure, the green node has three neighbors, which can have a maximum of 3 connections among them. In the left part of the figure, all three possible connections are realized (thick black segments), giving a local clustering coefficient of 1. In the middle part of the figure, only one connection is realized (solid black line), and two connections are missing (dotted red lines), giving a local cluster coefficient of 1/3. Finally, none of the possible connections among the neighbors of the green node are realized, producing a local clustering coefficient value of 0. The average across all nodes in the graph gives an average clustering coefficient. This feature has a correlation coefficient of 0.67.

Nodes and edges in largest connected components

A set of nodes are said to be strongly connected if there is a path between all pairs of nodes within the set. In Figure 4, it can easily be seen there are three strongly connected components (SCC). The number of nodes in the largest SSC tells us how big the most connected part of the graph is. This feature has a correlation coefficient of 0.61 towards execution time. Likewise, a set of nodes are said to be weakly connected if there is any path regardless of direction between all pairs of nodes within the set. In the same figure 4, all of the nodes belong to one large weakly connected component as there exist some path between all pairs of nodes which doesn't necessarily need to be direct like SCC. This feature has a correlation coefficient of 0.52.

Number of triangles and fraction of closed triangles

A triangle is three nodes that are connected by either two (open triangle) or three (closed triangle) undirected ties. The fraction of closed triangle is the ratio of closed triangles to the total number of triangles. The number of triangles has a

correlation coefficient of 0.44, while the fraction of closed triangles has a correlation coefficient of 0.48.

Diameter and effective diameter

The diameter of a graph is the maximum eccentricity of any vertex in the graph. That is, it is the greatest distance between any pair of vertices. To find the diameter of a graph, first, find the shortest path between each pair of vertices. The greatest length of any of these paths is the diameter of the graph. The effective diameter is the shortest hops in which 90% of the nodes are covered. They have a correlation coefficient of 0.32 and 0.39, respectively.

The graph density based features tend to have a high correlation to the execution time as they determine the number of iterations for the algorithms we chose. More iterations correspond to more execution time.

Apart from these features, other graph properties were not selected as their correlation coefficient was less than 0.3, which we used as a threshold for selected features. These include :

– **Wedge count** - A wedge is defined as a path with two hops. A triangle has 3 possible wedges.

– **Square count** - Similar to triangles, a square is a closed loop enclosed by 4 edges.

– **Fill** - It is a metric to measure the fraction of existing edges to total number of possible edges

– **Spectral norm** - It is the largest absolute eigenvalue of the graph's adjacency matrix.

4.1.1.3 Prepossessing. Although we have learning set with graph data, hardware configurations, and its execution time as ground truth, it is still not ready for a model to be trained on. That requires a series of additional prepossessing steps to clean the learning set fully. The prepossessing steps in its respective order are:

Removing noise The most basic step in any data cleaning is removing noise. A noise comes in multiple types, but the most common type is missing features. For synthetic datasets, this isn't a problem as all the features are manually generated, and as a result, we have control over what features are needed. But for real-world graphs, this can be a problem as we scrubbed the features provided by SNAP and Konect for each of its datasets. Since we are getting our real-world graphs from two different sources, the features provided don't necessarily match. One such example is the wedge count feature. Konect offers wedge count for its datasets, but SNAP doesn't. Hence we had to interpolate wedge count for the experiments with SNAP dataset or drop those points entirely from the learning set. Dropping data points mean a smaller learning set, and since we already don't have too many points to spare, we resort to imputing those missing values rather than dropping them.

In general, there are some excellent multi-feature imputers available, but since only a fraction of experiments uses Konect datasets, a simple imputer does fine with this learning set. The sklearn's simple imputer uses one of three methods to fill missing, e.g., mean, median, mode while for our framework, we used a simple imputer with median fill.

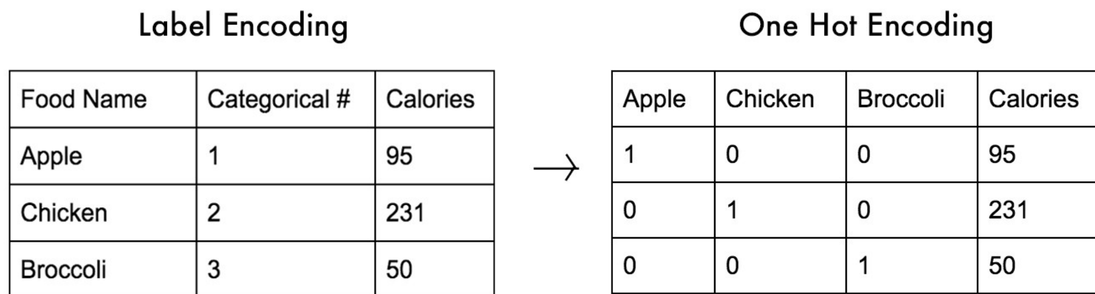
Another typical type of noise comes from features that have infinity as its value. This mostly occurs for features that rely on convergence for its final values.

During feature computation, when models don't converge, it sets its default value as float max, which can be considered as infinity. The only way to handle data points with unconverged features is to drop that data point entirely. But on a positive note, we ended up losing only 30 data points due to unconverged data.

Feature normalization Another important step for accurate predictions is for all the features to be scaled within the same range. Before normalization, different features have values in different ranges. For example, the value of the average clustering coefficient ranges between 0 and 1, while the number of edges and vertices are absolute counts. An ideal normalization would be to scale down all features between 0 and 1. For this purpose we use min-max scaling

Converting nominal data to numeric Learning set columns like algorithm and package have nominal categorical data but using a regression model(discussed ahead in the section) requires only numeric data, so we assign a unique numeric value to each category within a feature column. For example, there are four different algorithms; hence, we assign them unique values between 0 and 3. Likewise, we also assign values for the six packages.

One-hot encoding Now that the categorical data have been assigned numeric values, this promotes a new problem where the model doesn't recognize it as categorical data and treats those feature to scale rather than taking it for its face value. By not doing so, the value of category influences the weights assigned to features. For example, we have assigned the package GAP with category 2 and Galois with category 3. Although there is no underlying significance of the category number, the model treats Galois better than GAP.



Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

Figure 5. Example of 1-hot encoding

To avoid this problem, we encode the category into a 1-hot vector where the position in a vector addresses each category with its size being the number of categories. Figure 5 gives us an excellent example of how 1-hot encoding is done.

4.1.1.4 Fitting the model. After completing prepossessing, the learning set is ready for training. In this thesis, we will discuss four different configurations of models used with each one slightly tuned based on the previous. Results from each configuration and its analysis are discussed in the next chapter.

1. Linear regression with unnormalized learning set
2. Linear regression with normalized learning set
3. Ridge regression with unnormalized learning set
4. Ridge regression with normalized learning set

By analyzing the data points, we can see that they are not linearly separable. But going ahead in that direction, we initially tried using linear regression. Before exploring options for normalizing the learning set, we wanted to see how the models fared on raw data to identify new strengths and weakness

of each. The configurations with unnormalized learning set include features that aren't scaled to a 0-1 range.

Another problem we notice after analyzing the learning set is that the features undergo the phenomenon of multicollinearity. A collection of features are said to be multicollinear if they are linearly proportional to each other to a certain extent. In that case, the features can derive themselves from each other. This creates a problem as the main objective of regression analysis is to isolate independent features from each other and tune one feature at a time until predictions improve with keeping other features constant and thus finding the relationship between the dependent and independent features. But it's hard to estimate this relationship if the independent features change in unison.

For example, the synthetic graphs are generated with 16 edges per vertex on an average. As a result, the number of edges can be linearly derived to a certain extent from the number of vertices. Dropping one of these features is also not an option as when dealing with execution times in such small margins, the minor differences in the size of the graph contribute significantly towards the final predictions.

To solve this problem, we chose a ridge regression model, which is a class of regression that uses l2 regularization. Here the size of coefficients is limited by adding a penalty equalling the square of the magnitude of coefficients. To simplify, it adds a bias to each feature to avoid division by zero encountered by multicollinear features. Ridge regression on normalized and unnormalized learning sets make up the last two configurations. More on ridge regression and multicollinearity is given in the background section.

4.1.1.5 Experiments and Validation. To validate the accuracy of the model, the learning set is further divided into training and testing set. The split in data points consists of a 70% allocation to training set and 30% allocation to the testing set from a total of 4000 data points. This is done by Sklearn's test_train_split() with splits into random test-train subsets.

4.1.2 Classification of graph processing experiments. Predictive modeling of execution time is great, but interpreting and coming to a conclusion with such small values can be challenging. Minor inaccuracies in predictions will significantly add up across the course of an application running thousands of experiments. But instead, a user would greatly benefit if he or she knows whether a specific package is worth using for a particular experiment. For this reason, we have implemented a classification framework that predicts a binary class label of "good" or "bad" based on the features of the graph dataset and hardware configurations.

4.1.2.1 Preprocessing. To perform classification, we must perform the same preprocessing steps as the regression framework except for converting nominal to numeric. Apart from that, we must also add a class label for training.

Traversed edges per second We could use execution time solely as the metric for classifying between a good and bad experiment, but that wouldn't take the size of graph datasets into account. To use an appropriate metric that scales with the size of the dataset and is also a function of execution time, we use traversed edges per second(TEPS) as a metric for classification. TEPS is defined by :

$$TEPS = \frac{Number\ of\ edges}{Execution\ time}$$

TEPS is a useful metric for this problem as it not only scales with execution time but also captures both the computational and communication capabilities of

the experiment and hardware. One must note that larger TEPS indicates better performance as it's a factor of execution time rather than a multiple of it.

Binary label classification

For this framework, we decided to use a binary classification of "good" and "bad" for a particular experiment based on the TEPS value. For any experiment n, it's binary class label could be defined as:

$$label(n) = \begin{cases} good & \text{, if } x > 2/3((\sum_{n=1}^{N} x_n)/N) \\ bad & \text{, if } x < 2/3((\sum_{n=1}^{N} x_n\ )/N) \end{cases}$$

,where N is the number of experiments in the learning set and x is the TEPS value of a particular experiment. Intuitively, this function classifies an experiment as "good" if it has a TEPS value of less than 2/3 times the mean across all the TEPS values in the learning set and "bad" otherwise. This split approximately labels 30% of the learning set as "good" and 70% as "bad".

4.1.2.2 Classification model. Similar to predictive modeling, we will have two configurations of models with the latter improving upon the former.

1. Logistic regression with normalized learning set
2. Random forest with normalized learning set

Even though it was established that the learning set is not linearly separable, our first model is logistic regression just for the sake of comparison with our random forest model. More details on the two models are given in the background section.

4.1.2.3 Experiment and validation. To validate the accuracy of the model, the learning set is further divided into training and testing set. The split in data points consists of a 70% allocation to training set and 30% allocation

to the testing set from a total of 4000 data points. This is done by Sklearn's test_train_split() with splits into random test-train subsets.

4.2 Selection of solvers for a system of linear equations

For any software selection framework, a classification model based on performance is a great first step, but a binary classification has significant drawbacks on its own. First off, users need to provide their choice of package for which the framework will classify as "good" or "bad" based on the experiment setup. This is added work for a user who doesn't really care about the package and wants a good one to use. But more importantly, a binary classifier may yield a lot of possible good packages giving the user further decisions to make, thereby defeating the primary purpose of this framework. It would be a significant improvement if the framework could provide what the best package would be for that experiment. Or better yet, a ranked list of packages for that experiment.

4.2.1 Ranking framework. To achieve the goal mentioned above, we look at a different class of application to the previous graph processing one. Specifically, we aim to provide a framework that gives a ranked list of linear solvers for a system of linear equation.

4.2.1.1 Learning set. Similar to the previous application, this also contains experimental setup along with its features and ground truth value as data points of the learning set. The experimental setup, in particular, is a system of linear equation in the form of matrix A and right-hand vector B together represented as system M. Each system M has a list of solvers and preconditioners. We use two different learning sets with one each for parallel and serial execution.

4.2.1.2 Two stage ranking algorithm. After analyzing the results of both classification and predictive modeling, both of which are discussed in the

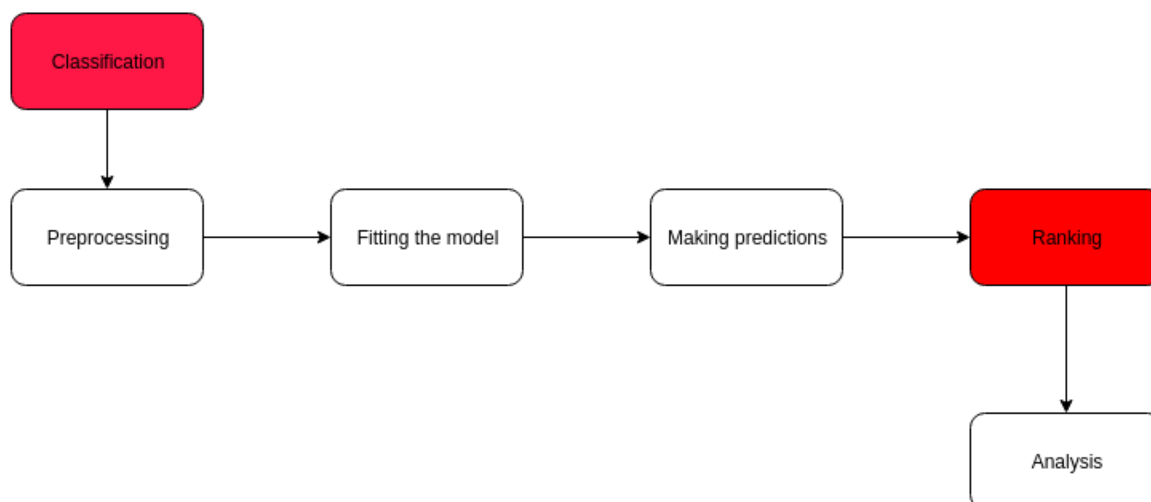


Figure 6. Overview of the ranking algorithm

next chapter, we provide a ranking algorithm that tries to benefit from the unique advantages that both offer on the learning set.

The basic idea behind this algorithm is the fact prediction works well on data that has been classified "good" or "bad". When the class label is added as an additional feature for predictive modeling, the quality of predictions improve. The two stages refer to the two models for classification and prediction being used by the algorithm. With a reasonable enough prediction accuracy of execution time for different experiments, they could potentially be sorted to give a good ranked list of solvers that work best for a particular system of linear equations. Figure 6 provides us with the workflow of the two-stage algorithm.

Preprocessing and classification

The prepossessing done on the learning set for linear systems before we can fit our models is similar to the steps done to the graph processing learning set discussed in section 4.1.2.1 apart from the fact that there are no categorical data and hence we don't need to convert to 1-hot encoding. The classification is also

similar to graph processing class of algorithms discussed in 4.1.2.2 but it uses a C.45 model rather than random forest.

Fitting the model and hyperparameters After classification, the learning set with a "good" or "bad" class label is ready for predictive modeling. By analyzing the results from predictive modeling in the graph processing application, We decided to use the final model configuration that uses ridge regression [10] on the normalized dataset. Hyperparameter tuning was done with the help of 5-fold cross-validation. After tuning, the best set values for hyperparameters were:

1. **Regularization strength** Since ridge regression uses L2 regularization, a regularization strength must be provided. After tuning from cross-validation, we settled on a value of 0.8.

2. **Intercept** This determines whether or not to fit an intercept for this model. Data that centered don't need an intercept, but that wasn't the case with our learning set. Hence we decided to use an intercept.

3. **Normalize** Because we performed our own normalization to the learning set, we opted not to choose Sklearn's default choice of L2-normalization.

4. **Solver** This determines what choice of solver must be used to compute the ridge coefficients. For this problem, we used a singular value decomposition (SVD) solver.

Sorting

After training, the model is ready for predictions and ranking. The ranking is done by sorting the predicted values using the quicksort algorithm. It is well established that quicksort is the fastest sorting algorithm with time complexity of $O(n\ log\ n)$.

Table 1. Hyperparamets and its final value

Hyperparameters	Values
Regularization strength	0.8
Intercept	True
Normalize	False
Solver	SVD

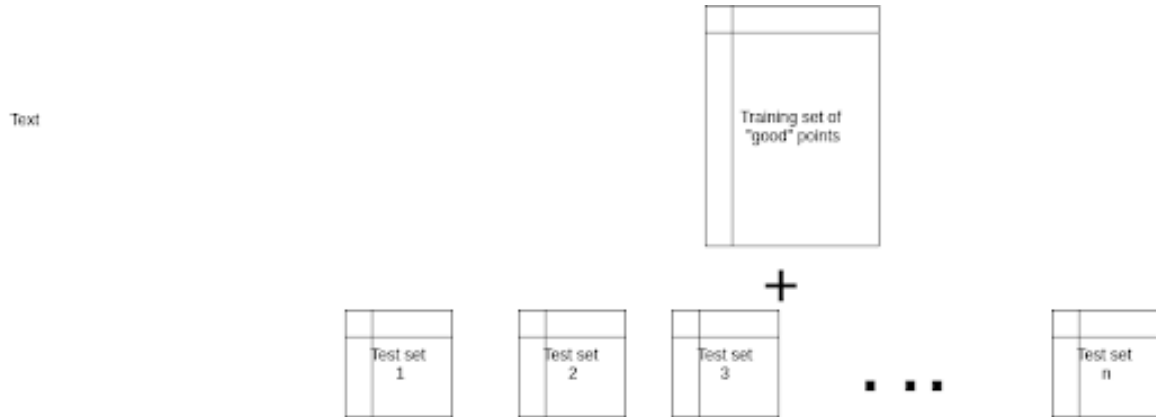


Figure 7. Validation setup for testing

4.2.1.3 Experiment and validation. Our experimental setup for the SuiteSparse matrix collection involves solving each system of equations by using a set of 120 linear solvers and preconditioners offered by the PETSc library thereby giving 127,569 different experiments. Similarly, systems in MOOSE are solved by a set of 70 linear solvers-preconditioner combinations offered by PETSc.

The learning set is further split into one combined training set and individual testing sets with 1 per each system of equations. This can be visualized by figure 7, which is the setup used to validate the experiment. After, classification and before predictive modeling, experiments that are tagged “good” make up 85% of the training set while the rest is made up of “bad” experiments. The selection of good experiments for training is made in random.

The idea behind having a training set dominated by “good” experiments is that it is a lot harder to predict those experiments since the execution times are to the power of -5 or lesser. This means even the slightest of inaccuracies can lead to drastic speed downs. On the other hand, “bad” experiments are to the power of -3, thereby reducing the impact of slight inaccuracies making it much easy to predict.

To make sure the model isn’t over-fitting to “good” experiments, we made the testing sets entirely with “bad” experiments and just a single “good” experiment. Although in reality, one would rarely encounter a system of equation with only the choice of bad solvers, we wanted to simulate how the ranking framework performs in the worst conditions.

CHAPTER V

RESULTS AND ANALYSIS

In this chapter, all possible results and metrics to evaluate the proposed framework, followed by its respective interpretations are provided. Final conclusions on the framework as a whole with its strengths and weaknesses are given in the conclusion chapter. We will first look at evaluation metrics for predicting the execution time of parallel graph processing experiments followed by metrics for evaluating its respective classification. Lastly, we will discuss the evaluation metrics for predicting and ranking experiments a for system of linear equation.

5.1 Parallel graph processing experiments

We have proposed solutions for two different classes of problems within graph processing experiments. The prediction problem is evaluated using three metrics, namely the R^2 value, Root-Mean Squared (RMSD) and Normalized Root-Mean Squared Deviations (NRMSD). We evaluate the classification problem on the number of false positive and negative predictions. This can be visualized using a confusion matrix. We also introduce a new metric that measures the improvement in performance resulting from choosing the experimental setup provided by the framework.

5.1.1 Execution time prediction.

Figure **??** shows us a consolidated comparison of all three metrics for each model configuration across four different algorithms. The R^2 value (left plot) signifies the goodness of fit in contrast to a mean model, which uses the mean for every predicted value. This is based on the observation that a well-fitting regression model results in predicted values close to the observed data values, and the proposed regression model should,

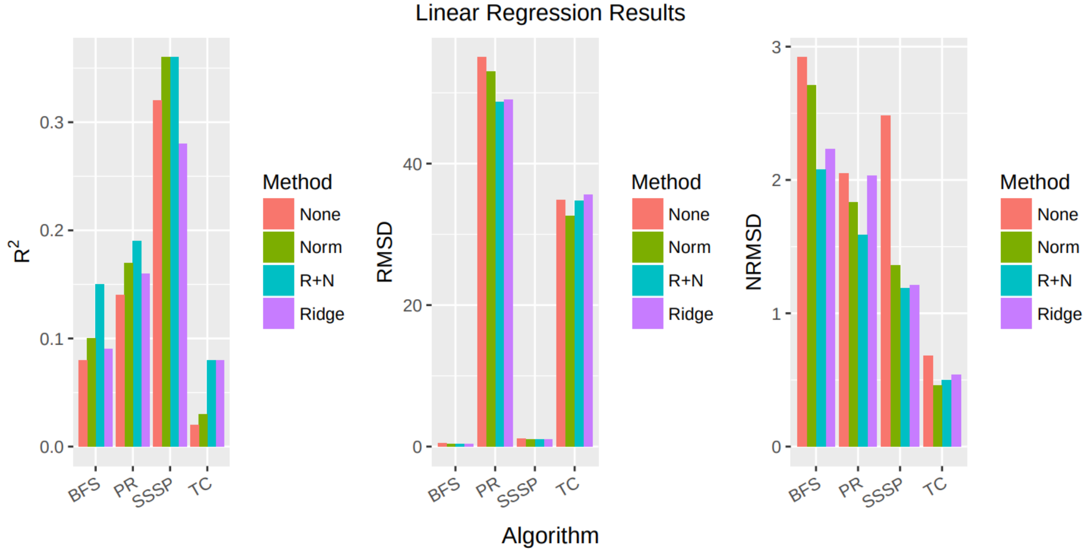


Figure 8. Linear regression with and without ridging and normalization

therefore, be better than the fit of the mean model. It ranges from 0-1 with higher values indicating a better fit.

On the other hand, RMSD (middle plot) measures the closeness of predicted values to the actual value. While R^2 is a relative measure of fit, RMSD is an absolute measure of fit. Lower values of RMSD signifies better fit. However, RMSD is hard to compare across algorithms. Algorithms like BFS has a much lower execution time compared to likes of TC. Thus, we use NRMSD (right plot).

It can be noted that for all algorithms across three metrics, the model configuration with ridge regression and normalized learning set is always best or at the very least tied for best. Although this configuration gives a significant improvement in the quality of predictions, it still is not reliable enough to replace empirical models that run experiments rather than predicting them. Based on the size of our learning set, we must see R^2 values above 0.7 for it to be considered a reliable replacement.

(a) Logistic Regression

		good	bad
TC	good	0	49
	bad	0	113
		good	bad
BFS	good	133	9
	bad	91	4
		good	bad
PR	good	161	0
	bad	81	0
		good	bad
SSSP	good	27	85
	bad	17	113

(b) Random Forest

		good	bad
TC	good	43	1
	bad	0	118
		good	bad
BFS	good	110	0
	bad	0	147
		good	bad
PR	good	75	0
	bad	1	193
		good	bad
SSSP	good	51	11
	bad	9	198

Figure 9. Confusion matrices for all algorithms

Poor prediction performances can be attributed to the fact that execution times of experiments are minimal, specifically to the power of -5. To complicate this, the range of possible values is also quite big, ranging anywhere between the power of -2 and -5. This makes predictions more like finding a needle in a haystack.

5.1.2 Classification of experiments. The second model we propose in the framework is a classification model that aims to categorize a particular experiment as “good” or “bad” based on its traversed edges per second(TEPS).

Figure 9 (b) gives us the confusion matrix for the random forest model and just for comparison, the confusion matrix for logistic regression in figure 9 (a). The confusion matrix is used to measure the number of true and false positives and likewise, its negative counterpart. The rows indicate class labels from actual observations or ground truth while the columns indicate the class labels that were predicted by the models. For example, the random forest model predicted 44 different experiments that run TC to be good but turns out only 43 were observed

to be good while a single experiment was actually bad. This means the model made 43 true positive predictions and one false positive prediction. Likewise, the model predicted 118 experiments to be bad, and indeed, all 118 experiments were actually observed to be bad. This means the model made 118 true negative predictions and zero false negative predictions.

A general rule of thumb for well-performing models is to have confusion matrices with large leading diagonals and small non-leading diagonals. With that in mind, one can see that random forest performs exceptionally for TC, BFS, and PR with a maximum of one misplaced prediction from over 200 predictions. For SSSP though, the predictions are not quite as accurate, but are nonetheless good with only 20 misplaced predictions.

This gives us a **total testing accuracy of 97%** across four algorithms. In comparison, one can see that logistic regression falls far behind with significant misplaced predictions. Apart from the overall count of misplaced predictions, the type of misplacement matters. Having more false positives is worse than having more false negatives as users don't mind having a few "good" experiments being termed "bad" resulting in them not using a package. But on the other hand, having "bad" experiments tagged as "good" leads them to go ahead and use a package and as a result cause significant slowdowns. It is important to note that the end objective of this framework is to provide reliable options for package selection, and having few false negatives is fine at the cost of minimal false positives.

Another metric we introduce to evaluate the performance of the classification model is **improvement**. This is mainly used to signify how much of performance improvement a user gets by choosing a "good" labeled experiment

Algorithm	Min	Max	Mean (all)	Mean (good)	Improvement (%)
TC	1.2e2	2.3e3	6.8e2	1.6e3	66.6
BFS	2.8e3	3.1e10	1.1e8	8.1e8	700.7
PR	7.6e1	1.1e5	4.4e3	4.7e3	6.8
SSSP	2.3e2	2.6e10	8.2e7	1.1e8	34.1

Figure 10. TEPS for each algorithm

setup given by the framework rather than using any random experiment setup. Improvement can be defined as

$$Improvement = \frac{Mean(TEPS\ of\ data\ labeled\ as\ Good)}{Mean(TEPS\ of\ all\ data)}$$

Figure 10 gives us the performance improvement along with the min, max, and mean across the four algorithms for the random forest model. It should be noted that larger TEPS value indicates better performance since it is a factor of execution time rather than its multiple. We can see that TC and SSSP have a good performance improvement while BFS tops the chart with a remarkable 700% improvement. The performance improvement for PR is not as high as the other algorithms. The low variance explains this in execution time and TEPS of PR. The coefficient of variation (standard deviation/mean) of TEPS for BFS is 1.5, and PR is only 0.21.

5.2 Experiments on solving linear systems

The second class of algorithm in which we implemented our selection framework is for solving linear systems. Similar to graph processing algorithms, we have implemented predictive modeling but instead of classification, we have

implemented a ranking framework which is discussed in detail in section 4.2.1. The results for all the proposed models are discussed in detail in this section.

5.2.1 Predicting execution times for experiments.

In this section, we will first discuss the results for predicting execution times of different experiments solving a system of linear equations. Once again, the three metrics, namely, R^2 values, root-mean-squared-deviation, and its normalized counterpart, are used for evaluating the prediction accuracy. From ***, you can see that the R^2 values are similar or even slightly worse compared to that of the graph processing application. This is once again because the data points are not linearly separable, and even a ridge regression model is not accurate enough. The RMSD and NRMSD values are slightly better with all test sets having values less than 0.8.

With these results, we can say the predictions are not yet reliable enough. But a critical aspect that we can observe from the learning set is that the range of execution times across experiments is pretty big and ones that are considered good have far smaller times than the ones considered bad. As a result, there is a strong separation between them, even with the predictions. This was the key inspiration behind the proposed ranking algorithm.

5.2.2 Ranking experiments for linear systems.

This section will provide the ranking results for the two different learning sets we ran experiments on. The main metric we use to draw conclusions on is the speedup, which is given by:

$$Speedup = \frac{Time\ taken\ by\ baseline\ solver}{Time\ taken\ by\ \#1\ solver}$$

This definition signifies the speedup in execution time one gets by choosing a solver recommended by our framework rather than choosing the default solver offered by PETSc, which is our baseline. To clarify, the speedup isn't actually from

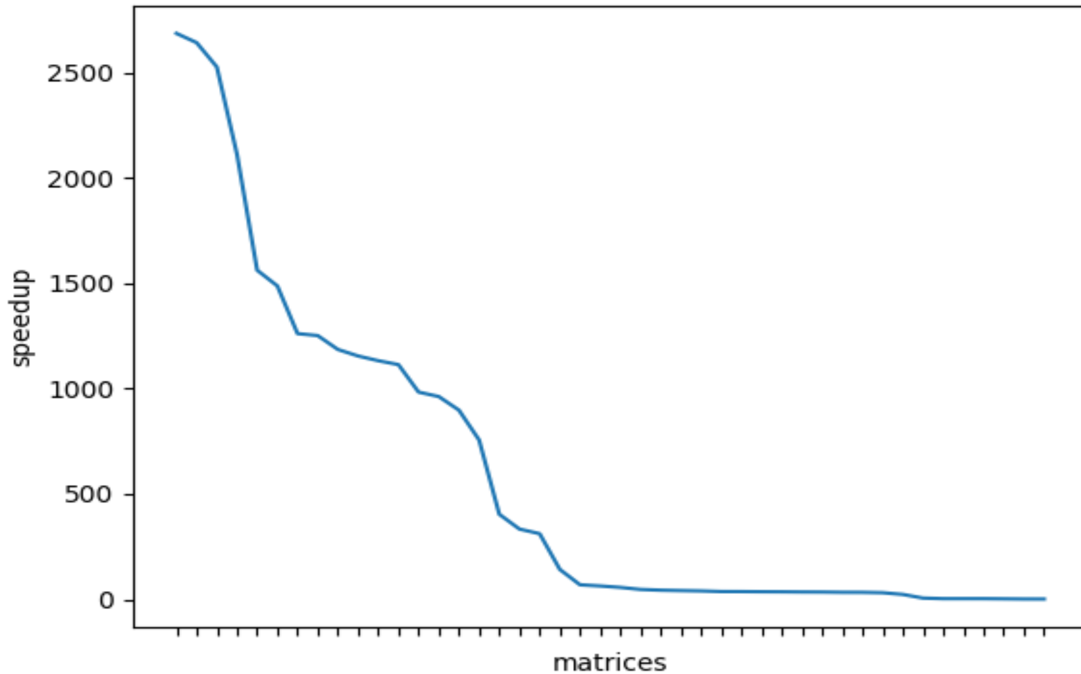


Figure 11. spread of speedups across testing sets

improving the solvers by themselves but rather from selecting better solvers for that particular linear system and its experimental setup. We will also discuss other metrics like average precision and reciprocal ranks to solidify our conclusions.

SuiteSparse Matrix Collection:

As previously discussed, we have trained our model with a training set dominated by experiments labeled “good” and tested it across 1041 different linear systems with each having its own testing set dominated by experiments labeled “bad” From table 2, we can see that even the minimum speedup is 1.14, which means the number one solver that the framework provides is better than the baseline solver for all the testing sets, which in this case is GMRES with Block Jacobi [17]. In fact, the proposed solver is way better than the baseline solver with an average speedup of 746 and peaking at 92,464 across all testing sets.

Table 2. Statistics of speedups across all testing sets

speedup stats	Values
Min	1.14
Max	92464
Mean	746
Standard deviation	37

Figure 11, shows us the spread of speedups across 100 randomly chosen testing sets (linear systems). The x-axis denotes all the systems in decreasing order of execution time while y-axis indicates the speedup. We can note that speedups decrease in proportion to the execution time. This can be attributed to the fact that the prediction accuracy of our ridging model also decreases with execution time, and as a result, the proposed rank positions are way off from their actual rank positions. But considering speedups alone can be misleading.

As mentioned in the introduction, the objectives of the ranking framework is to not only attain better speedups but to also ensure reliability. Reliability here means always suggesting good solver selections. Average speedups amount to very little if at the end of the day, the user still has to use a bad solver for an experiment. The user is most likely to choose solvers in order of rank and we need to ensure there isn't a choice of bad solvers before good. Figure *** shows us the top 10 ranked list of solvers for 1 example test set. It could be seen that the #1 and #2 ranked solvers are indeed solvers labeled "good" followed by the rest of bad solvers. This trend has followed in all test sets where the #1 ranked solvers are good. A good metric to signify this trend is Mean-Reciprocal-Rank (MRR). RR in general for a particular test set is given by :

$$RR = \frac{1}{Highest\ rank\ of\ a\ good\ solver}$$

label	execution time	AbsoluteTrace	Trace	ColumnDiagonalDominance	AbsoluteNonZeroSum	Dimension	DiagonalMean	FrobeniusNorm	solver	rank
good	0.0015118	271901	271901	0	564748	420	647.382	143757	74995666	1
bad	0.0021513	271901	271901	0	564748	420	647.382	143757	99720138	2
bad	0.014238	271901	271901	0	564748	420	647.382	143757	96689089	3
bad	0.0019808	271901	271901	0	564748	420	647.382	143757	96590062	4
bad	0.002701	271901	271901	0	564748	420	647.382	143757	96590061	5
bad	0.0020504	271901	271901	0	564748	420	647.382	143757	95762355	6
bad	0.0028265	271901	271901	0	564748	420	647.382	143757	95279263	7
bad	0.0016584	271901	271901	0	564748	420	647.382	143757	94599140	8
bad	0.001965	271901	271901	0	564748	420	647.382	143757	94599137	9
bad	0.03324	271901	271901	0	564748	420	647.382	143757	92331214	10

Figure 12. Top 10 ranked experiments

For example, from figure 12, we can see that the highest rank of a good solver is #1 and as a result, it has a reciprocal rank of 1 which is the highest possible value. Had the good solver been placed at #3, the RR would have a value of 1/3. The mean value of RR across all testing sets is MRR. The MRR value we got is again 1 which signifies that the predicted #1 solver is never bad. This goes a long way in ensuring reliability.

Table 3. Statistics of speedups across all testing sets(MOOSE)

speedup stats	Values
Min	1.2
Max	857
Mean	7.58
Standard deviation	34

MOOSE:

Likewise, the experiments run on our second learning set also yielded similar results. For the model tested across 435 linear systems, table 3 shows us the statistics of speedups across testing sets. Although the speedups are not as high as that of SuiteSparse Matrix Collection, it still follows the same trend of always having a positive speedup meaning, the #1 ranked solver is always better than the baseline solver which in this case is GMRES with ILU, having a factor level of 0. The relatively low speedup is mostly due to the smaller size of training set. From the table, we can also say that on an average, our recommended solver is

seven times faster than the baseline solver which PETSc offers, for any specific experiment setup.

CHAPTER VI

CONCLUSION AND FUTURE WORK

In this thesis, we have proposed a framework for suggesting high-performance scientific toolkits catered to specific problems and experimental setup by using a model-based approach. On the way to guarantee both improved performances and reliability, we have explored a variety of models. Some models are not as of yet reliable, but we have analyzed and discussed them in this thesis nonetheless. In specific, predictive modeling has been a challenge due to various constraints in available data.

But from our proposed results, we can also say that our framework is reliable in classification. It can classify if a specific experimental setup for processing a graph dataset in parallel is good or not with **97% accuracy**. We can also say that selecting the experiments which are predicted to be good by the framework provides an average of 200% improvement in performance across four algorithms.

Similarly, we can also say that our framework is reliable in ranking experiments. In the previous chapter, we established that from our ranked list of solvers for over 1,500 different test sets across two separate learning sets for parallel and serial execution, **the #1 ranked solver is never a bad solver**. Also, the **the #1 ranked solver is always better than the baseline solver**, which is the default solver in PETSc. Based on the speedups for serial execution with MOOSE learning set, we can say that the #1 ranked solvers are on average seven times better than the baseline with a few solvers peaking at 800-fold improvement. Likewise, we can say that for parallel execution with the SuiteSparse learning set,

the #1 ranked solvers are 700 times better than the baseline with a few solvers extraordinarily peaking with 90,000 times improvement in performance.

To summarize, we have provided reliable solutions for two of the three objectives set out for the framework, thereby making it a much better alternative for choosing HPC toolkits than choosing it based on empirical results from running time-consuming experiments.

In the future, we plan to add auto-generated features that are created using graph2vec embeddings [19] rather than hand crafted features we currently use. By doing so, we are not restricted by the number of features and can create a much larger feature set in the order of 100's. A larger feature set also gives us the ability to accurately run more complex deep learning models

REFERENCES CITED

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

[2] Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, Sherif Sakr, et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[3] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

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

[5] Are Magnus Bruaset. *A survey of preconditioned iterative methods*. Routledge, 2018.

[6] Boguslaw Butrylo, François Musy, Laurent Nicolas, Ronan Perrussel, Riccardo Scorretti, and Christian Vollaire. A survey of parallel solvers for the finite element method in computational electromagnetics. *COMPEL-The international journal for computation and mathematics in electrical and electronic engineering*, 23(2):531–546, 2004.

[7] Edmond Chow, Robert D Falgout, Jonathan J Hu, Raymond S Tuminaro, and Ulrike Meier Yang. A survey of parallelization techniques for multigrid solvers. In *Parallel processing for scientific computing*, pages 179–201. SIAM, 2006.

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

[9] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[10] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[11] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[12] Casey Icenhour, Shane Keniley, Corey DeChant, Cody Permann, Alex Lindsay, Richard Martineau, Davide Curreli, and Steven Shannon. Multi-physics object oriented simulation environment (moose). Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2018.

[13] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[14] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 454–469. Springer, 2011.

[15] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[16] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Parallel sat solver selection and scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 512–526. Springer, 2012.

[17] Pate Motter, Kanika Sood, Elizabeth Jessup, and Boyana Norris. Lighthouse: an automated solver selection tool. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 16–24. ACM, 2015.

[18] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[19] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[20] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

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

[22] George AF Seber and Alan J Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.

[23] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[24] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[25] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25. IEEE Computer Society, 2005.