Performance Prediction of Multithreaded Applications

Abhinav Gupta
Computer Science Dept.
New York University
New York, NY, USA
gupta.abhinav@nyu.edu

Rachit Jain
Computer Science Dept.
New York University
New York, NY, USA
rj2219@nyu.edu

Mohamed Zahran
Computer Science Dept.
New York University
New York, NY, USA
mzahran@cs.nyu.edu

Abstract

As multicore programming is hugely dictating modern day computing, parallel programming is becoming increasingly important. As multithreaded applications are not guaranteed to attain speedup and are generally costlier to setup, it is becoming increasingly necessary to profile and predict performance of parallel programs before deploying them over multiple cores. In this paper, we compare several machine learning algorithms to accurately predict the speedup which can be attained by parallelizing the given program as compared to its serial counterpart. We evaluate our methodology by testing our algorithms on publicly available benchmark of Princeton Application Repository for Shared Memory Computers (PARSEC 3.0) and several other self written programs in OpenMP. We have implemented Synthetic Minority Oversampling Technique (SMOTE) for removing data skewness which seems to play a major role in generalization of the program. As per our findings, Random Forest Regressor performs the best with SMOTE applied to it with a mean average error of 0.7270.

Keywords - Performance modelling, parallel applications, PARSEC, execution time prediction, multicore, machine learning, SMOTE.

1 Introduction

With rising architecture requirements, need for parallel software is increasing dramatically. A key challenge in predicting the performance of parallel code is finding whether a significant speedup can be achieved and what are the optimal set of parameters which help us achieve that speedup. One way of approaching this is by running simulations. In practical circumstances, it is very difficult to run simulations on such large problems since the time to run these simulations is significant. Furthermore, running fixed analysis models generally do not lead to accurate results. Converting a serial code to a parallel version of the code should be able to justify the man hours which are spent on the conversion. There is a possibility that the speedup being achieved is not significant/comparable to the process. This will result in a loss of development time and effort without leading to any efficient results. For this purpose, we propose to predict the performance of multithreaded applications by using various regression and neural network methods. By doing so, we can achieve results quickly on large scale problems and optimize the level of parallelism which is required.

We propose a learning based approach to the prediction process in which we have built the dataset from the publicly available benchmark - PARSEC 3.0 Bienia et al. [4] and written several of our own programs in OpenMP for testing. We profile the code using *perf* and get a list of features which serve as features for the programs. Since the dataset is highly imbalanced, we use the SMOTE Branco et al. [6] technique for removing data skewness. After that we use several learning algorithms for performance prediction. We use the correlation matrix for important feature selection. Our main contributions are the following:

- Designing a framework for performance prediction for parallel programs.
- Implementing the SMOTE technique for data balancing.
- Predicting the speedup and the optimal set of parameters with which maximum speedup can be achieved.
- Using correlation coefficient for feature/parameter selection.

The remainder of the paper is as follows: Section-2 discusses the literature survey of the performance prediction topic. Section-3 outlines the proposed idea and the methodology used for dataset creation, feature selection, and the learning algorithms. Section-4 discusses the experimental setup of the project and the specifications on which the experiments were run. Section-5 discusses the results and their analysis of the experiments. Finally, in Section-6 we conclude the project and discuss future enhancements.

2 Literature Survey

There have been numerous studies of performance prediction of multithreaded applications. In Adve et al. [1], performance is being predicted by building analytical models which require knowledge of parallel code. These models are accurate but do not generalize well on unseen data which makes it less scalable. Bagrodia et al. [3] analyzed performance by running simulations on the programs to measure time for execution and calculating speeds ups. This method proves to be accurate but suffers from high costs for running simulations and does not scale well. In Joseph et al. [8], the authors develop regression models for the prediction but for superscalar processors and run simulations to test their programs. Ipek et al. [7] developed a training based approach on SMG200 using Artificial Neural Networks. Agarwal et al. [2] employs the use of several regression algorithms by varying the number of input threads and size of the program. This method is able to generalize well but is not able to scale well to larger problems since dataset skewness arises for bigger problems. By contrast, our approach uses several machine learning algorithms for the prediction and handles data skewness by applying the SMOTE technique and feature extraction using correlation matrix. We believe that the novelty for our method lies in the part where our model is handling the data skewness which makes the approach scalable on unseen data and more reliant in the practical scenario.

3 Proposed idea

We propose a methodology for testing multi domain workloads on a machine in OpenMP for carrying out performance prediction and logging the parameters which affect the performance the most. We define a methodology flowchart as shown in Figure 1 to show the basic outline of our idea. We propose to see how the performance changes when the number of threads and the input size is varied. This will give us an insight into what parameters are affecting the execution the most. With this light weight approach, the programmer will be able to get an estimate speedup before implementing the solution. We are carrying out our

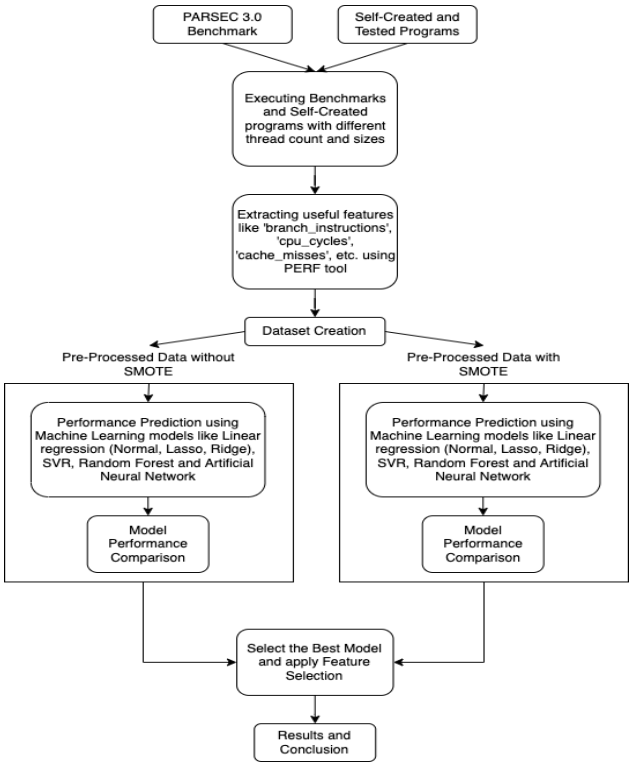

Figure 1: Methodology Flowchart

experiments on a single machine whose specifications are listed in Table 2.

3.1 Dataset

According to Agarwal et al. [2], a combination of Perf and Valgrind helps in extraction of various features for each application and they found that *Number of cycles*, *Branches* and *Branch miss-predictions* were the most important features. Taking inspiration from the paper and after extensive research using the Perf profiling tool, we extracted the following 14 intrinsic features for each of the application (sampled on different number of threads and sample size). We think that these features combine both hardware and program features and they impact the performance of a multithreaded application the most without needing information about various microarchitecture features. The features we used are:

- **branch-instructions, branch-misses:** Both are hardware events. We can have a negative effect on the performance if there are high number branch misses but branch instructions does not affect the speedup that much as now-a-days machines are highly capable.
- **L3-cache-misses, L3-cache-miss-rate, L3-cache-references:** Both cache misses and references are hardware events. Cache References denote the count of total L3 cache references ir-

respective of whether it is a miss or pass. Cache misses denote the miss count that happens while accessing the L3 cache level which means the number of times the application had to access the disk to procure the data frame. As accessing disk is much more costly than accessing cache/memory and as cache coherency will kick in, so there will be a higher number of cache misses which can affect the speedup.

- **L1-data-cache-loads, L1-instruction-cache-load-misses:** Both are hardware events and tell us about the count of L1-cache load hits/misses. Number of cache loads/hits definitely affect the application running time.
- **LLC-load-misses:** This is a hardware event which tells us about the number of loads that miss in the last level cache.
- **cpu-cycles, cpu-clocks** Both are software events where *cpu-clocks* tells us about the time passed in CPU clock events (milli-seconds) and *cpu-cycles* tells us about the size of the application (i.e. proportionate to the number of instructions). Large applications may benefit from parallel architecture and can show speedup.
- **total-instructions, IPC:** Both are hardware related features where the first one tells us about the total number of instructions fired from the user space of the application and the later tells us about the efficiency of the processor (i.e. instructions handled per cycle)
- **page-faults:** This is a software event which tells us about the number of page faults that happened while running the application. Poor memory management can be a big reason for lack of speedup.
- **Exe-time:** This is a software event which tells us about the time spent by the processor in running the application. Although we cannot correlate execution time with speedup directly as speedup depends on the ratio of serial/parallel running times, but it still gives some vague idea about it.

3.2 Feature selection

Feature selection is an important step of our methodology. Using *perf*, we were able to extract multiple performance counters. After careful analysis, it was found that some of the features might not identify with the problem statement and will be used as extra parameters. To handle this scenario, we use a correlation matrix to take the features which align most with our output variable **speedup**. We use the correlation function present in *pandas* library. We plot the results of those correlation values in Figure 2. The figure depicts the correlation value that each feature has with every other feature. Since we require features most correlated to the speedup, we sort the matrix with the most relevant top 10 features and select them for usage in our methodology.

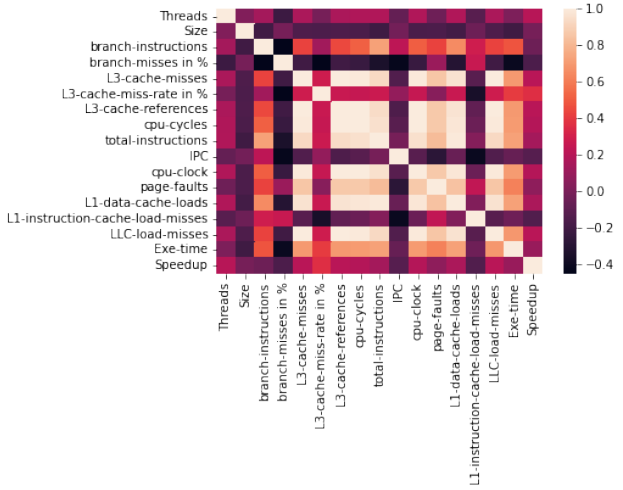

Figure 2: Correlation Heatmap

3.3 Machine learning algorithms

Learning algorithms form the basis of this paper. In this section, we analyze the different machine learning algorithms that were used in the testing of this methodology. We have compared - Linear, LASSO, Ridge, Random Forest, Support Vector Regressions and an Artificial Neural Network.

3.3.1 Linear Regression

Linear regression is a linear model which assumes a linear relationship between the input variables (X) and a single output variable (y). For our case, we are using multiple linear regression since we have mulitple input variables as features. This model uses linear predictor functions for calculating weights of the features. We use this model from scikit-learn library.

3.3.2 LASSO Regression

Lasso regression is a type of linear regression which improves upon it. It stands for Least Absolute Shrinkage and Selection Operator. It not only helps in avoiding overfitting but it also helps in feature selection. This model uses L1 regularization where it uses shrinkage which means the data values are shrunk towards a central point thereby removing extraneous features. This model produces sparse models. We define max iterations as 1000 and normalize the data and the cross

Application	Domain	Program Size (simlarge)
Blackscholes	Financial Analysis	65,536 options
Bodytrack	Computer Vision	4 frames, 4000 particles
Facesim	Animation	1 frame, 372,126 tetrahedra
Ferret	Similarity Search	256 queries, 34,93 images
vips	Media Processing	1 image, 2662x5500 pixels
Canneal	Engineering	400,000 elements
Swaptions	Financial Analysis	64 swaptions, 20,000 simulations
Streamcluster	Data Mining	16,384 points per block
x264	Media Processing	128 frames, 640x360 pixels
Fluidanimate	Animation	5 frames, 300,000 particles
Freqmine	Data Mining	990,000 transactions

Table 1: PARSEC 3.0

validation parameter as 10. We use this model from scikit-learn library.

3.3.3 Ridge Regression

Ridge regression is a type of linear regression. It solves a regression model which uses the linear least square function as its loss function and l2-norm regularization. This regression is built for multiple variable regressions. This type of model does not eliminate coefficients like LASSO regression. We normalize the data before sending it as input to the model and the cross validation parameter as 10. We use this model from scikit-learn library.

3.3.4 Random Forest Regression

Random Forest regression is a supervised learning algorithm which uses ensemble learning method for regression. This technique combines multiple algorithms to build a final model. All the decision trees run separately and finally the mean of all the models is taken to produce the final output. This model will prevent overfitting as average of the models is utilized. It performs feature sampling to build smaller datasets for each forest of the model. We use this model from scikit-learn library.

3.3.5 Support Vector Regression

Support Vector regression is a type of regression which is flexible for defining the amount of error which is acceptable. This model aims to minimize the coefficient. We use the radial basis function for the kernel of the model. It minimizes the distance between the threshold and the boundary line of the input points. We define the threshold as 10 in our experiments. We use this model from scikit-learn library.

3.3.6 Artificial Neural Network

Artificial Neural Networks are a series of algorithms which try find relations between data. They learn these relations using learning algorithms and by minimizing their loss functions. Each node in a neural network is a neuron like structure. It consists of an input layer and an output layer and can consist of hidden layers. Each neuron to neuron link will have a weight associated to it which signifies the importance of a feature. We use this model from the keras library. Our input parameters:

- Standardized the data using scikit-learn StandardScalar. Data standardization becomes important so that the extreme values do not affect the output so much.
- We have implemented a Sequential model from Keras in which we add 3 dense layers in the code with 16, 8, 8 hidden dimensions neurons.
- Activation function being chosen is *relu* and the optimizer is *adam*.
- Our input size is 16 neurons and output size is 1 neuron.
- Our loss function is Mean Absolute Error (MAE).
- We create checkpoints of the model to store it in case of some unexpected errors.
- We run it for 100 epochs with a batch size of 32.

3.3.7 SMOTE - smogn

Synthetic Minority Oversampling Technique (SMOTE) is the process of balancing dataset skewness. In our case, there is data imbalance due to most of the data being around the speedup of 1. Due to this, there is a huge class imbalance as can be seen in Figure 3. Since the sample points are close

in our case, we use the nearest neighbour concept for applying the *smoter* function which is SMOTE for Regression. This technique works by selecting a random point, checking its nearest neighbours, selecting a random neighbour from that sample space, and then creating a sample data point between the two randomly selected points. In this way, the minority class is oversampled towards the balanced side. This data balancing process will help the final model generalize on unseen data. We use the smogn library for this.

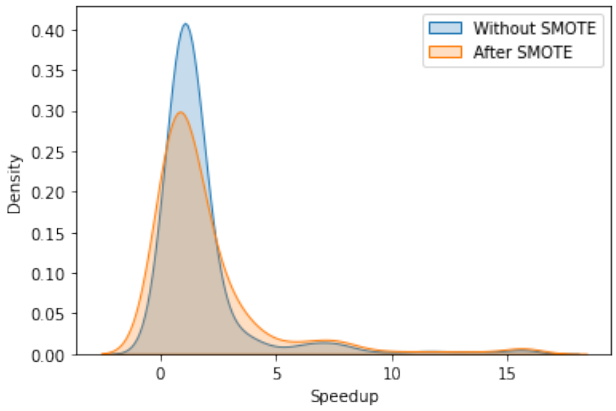

Figure 3: SMOTE vs Without SMOTE

Feature	Value
Architecture	x86_64
Address size	39 bits physical, 49 bits virtual
CPU(s)	4
Thread(s) per core	2
Core(s) per socket	2
BogoMIPS	4800.00
L1d cache	64KiB
L1i cache	64KiB
L2 cache	512KiB
L3 cache	3MiB

Table 2: Specifications of the Machine

4 Experimental Setup

4.1 CPU Specifications

We used a linux system for testing our approach. To get the CPU information, we ran the command *lscpu* and logged the parameters. All the experiments were performed on the machine mentioned in Table 2. Table depicts the configuration of the machine used which are of importance to the methodology.

4.2 Benchmarks

We have used the PARSEC 3.0 Bienia et al. [4] for benchmarking our methodology. This is a publicly available benchmark which has state of the art multithreaded workloads in several types of application domains. We have taken a total of 11 workloads from PARSEC 3.0 for benchmarking our methodology.

4.3 Profiling

We have used the linux profiling tool - *perf* for extraction of features when the program runs. It measures performance counters which count hardware events such as those employed in section 3.1. We have used various machine learning frameworks for managing data skewness (smogn), feature selection, and training algorithms (keras, scikit-learn).

4.4 Dataset

The validity of any learning based approach is based on the training dataset. We had to analyze the available workloads in the benchmark. We have carefully selected various workloads from PARSEC 3.0 which fall under different application domains as shown in Table 1. We used a parsing script to perform profiling using *perf* to get the performance counters during executions of the programs. Additionally, we wrote our own programs in OpenMP to get a diverse dataset as shown in Table 3. The programs that we wrote are under different categories of applications so as to generalize. All the programs were run on 1, 2, 4, 8, 16 number of threads with varying input sizes (simsmall, simmedium, simlarge). The dataset was built by logging the performance counters and some additional parameters which stored size of the input, execution time and speedup of the program relative to number of threads. With this method, we were able to build 316 rows of data with 17 columns without SMOTE. With SMOTE, due to upsampling and downsampling of the data, we build 329 rows and 17 columns. We divided the data into training and testing set in which 80% was put in the training set while the rest 20% was the testing set.

5 Results and Analysis

We managed to create a dataset of 316 samples and 17 columns (each sample representing a feature set of a specific run of some application). Throughout the performance prediction modelling, we had the final **speedup** as the deciding feature and all other columns as supporting features.

We conclude two points from our experiments:

- **Important Program/hardware Features:** According to Figure 5, which is created such

Application	Domain	Program Size (simlarge)
Quick sort	Sorting	16,334 elements
Histogram	Arrays	100,000,000 elements
Merge sort	Sorting	16,334 elements
Linked list processing	Linked List	2048 elements
Counting sort	Sorting	8192 elements
Vector addition	Vectors	16,334 elements
Odd-even transposition sort	Sorting	8192 elements
Matrix multiplication	Matrices	2048 elements
Bubble sort sort	Sorting	8192 elements

Table 3: Experimental Programs

Algorithm	MAE (without SMOTE)	MAE (with SMOTE)
Linear Regression	1.7178	0.9623
Lasso Regression	1.6048	0.9720
Ridge Regression	1.7829	0.8967
Support Vector Regression	7.181	6.6164
Random Forest	1.0350	0.7270
Artificial Neural Network	1.3164	1.3023

Table 4: Comparison of Different Machine Learning Models

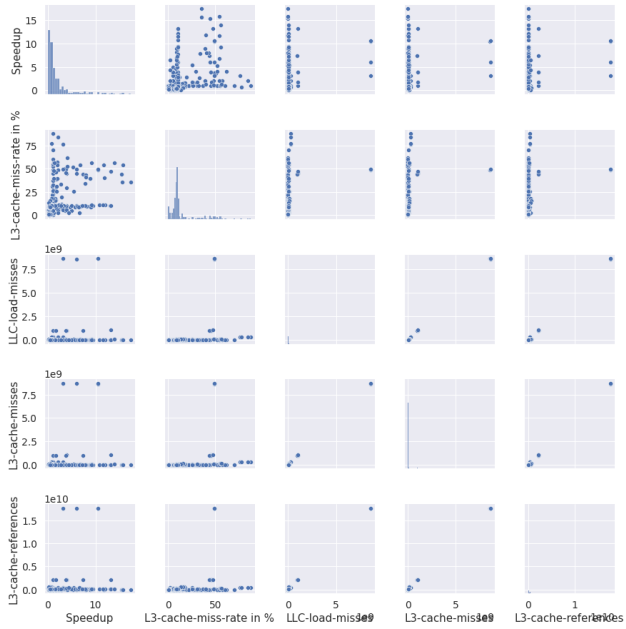

Figure 4: Pairplots of features

that features align in a decreasing order of correlation with *speedup*. From the correlation (magnitude) diagram, we can easily conclude that *speedup* is mostly affected by L3-Cache-Misses/rate/references, L3-load-misses, Number of Threads and CPU cycles/clocks. This data align with our understanding as if there is a high number of cache misses then the *speedup* is very badly affected (i.e. high magnitude of correlation). Also, increase in number of threads would seemingly be improving the *speedup* to some point. We also concur that high number of cpu-cyles mean that CPU is working hard and dealing with a large application thereby affecting speedup.

- **SMOTE - Synthetic Minority Oversampling Technique:** During our experimentation, we employed various machine learning regression algorithms in search for the best performance. Across all the ML models, one thing was common and that helped improve test/validation accuracy which was SMOTE (See Table 4). We employed SMOTE for regression on the input data before fitting machine learning models and eventually found that all algorithms generalize the dataset better than before. SMOTE helps in balancing (Figure 3) our skewed dataset (i.e. high number of samples with speedup=1) and improve the accuracy of the model on unseen minority samples.
- **Best Trained Model:** We compared several regression models like Random Forests, Normal/LASSO/Ridge regression, Support Vector Regression and Artificial Neural Network, out of which Random Forest gives the least Mean Average Error (MAE) of 0.7270 which is a statistical measure of how close the data are to the fitted regression line. Artificial Neural Network is not

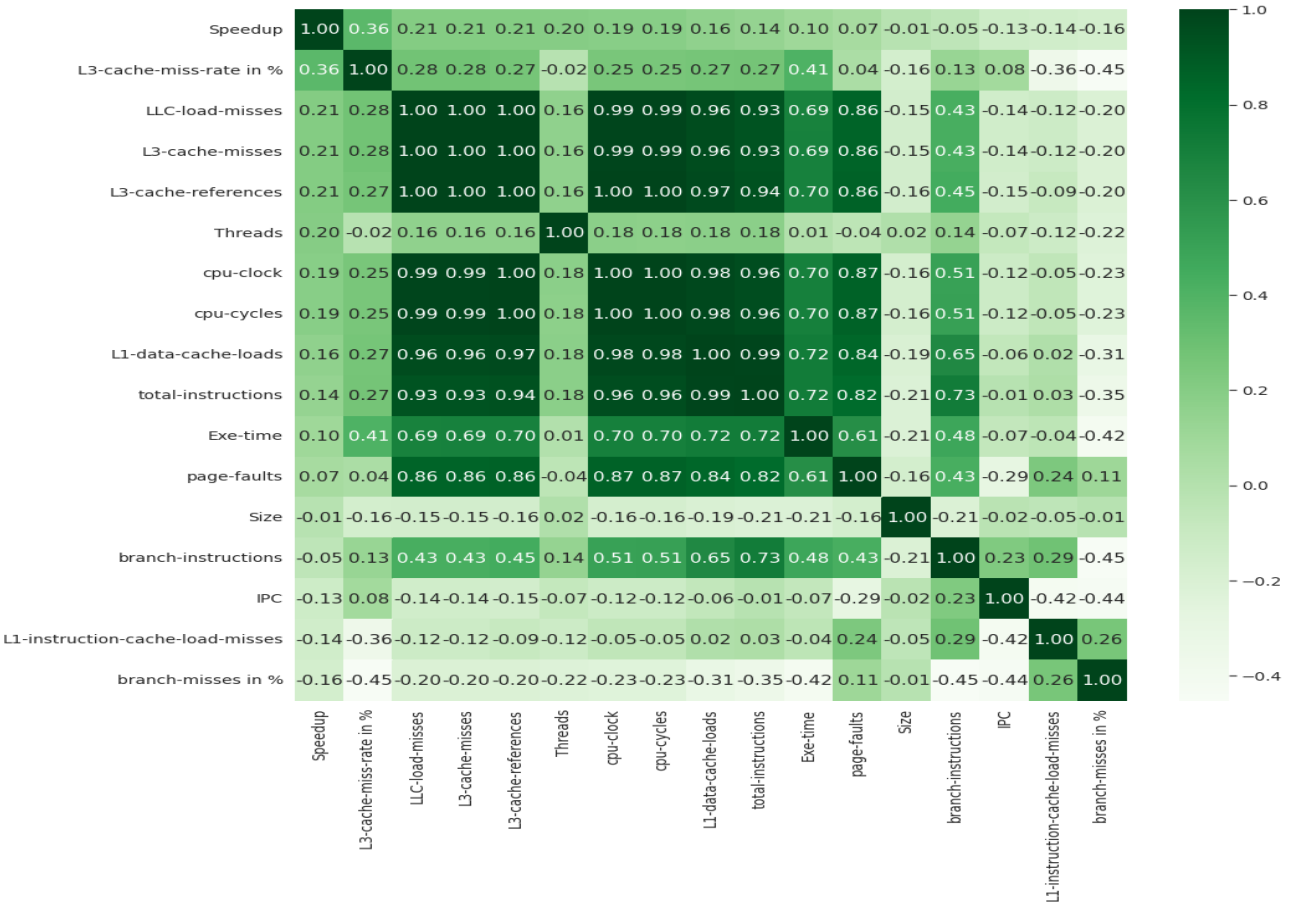

Figure 5: Feature Correlation with Speedup (in Decreasing order)

performing so well because it is probably overfitting on the data as it is dense and requires high number of parameters. Random Forest performs the best maybe because it creates multiple decision trees and then takes the mean of their predictions. Hence, not overfitting on the data while providing good results. Table 4 shows the final results.

6 Conclusions and Future Work

Our paper outlines the major regression algorithms and the artificial neural network methods which are used for prediction of the performance of multi-threaded applications.

- We have refined the existing methodology by removing the data imbalance which can skew the results using the SMOTE technique.

- We also remove extraneous features by analyzing the correlation matrix on the set of available features.

- Our approach is attractive as it provides a fast mechanism for performance prediction without high costs and efforts.

In the future, we plan on gathering more data for training more precise neural network models. We plan on experimenting on self written codes to predict the speedups. As of now, we have experimented with OpenMP programs. We want to increase the scope and include *pthreads* as well. While this approach is exciting, a lack of a more structured database could increase its utility. We plan on reducing model granularity and incorporating other workloads and their features for a generalizable model.

References

[1] Vikram Adve and Rizos Sakellariou. Application representations for multi-paradigm performance modeling of large-scale parallel scientific codes. *INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS*, 14:304–316, 2000.

[2] Nitish Agarwal, Tulsi Jain, and Mohamed Zahran. Performance prediction for multi-threaded applications. In *International Workshop on AI-assisted Design for Architecture*, 2019.

[3] Rajive Bagrodia, Ewa Deeljman, Steven Docy, and Thomas Phan. Performance prediction of large parallel applications using parallel simulations. *SIGPLAN Not.*, 34(8):151–162, may 1999.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[5] Paula Branco, Luís Torgo, and Rita Ribeiro. Smogn: a pre-processing approach for imbalanced regression. 09 2017.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16, Jun 2002.

[7] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 196–205, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[8] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 161–170, 2006.