Performance Prediction of Multithreaded Programs

Aanchal Khandelwal*
New York University
New York, NY
ak8257@nyu.edu

Arnab Arup Gupta*
New York University
New York, NY
ag7654@nyu.edu

Jaimin Dineshbhai Khanderia*
New York University
New York, NY
jk7178@nyu.edu

Mohamed Zahran
New York University
New York, NY
mzahran@cs.nyu.edu

***Abstract*—With rising complexity in modern processors, predicting performance in preliminary stages of hardware and software development in a fast manner accurately has become very important. However, predicting performance of a given software efficiently and precisely is a very challenging problem in itself. With the advancement of multi-processing, predicting performance for such complex architectures becomes even more demanding.**

Accessing the most efficient setup for a large parallel program by executing different setup in real time could be very costly in terms of time and resources, which is often not feasible. We propose an efficient learning based approach to predict the speedup for unknown parallel applications. We use very well-known benchmarks such as PARSEC-3.0 and SPLASH-3.0 for dataset creation which we use to train and test our models. We experiment with various machine learning models and techniques to train the data in different ways so that we can find out the best performing model. We find that ensemble based learning algorithms like gradient boost, decision tree and random forest produce the best results in all the experiments that we carried out to predict the execution time of parallelized applications.

***Index Terms*—multithreading; parallelized applications; PARSEC; SPLASH; performance prediction; speedup; execution time; random forests; linear regression; artificial neural network**

I. Introduction

With multi-core processors becoming the norm, parallel programming is routinely being employed in order to gain an efficient use of hardware resources. However, with increased parallelism, performance gain is not always guaranteed as the gains achieved are susceptible to be overwhelmed by the overheads of creation of threads and communication among processes/threads. [1] Generally, large scale applications take a lot of time to execute and spending as many resources only to result in ineffective speedup is not ideal. Predicting speedup without executing the applications with different configurations, helps in identifying the optimal level of parallelism, and allows us to determine the number of threads required for optimal performance, without having to carry out a theoretical analysis of all algorithms.

In this paper, we utilise different properties of parallelized applications to predict performance of the applications. We do so by collecting various applications from the PARSEC-3.0 [2] and SPLASH-3.0 [3] benchmark suites and executing them with different configurations (number of threads and different problem sizes) to create a rich dataset. After selectively choosing the most important features that affect the performance of parallelized applications, we train the dataset using different machine learning algorithms like *Random Forest, Linear Regression* and *Artificial Neural Networks*. We have compared the accuracy of each of these models and experimented using different configurations of our workflow and observed that ensemble based learning algorithms along with selected significant features using feature selection which is a standard machine learning practice performed better. For the combined model, we achieved an R2 score of 0.964 for the *Gradient Boost* model with the train-test split technique whereas we achieved R2 score of 0.93 for the *Decision Tree* model with the leave-one-out cross validation technique. For the thread wise experiment, we found *Random Forest*t model to be the best performing model using R2 score metric with score ranging from 0.6 to 0.82 for different number of threads.

The rest of the paper has been categorized in the following manner: Section II lays out the related work that has been previously done, Section III displays the proposed idea of the paper, Section IV goes into detailed methodology that we have used for our experiments, Section V details out the experimental setup, Section VI mentions the results and analysis of all the experiments and Section VII states the conclusion.

II. Literature Review

A majority of the work in this sub-domain has been to address the omnipresent problem in benchmarking that is identifying the optimal platform to execute a program which would yield the best performance.

The authors of [4] have chosen to work with multilayer neural networks rather than analytic predictive models, as "analytical models are useful, but often fail to capture subtle interactions between architecture and software". Automatic development of the model resulted in 0.05 to 0.07 errors for them.

The authors in [5] have taken into consideration the similarity in micro-architecture-independent features of applications with programs in benchmark suites to predict performance. They have utilised the concept of distance as a measure of similarity of programs, and identified a number of proxies to be incorporated in predicting performance, achieving an average rank correlation coefficient of 0.83.

The authors in [6] have studied cross-platform prediction and developed a model to predict the performance of a

*Equal contribution.

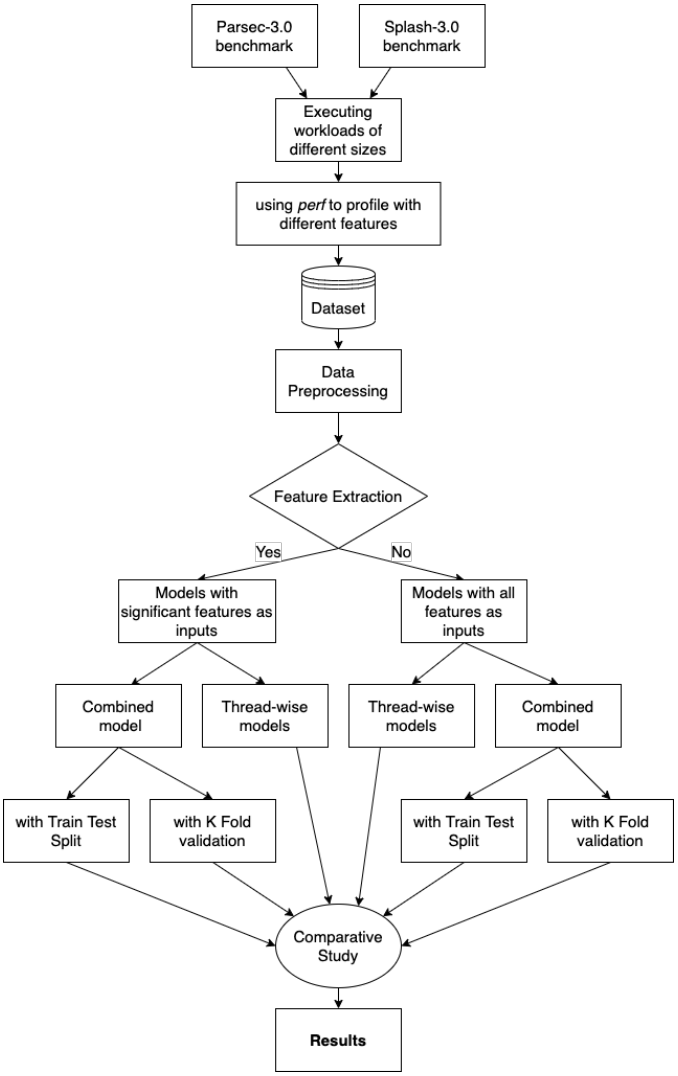

Fig. 1. Workflow of the project

particular algorithm on system Y based on its performance on system X by taking pairs of similar as well as very dissimilar system architectures and instruction set architectures. Their model was trained over a set of 15 benchmark programs and achieved an average accuracy of 0.9.

In [7], instead of determination of the application's characteristics by feature extraction, or static code analysis, the authors ran a few short executions of the applications and collected runtime profile data to match application phases to kernels and output accurate time predictions. Their prediction errors ranged from 0.01 to 0.15.

By evaluating various machine learning models, in terms of how they predict speedup gains in different levels of multi-threading with respect to a single-thread execution, the authors of [1] have proposed an efficient learning based approach. Using different models, for programs from the PARSEC 3.0 and SPLASH 3.0 benchmark suites, they have developed models which achieve a correlation coefficient ranging from 0.67 to 0.82.

We have evaluated the aforementioned works to determine which approach to adopt in our study, resulting in us deciding to conduct a comparative analysis of different methods and combinations of more than one techniques to study their combined behavior and effect on accuracy. Our approach is described in a later section in detail.

III. Proposed Idea

Since speedup is the ratio of elapsed time in serial execution (with a single thread) to its parallel executions (with a range of different number of threads) of the code, we predict execution time of parallelized applications so that speedup can be calculated using the execution time. We do this using a feature set obtained from PARSEC-3.0 and SPLASH-3.0 benchmarks' applications and we obtain the features using performance tools like *perf* which serve as the input to our models. Actual execution time is used as the output in the models that employ supervised learning. We have used a total of 17 applications from these benchmark suites, and each of them have been run for 3 problem sizes and 6 thread configurations, making a 306 points dataset. We run various experiments using different machine learning algorithms to properly understand which model is the most efficient for the prediction of execution time of parallelized applications. Figure 1 illustrates the workflow that we have employed.

IV. Methodology

A. Dataset preparation

All supervised ML tasks rely heavily on the data that they are trained on. It's the most crucial aspect that makes algorithm training possible and has led to popularity of machine learning in recent years. If the model is not presented with good and reliable information, any ML model will behave in the manner garbage in, garbage out. That's why data preparation is such an important step in the machine learning process. In a nutshell, data preparation is a set of procedures that helps make your dataset more suitable for machine learning.

We ran each of the selected workload from the two benchmarks for different input sizes by varying the number of threads ($[1, 2, 4, 8, 16, 32]$) separately using *perf* and captured the required program characteristics averaged over five runs. The details of the dataset and the features that are used for the experiments are explained in detail in the section *Experimental Setup*.

B. Models

Given that we collect and build the dataset for different workloads from the two benchmarks using *perf*, we try out various experiments to predict the execution time for a single data point by using several machine learning models. The aim of these experiments is to try to train a function that will later on help us in predicting the execution time for an unobserved application. In the following paragraph, we discuss the three best performing models based on the results from all the experiments that we conducted:

1) Linear Regression

Linear Regression is a simple regression machine learning technique where it assumes that there is a linear relationship between the input variables and target variable and tries to find that linear function such that it minimizes the sum of the squares of the differences between the target variable and the predicted output variable. As the normal Linear Regression is prone to overfitting, it is generally trained with regularization which is a technique that prevents overfitting by reducing the magnitude of the independent variables by keeping the same number of variables while maintaining the accuracy as well as a generalization of the model. We have used the regularized version of the linear regression namely Lasso regression and Ridge Regression for all the experiments.

2) Gaussian Process Regression

Gaussian Process Regression finds a distribution over the possible functions $f(x)$ that are consistent with the observed data. It is fully specified by its mean function $m(x)$ and its covariance function $k(x, x')$. This is a natural generalization of the Gaussian distribution whose mean and covariance is a vector and matrix, respectively. [8]

Gaussian process regression (GPR) models are non parametric kernel-based probabilistic models. Gaussian process regression is non-parametric (i.e. not limited by a functional form), so rather than calculating the probability distribution of parameters of a specific function, GPR calculates the probability distribution over all admissible functions that fit the data. However, similar to the above, we specify a prior (on the function space), calculate the posterior using the training data, and compute the predictive posterior distribution on our points of interest. Consider the training set $\{(x_i, y_i); i = 1, 2, ..., n\}$, where $x_i \in R_d$ and y_i, drawn from an unknown distribution. A GPR model addresses the question of predicting the value of a response variable y_{new}, given the new input vector x_{new}, and the training data. A linear regression model is of the form $y = x^T\beta + \epsilon$, where $\epsilon \sim N(0, \sigma^2)$. The error variance σ^2 and the coefficients β are estimated from the data. For our case, x is the feature vector and y is the execution time.

3) Random Forest

Ensemble learning is a machine learning paradigm where multiple models (often called "weak learners") are trained to solve the same problem and combined to get better results. The main hypothesis is that when weak models are correctly combined we can obtain more accurate and/or robust models. Bagging is a type of ensemble learning where it tries to decrease the variance in the prediction by generating additional data for training from the dataset using combinations with repetitions to produce multi-sets of the original data. Random forest is a type of bagging algorithm.

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned. Random decision forests correct for decision trees' habit of over-fitting to their training set. So in our random forest, we end up with decision trees that are not only trained on different sets of data (thanks to bagging) but also use different features to make decisions.

C. Experiments

All the experiments are carried out in a supervised manner for all the workloads obtained from both the benchmarks. The objective is to predict the execution time in all the experiments which will eventually help us in calculating speedup for those applications. For this particular reason, we only use the data for threads > 1 in training the models as those are the main data points for which we would like to predict the execution time and thus calculate the speedup. The following is a list of approaches for the models that were tried out in the experiments:

1) Combined Models

In this experiment, we train a single model on the combined set of observations, with 2, 4, 8, 16, and 32 threads input together. We have used this dataset to train nine different ML models to carry out supervised regression to predict execution time, conducting a comparative evaluation to pick the model with the best performance. We have used a simple Test-Train split of the data set, as well as Leave One Out Cross Validation in our experiment. The performance of these models are evaluated using metrics such as the *MAE (Mean Absolute Error)*, and the *R2 score*. We have one model for execution for all sizes of the thread pool. Additionally, we have evaluated a mutlilayer Artificial Neural Network with the Train and Test Split data in our comparison. We have further conducted a study of how each workload from the two benchmark suites in our project (PARSEC-3.0 and SPLASH-3.0) perform individually on the best model chosen and present a graph of our findings.

TABLE I
PARSEC-3.0 APPLICATIONS

Program	Application Domain	Parallelization	Benchmark Suite
Blackscholes	Financial Analysis	Data-parallel	PARSEC-3.0
Bodytrack	Computer Vision	Data-parallel	PARSEC-3.0
Canneal	Engineering	Unstructured	PARSEC-3.0
Facesim	Animation	Data-parallel	PARSEC-3.0
Ferret	Similarity Search	Pipeline	PARSEC-3.0
Fluidanimate	Animation	Data-parallel	PARSEC-3.0
Streamcluster	Data Mining	Data-parallel	PARSEC-3.0
Swaptions	Financial Analysis	Data-parallel	PARSEC-3.0
Vips	Media Processing	Data-parallel	PARSEC-3.0

2) Thread-wise Models

In this experiment, we train different models for different number of threads i.e. we have a separate model for 2,4,8,16 and 32 threads. In order to select a best model for a particular thread size, nine different ML models were trained in a supervised manner using *LOOCV(Leave One Out Cross Validation)* for dataset corresponding to that thread size obtained from all the workloads using the two benchmark suites. We evaluate and compare the different models trained for each thread size based on metrics like *MAE (Mean Absolute Error)*

and *R2 score* and finally choose the best performing model for each thread size. Thus, we will have a different model corresponding to a thread size which can later be used to predict the execution time for an unobserved application which is ran with that number of threads and hence its speedup. We present how the model for a particular thread number works by presenting our findings in form of a graph in the Results section.

V. Experimental Setup

A. Benchmarks

The two benchmarks that we use to assess our implementation are - PARSEC-3.0 [2] and SPLASH-3.0 [3] which are very well-known.

1) PARSEC-3.0

PARSEC is a highly recognized benchmark suite for parallelized applications covering a diverse set of fields including animation, similarity search and data mining. Applications in this suite have been parallelized using p-threads and OpenMP in C but we have carefully selected the ones using p-threads only so that it blends well with SPLASH benchmark. We have used total of 11 applications from PARSEC suite where we executed our workload with three different input sizes and six different thread configurations. Table I displays the programs we have included in our study.

TABLE II
SPLASH-3.0 APPLICATIONS

Program	Application Domain	Benchmark Suite
Radix Sort	Non-comparative Sorting	SPLASH-3.0
Ocean Contiguous	High Performance Computing	SPLASH-3.0
Ocean Non-contiguous	High Performance Computing	SPLASH-3.0
LU Contiguous	High Performance Computing	SPLASH-3.0
LU Non-contiguous	High Performance Computing	SPLASH-3.0
FFT	Signal Processing	SPLASH-3.0

2) SPLASH-3.0

This benchmark is a sanitized version of the decades old Splash-2 benchmark, and has applications parallelized using p-threads library in C. Figure 2 displays the programs we have included in our study. It contains applications in a wide-variety of fields including high performance computing, signal processing and non-comparative sorting. We have used a total of 6 applications from the SPLASH benchmark suite where we executed our workload with three different input sizes and six different thread configurations. Table II displays the programs we have included in our study.

B. Data Collection

For a machine learning model to perform, a rich and useful dataset is one of the most important things. We have put in most of our efforts into making sure we are able to fulfill this. Each application from each benchmark suite is run with 3 different input sizes and with thread combination of 1,2,4,8,16,32. We collect the required metrics of our feature set using *perf* and *time* on a real machine, averaging over 5 runs.

Taking inspiration from [1], we capture 14 features including branch instructions, cache misses, number of instructions and number of cycles using *perf* and we capture the real time in which the application runs using *time*. We use this feature set to predict the speedup of applications. We detail out the dataset that we collected in Table III.

TABLE III
DATASET DESCRIPTION

Feature	Description
application	Name of the workload
threads	Number of threads for which the experiment runs
size	Size of the problem set
branch-misses	Number of branch misses
branch-miss-rate	Number of branch misses in percentage
cache-references	Total number of requests that hit the L3 cache
cpu-clock	Number of CPU clock events in milliseconds
L1-icache-load-misses	Number of L1 instruction cache load misses
cache-misses	Total number of requests that miss in the L3 cache
cache-miss-rate	Number of cache misses in percentage
instructions	Total number of instructions
IPC	Number of instructions per cycle
cycles	Number of cycles
L1-dcache-loads	Number of L1 data cache load misses
LLC-load-misses	Number of last level cache load misses
branch-instructions	Number of branch instructions
page-faults	Number of page faults
real-time	Real execution time of the application
speedup	Ratio of execution of a single thread and multiple threads

C. Models

For the experiments mentioned in the Methodology section above, we tried out various supervised ML models to see which one performs the best for our use case. We trained out the following models: Linear Regression, Lasso Regression, Ridge Regression, Support Vector Regressor (SVR), Decision Tree, Random Forest, Ada Boost, Gradient Boost, Gaussian Process Regression (GPR) and feed forward neural network in train-test split manner as well as in Leave One Out Cross Validation (LOOCV) manner.

Additionally, we conducted a feature extraction step, to further multiply the size of our model set by 2, where one half is trained with all features that the *perf* command returns us, and the other is trained with just the top 6 features as determined by the models themselves. The latter helps us in filtering out unnecessary features which bloat the training process and act as noise, allowing our models to focus on the features that matter, such as: cache-miss-rate, L1-icache-load-misses, cpu-clock, LLC-load-misses, page-faults, instructions.

VI. Results and Analysis

A. Combined Models

In evaluating the performance of different combined models, we look at the *R2 score* and the *Mean Absolute Error (MAE)* as our metrics. In this experiment, as discussed before, we have trained nine different models first using a simple 70-30 train-test split of the data, and then using Leave One Out Cross Validation (LOOCV). The *MAE* and *R2 Scores* from these experiments can be found in Table IV and Table V. We find the Decision Tree model to outperform all the other models with LOOCV, whereas the Gradient Boost model does better

than all others when we use the simple Train Test data split. Although, the Gradient Boost model has a lower MAE for LOOCV, the R2 score is a better measure of the performance of the model because it is a measure of the goodness of the fit.

TABLE IV
R2 SCORE BETWEEN ACTUAL AND PREDICTED EXECUTION TIME IN COMBINED MODELS

Model	Train Test Split	LOOCV
Linear Regression	0.7558	0.6484
Lasso	0.7301	0.6538
Ridge	0.7442	0.649
SVR	0.7504	0.8038
Decision Tree	0.9473	**0.9293**
Random Forest	0.9409	0.9234
Ada Boost	0.8823	0.8652
Gradient Boost	**0.9640**	0.9426
Gaussian Regression	0.6348	0.6489
Artificial NN	0.7866	–

Not surprisingly, all of the models that were trained with only the significant features perform significantly better than their counterparts trained with all features, and this observation is true for both the Train-Test Split experiment and the LOOCV experiment. The reason for this is discussed in the conclusions section of this paper.

Gradient Boosting is an ensemble-based learning method which improves upon weak learners, by sequentially, improving upon previously trained weak learners. Boosting technique have been proven to outperform other models especially when numerical data is in question. They are computationally cheaper since constructing weak models is cheaper. Weak models help the model to learn slowly. This helps avoid overfitting due to small incremental improvements with each model in the ensemble. By wielding the power of cross validation, this allows us to avoid overfitting.

The Decision Tree model is also am ensemble-based learning model the advantages of which have been aforementioned. This model helps reduce variance in the data and performs very well with non-linear datasets. The decision tree model does not require extensive preprocessing and normalization of the input data, and they are not largely affected by outliers.

Figure 3 shows the scatter of points where the x-coordinate is the actual execution time and the y-coordinate is the predicted execution time. The first of these charts is from the output of the Decision Tree model, and the second one is of the Gradient Boost model.

TABLE V
MEAN ABSOLUTE ERROR BETWEEN ACTUAL AND PREDICTED EXECUTION TIME IN THREAD-WISE MODELS

Model	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
Linear Regression	0.7776	0.6996	0.8655	1.0785	0.9466
Lasso	1.2926	1.3465	1.3625	1.4719	1.387
Ridge	0.8196	0.725	0.8629	1.0694	0.9576
SVR	0.8507	0.6842	0.728	0.8727	0.8693
Decision Tree	0.5605	0.5572	0.9367	**0.6732**	0.79
Random Forest	0.5392	**0.4946**	**0.7022**	0.7395	**0.6475**
Ada Boost	0.8213	0.682	0.9234	0.7518	0.8436
Gradient Boost	**0.5158**	0.5069	0.7044	0.6871	0.6491
Gaussian Regression	0.8385	0.7244	0.8681	1.0836	0.9695

B. Thread-wise Models

For the evaluation of thread-wise models, we consider the metrics *Mean Absolute Error (MAE)* and *R2 score*. In order to check how the different models performed in the thread-wise experiment, we have plotted the actual versus the predicted execution time in Figure 2 for all the data points obtained from the benchmark suites. We have actual execution time on x-axis and predicted execution time on y-axis. The datapoints in a particular scatter subplot correspond to the data points for all workloads for that particular thread size. All the subplots have been plotted from the predictions made by the *Random Forest* model trained in Leave One Out Cross Validation (LOOCV) manner. It can be observed from the plots that the *Random Forest* model is able to predict the execution time well for larger number of threads as compared to smaller number of threads. This fact can be attributed to the program characteristics/features for larger number of thread data points as the application becomes scalable and so the features will have corresponding values that will help make the predictions better.

TABLE VI
MEAN ABSOLUTE ERROR BETWEEN ACTUAL AND PREDICTED EXECUTION TIME IN COMBINED MODELS

Model	Train Test Split	LOOCV
Linear Regression	0.9619	0.9361
Lasso	0.9692	0.9539
Ridge	0.9689	0.9446
SVR	0.7365	0.6181
Decision Tree	0.2355	0.2968
Random Forest	0.2730	0.2935
Ada Boost	0.6737	0.7037
Gradient Boost	**0.2128**	**0.2759**
Gaussian Regression	0.9757	0.9520
Artificial NN	0.5563	–

Table VI contains the R2 score between the actual and predicted execution times for different number of threads for all the models that have been trained in the thread-wise experiment. It can be observed that the *Random Forest* model has the best *R2 score* for almost all the threads. The reason for this performance is because Random Forest is an ensemble learning algorithm which combines different learners and improves open their predictions. This finding is also validated by observing the metric Mean Absolute Error (MAE) for Random Forest model for different number of threads.

Table VII contains the Mean Absolute Error (MAE) between the actual and predicted execution times for different number of threads for all the models that have been trained in the thread-wise experiment. It can be observed that the *Random Forest* model has the best *Mean Absolute Error (MAE)* for almost all the threads. The reason for this performance is because Random Forest is an ensemble learning algorithm which combines different learners and improves open their predictions. This finding is in sync with the observation of the metric R2 score for Random Forest model for different number of threads in Table VI.

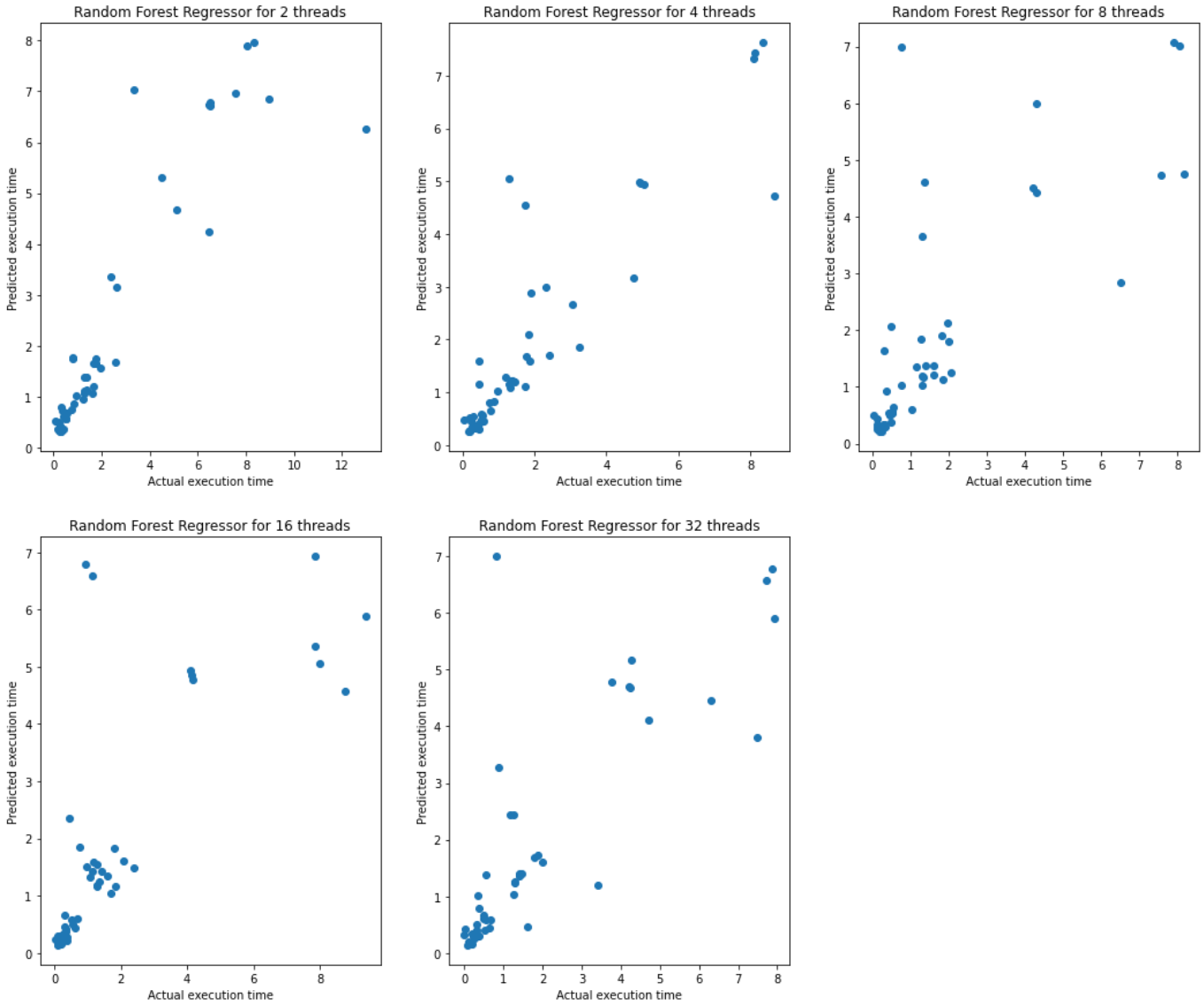

Fig. 2. Scatter plot of the predicted and actual values of execution time in the Random Forest thread-wise models

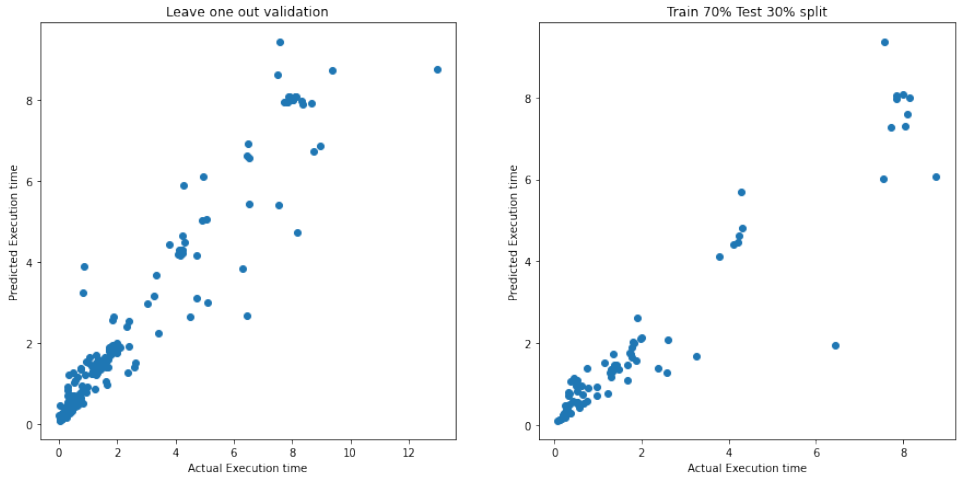

Fig. 3. Scatter plot of the predicted and actual values of execution time in the best combined model

TABLE VII
R2 SCORE BETWEEN ACTUAL AND PREDICTED EXECUTION TIME IN THREAD-WISE MODELS

Model	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
Linear Regression	0.7362	0.7233	0.5941	0.4981	0.5548
Lasso	0.5314	0.3407	0.2403	0.1844	0.3038
Ridge	0.7304	0.7239	**0.6095**	0.5131	0.5728
SVR	0.6505	0.6596	0.5563	0.4032	0.5274
Decision Tree	0.8431	0.6677	0.1759	0.3664	0.4234
Random Forest	0.8241	**0.8168**	0.6001	**0.5969**	**0.6819**
Ada Boost	0.7867	0.7648	0.4391	0.5461	0.5947
Gradient Boost	**0.8438**	0.7877	0.5895	0.5305	0.6367
Gaussian Regression	0.7244	0.7181	0.5984	0.4977	0.5678

VII. CONCLUSIONS AND FUTURE SCOPE

The paper demonstrates how to use the benchmark suites for data collection and preparation for the machine learning based methods to predict the execution time of multi-threaded applications present in the benchmarks so that we can eventually compute the speedup for those multi-threaded applications. Based on all the program characteristics that were present in the dataset, we selected a few subset of those features to perform the training of the various ML models. Feature selection is a common procedure for training the ML models as it leads to better performance as compared to using all the features. We found that the features corresponding to cache misses, branch misses, number of instructions, cpu clock rate play a significant role in determining the execution time of an application. This is supported by the fact that cache misses will result in more time to fetch the data needed for computation whereas total number of instructions and cpu clock rate also determine how long a program will run. Thus, feature selection helps us in improving the model performance for the prediction of execution time.

We have adopted two different approaches in our study, a thread-wise approach and a combined approach. In the former, a set of model have been compared for each size of the thread pool (powers of 2) and the best one has been picked for each thread pool size. This study has allowed us to find and train models which perform the best given a specific number of threads, and the execution time of any unseen programs given a specific number of threads can be best predicted by the chosen model corresponding to those many threads.

The combined model on the other hand, paints an overall picture of the performance of a program for multiple threads. It takes into account the behavior of a program with different input sizes, branching factor, cache misses, number of instructions, threads available and other significant features. This model provides a reliable prediction of the execution time since it has been trained over the entire data set of all executions with all number of threads and problem sizes. It helps us to fine tune our system based on the given algorithm so as to serve all kinds of thread pool sizes.

In the future, one would be in a much more advantageous state were there to be a larger dataset to be leveraged in training our models. Having just over 300 data points in our experiments, we are constrained in the training and testing of our models even with cross-validation. Additionally, exploring neural networks of different configurations and layers would enable one to include a broader spectrum of techniques to comparatively analyse. Given Neural Networks are trending and being used for all kinds of predictive applications, it would serve one well to apply many more NN models in this domain of performance prediction by leveraging the learning capacity of these algorithms.

ACKNOWLEDGMENT

We would like to thank Professor Zahran for his guidance throughout this course and in all of its components.

REFERENCES

[1] T. Jain, N. Agarwal, and M. Zahran, “Performance prediction for multi-threaded applications,” 2019 International Workshop on AI- assisted Design for Architecture (AIDArc), 2019.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pp. 72–81, ACM, 2008.

[3] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, “Splash-3: A properly synchronized benchmark suite for contemporary research,” in Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium On, IEEE, 2016.

[4] Ipek E., de Supinski B.R., Schulz M., McKee S.A. (2005) An Approach to Performance Prediction for Parallel Applications. In: CunhaJ.C., Medeiros P.D. (eds) Euro-Par 2005 Parallel Processing. Euro-Par2005. Lecture Notes in Computer Science, vol 3648. Springer, Berlin,Heidelberg.

[5] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John and K. De Bosschere, ”Performance prediction based on inherent program similarity,” 2006 International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006, pp. 114-122.

[6] X. Zheng, P. Ravikumar, L. K. John and A. Gerstlauer, ”Learning-based analytical cross-platform performance prediction,” 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015, pp. 52-59.

[7] R. Escobar and R. V. Boppana, ”Performance Prediction of Parallel Applications Based on Small-Scale Executions,” 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), 2016, pp. 362-371.

[8] C. E. Rasmussen, ”Gaussian processes in machine learning” in Summer School on Machine Learning, pp. 63–71, Springer, 2003.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, ”The SPLASH-2 programs: characterization and methodological considerations,” Proceedings 22nd Annual International Symposium on Computer Architecture, 1995, pp. 24-36.