Performance Prediction of Multithreaded Applications

Anand Kumar
New York University, NY
Email: ak8288@nyu.edu

Tasbiha Baig
New York University, NY
Email: tfb9946@nyu.edu

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

***Abstract*—In the era of parallel computing, with continuous improvements in the software and hardware capabilities, there has been a stark increase in the complexity of modern processors. As the importance of parallel computing continues to grow with an increase in the usage of multicore processors and GPUs, it has become imperative to understand the degree of parallelism that can be achieved in order to maximize the performance. Thus, a lot of efforts are being put into the study of performance prediction for parallel or multithreaded programs in recent years. In this project, we propose a machine learning based model that predicts the execution time for multithreaded programs on a target platform by leveraging a rich dataset generated by collating execution times for applications provided as workloads by well known benchmark suites such as PARSEC 3.0, SPLASH2x and CRONO. We experiment with different machine learning techniques like Multiple Linear Regression, Xgboost(Gradient Boosted Trees), Support Vector Regression, Ridge Regression and Lasso Regression. We use metrics like R^2 score, RMSE (Root Mean Squared Error) and MAE (Mean Absolute Error) as our metrics to evaluate each model. Through our experimentation we conclude that regularized version of regression techniques such as, Ridge Regression, Xgboost and Lasso Regression give the best results for the dataset generated by the various benchmark suite workloads.**

***Index Terms*—PARSEC3.0; SPLASH2x; CRONO; multicore processors; execution time; parallelism; machine learning; Gradient Boosted**

I. Introduction

Predicting Performance of multi-threaded programs has been in the limelight for quite some time now. With emergence of powerful CPUs and the concepts of multiple cores, parallelism etc, the scenario is becoming more and more complex.Many variables play a vital role in the performance of an application, problem size, parallel fraction of the program, cores, threads, being a few of them. Across the globe, attempts have been made to successfully predict the performance with varying resources and identifying the optimal structure for the best performance.Threads play a crucial role while writing parallel programs. In [8], Ashkan Tousimojarad and Wim Vanderbauwhede have emphasized the fact that, the programmer has to decide not only the tasks but also the optimal number of threads to achieve good performance.

In this paper, we focus on identifying the optimal number of threads on a target machine for the best performance. We extract various performance indicators for parallelized programs from 3 popular benchmark suites, PARSEC3.0, SPLASH2x and CRONO. They provide versatile parallel programs with varying input sets for rigorous simulation. We run the workloads on varying number of threads on the NYU CIMS server. We used Linux $perf$ suite to profile each workload and then select k best features with techniques using Correlation coefficient and removing multi-collinear features.

We deployed various Machine Learning algorithms to predict the performance. We collected both the real execution time and the speedup for each data point and decided to pick the execution time as our target value. This is because, the speedup is always relative to one thread and it's 1.0 for the application with single thread. The models find it hard to figure the relation among the various features and the fixed value of 1.0. On the other hand, picking the execution time gave us promising results. The problem of predicting the performance falls under the domain of regression and so we implemented the following regression techniques for the same, namely, polynomial regression, XGboost (Gradient Boosted Decision Trees), Support Vector Regression, Ridge and Lasso regression. We noted, RMSE (Root Mean Squared Error), R^2 score and MAE (Mean Absolute Error) as our metrics to evaluate each model.

The rest of this paper is organized as follows. Section II presents previous work on the field of performance prediction. Section III presents our proposed methodology. Our experimental setup and results are presented in section IV and V respectively. Conclusions are provided in section V.

II. Literature Survey

A lot of work and efforts have been made in this problem domain, majority being to identify the optimal platform for the best performance. A variety of techniques have been tested and experimented with, ranging from analytical models to Neural Networks.

Predicting performance of an application of interest comes in many flavours. A lot of people have tried to predict the performance of an application of interest on a number of platforms in order to determine which platform yields the best performance [4]. In [4], Hoste et al. proposed microarchitecture-independent characteristics to determine the best platform for the application. Their work is further enhanced by approaches using PCA and Genetic Algorithm. This falls in analytical category and translating the differences in the microarchitecture-independent program characteristics into minute differences in performance is substantially tougher.

In yet another attempt to affordably predict performance across different platforms, Leo T. Yang et al. [9] argued that cross-platform performance translation can be inferred based on relative performance between the two platforms in question.Their approach is novel observation based that relies upon observing a very short partial execution of an application, since, most parallel codes behave in a predictable manner after some period.

Analytic predictive models are difficult to construct and often fail to capture the intricate relations between the underlying hardware architecture and the software. With the increase in the complexity of the systems and the softwares, the feature space scales drastically with large dimensions and it becomes increasingly difficult to ascertain the subtle relations among them and the target variable. Carefully designed neural networks can be especially useful in such scenarios as it is certainly capable of capturing the full system complexity. Engin Ipek et al. in [5] used Neural networks to address this issue and predicted performance on two large-scale parallel platforms within 5% - 7% error across large multi-dimensional parameter space. In contrast to this, we have not included Artificial neural networks in our prediction model because of the sheer lack of data. We have a total of 689 samples (benchmark data) out of which we used 92% for training purpose and remaining 8% for testing. This data is not enough for the Artificial neural network to converge. In the future, we can extend our approach to ANN as well with enough simulated data.

In addition to the models discussed in [8], S. De Pestel et al. [3], proposed an analytical model: RPPM, a mechanistic analytical performance model for multi-threaded applications on multicore hardware.They collected microarchitecture independent characteristics of a multi-threaded workload to predict performance on a previously unseen multicore architecture. They achieved an average prediction error of 11%.

After going through all these papers, we concluded that this problem domain is still in its infancy. A plethora of variables affect the execution of a program, hardware architecture designs, ISAs, memory bandwidth, cache replacement policies, etc. to name a few. The data trend doesn't necessarily remain unchanged over the period of simulation, meaning, the nature of data collected might be heavily affected by some underlying variables which are not recorded by the profiling tools efficiently, for example, the workload on the system (server) by other users etc. This makes it inherently difficult to come up with a model which always works accurately. However still, we collected the data from popular benchmark suites such as PARSEC3.0, SPLASH2x, and CRONO over a shorter duration and applied mentioned regression techniques to predict the performance. In our scenario,Ridge regression worked the best followed by XGboost (Gradient boosted tree) and Lasso regression, with a R^2 score of more than 90%. We describe our methodology in detail in the next section.

III. Proposed Idea

A. Methodology

In order to predict the performance of multi-threaded applications on a single platform, we have stick to the traditional model of regression on numerical values. A regression model is a statistical model, in which we try to find the relationship among various predictors (independent variables) and the target (dependent variable). Our data set has many features as listed by the $perf$ profiling tool in Linux (approximately 39), out of which we identify top k features using correlation coefficient and Principal Component Analysis. Since, we have a large dimensional data set, we might need to use a hyper plane to fit the model to the data. We have used Multiple linear regression as proposed in [6] by Yunus Koloğlu et al. We also tested with other popular regression techniques as listed in table 1.

Index	Regression Algorithm
1	Multiple linear Regression
2	Xgboost (Gradient Boosted Decision Trees)
3	Support Vector Regression
4	Ridge Regression
5	Lasso Regression

Table I: Various regression techniques used

The various steps in our methodology are described in below sub-sections.

B. Data Collection

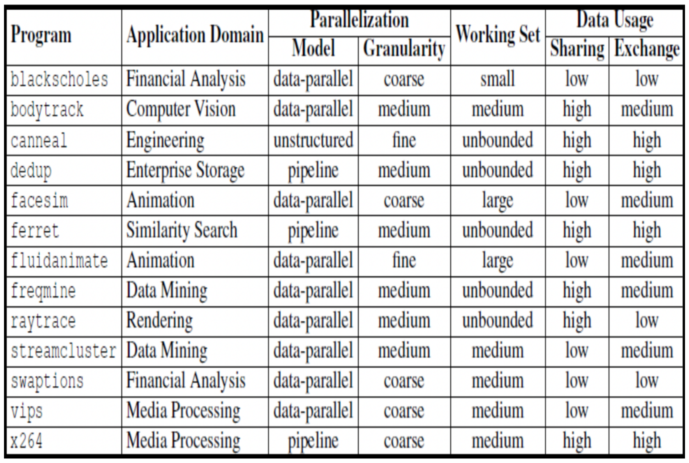
Data is the heart of any machine learning algorithm. The more qualitative data we have the better. We researched extensively for benchmark suites diverse enough to meet our project needs. Merrett et al. in [7] [page 13] has extensively compared various benchmark suites. For collecting data, we used the popular and widely available benchmark suites, PARSEC3.0 , SPLASH2x (integrated with PARSEC3.0) and CRONO. The benchmarks are available as open-source and can be started with a little effort. We selected a total of 23 applications, and wrote a python script to automate the building and running the workloads for different number of threads namely [1,2,4,8,16,32,64,128]. We used Linux $perf$ tool to get a list of available performance counters serving as application/program features (benchmark workloads). We extracted all $Hardware\ event$, $Software\ event$ and $Hardware\ cache\ event$ performance counters for each workload on all the threads in the list. Table 2, shows the top 5 features from each of the events category. However, there is a total of 37 such events. Figure 1, shows the list of available applications on PARSEC3.0 benchmark suite. We included all except $dedup$.

Besides PARSEC3.0 and SPLASH2x (integrated with PARSEC3.0), we also used a popular benchmark suite, CRONO [1]. We explain in detail in the experimental Setup section.

C. Data Preprocessing

We have shown in table 2 the top 5 performance counter (features) from each category, however there is a total of 37 of them. We appended 4 other features to these, problem_size, Number of threads, execution time and speedup. This makes a total of 41 features for each workload simulation.In the data cleaning phase, we identified the features containing zero values and dropped those features from the data set. A lot of these features are unnecessary and doesn't affect the execution time remarkably and in order to make the model simpler, we derived

Figure 1: List of PARSEC workloads
src: https://arco.e.ac.upc.edu/wiki/images/8/8a/Seminar$parsec$3.$pd$



Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

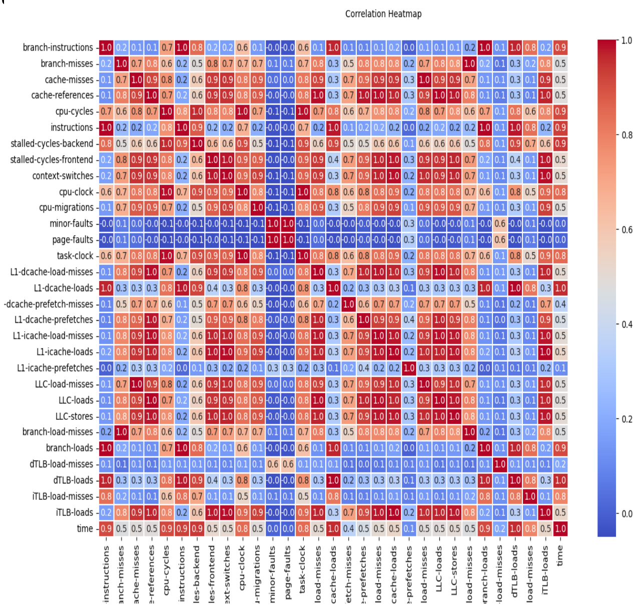
Name	Description
branch-instructions	Number of Branch Instructions
branch-misses	Number of Branch Misses
cache-misses	Number of Cache Misses
cache-references	Number of Cache references
cpu-cycles	Number of CPU cycles
alignment-faults	access to an address that is not aligned for the size of the access
bpf-output	kernel and user-space observability scheme for Linux
context-switches	Number of Context-switches
cpu-clock	high-resolution per-CPU timer
cpu-migrations	number of times the process has migrated to a new CPU
L1-dcache-load-misses	Number of data cache load misses at Level 1 Cache
L1-dcache-loads	Number of data cache loads at Level 1 Cache
L1-dcache-prefetch-misses	Number of data prefetch missed in L1 cache
L1-dcache-prefetches	Number of data prefetches from memory to L1 cache
L1-icache-load-misses	Number of instruction cache misses at level 1 cache

Table II: Various regression techniques used

the correlation coefficient among the features and extracted the top 14 features from the list using sklearn selectKbest library. Fig 2 shows the heatmap of all original features and Fig 3 shows the heatmap of top 14 features after selecting 14 best features.

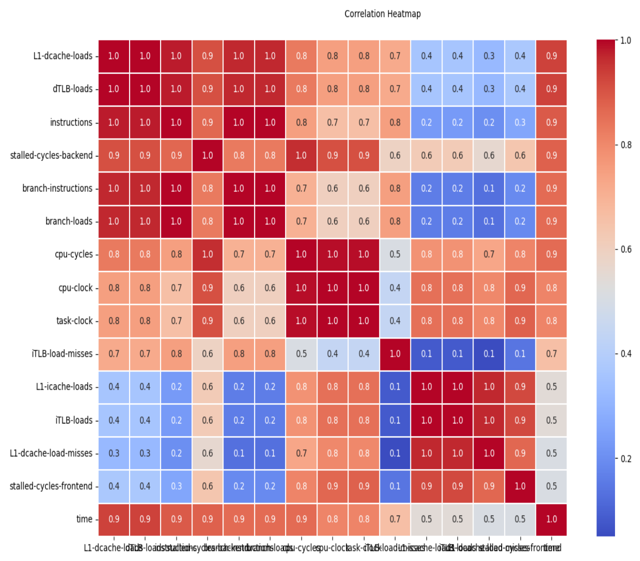
In fig 2, the red shades depict high correlation (positive) and blue shades depict high correlation (negative). We can safely remove all the multi-collinear features from the data set to reduce the dimensionality. For example, stalled-cycles-frontend, context-switches,cache-loads and cache-load-misses are highly correlated (1.0), therefore, we can keep just one of them in our data set and remove rest safely. Also, it makes sense to remove the features that have no correlation with the target (execution time). From the heatmap, we can see there are just 10 features which are highly correlated with the target

Figure 2: Heatmap of all features



but we selected 14 features to make sure capturing the intricate relations that might be affected by some underlying features. The value of k can be experimented with for more diverse results. In Fig 3, we show the heatmap for the top 14 selected features. We can see that we have included 4 features with correlation coefficient 0.5 with the target 'time' in our data set as proposed earlier. Although, we can see that still there are many multi-collinear data but we have included them anyway for our feature data set to be large enough for algorithms.

Figure 3: Heatmap of selected 14 features



D. Data Visualization

Before proceeding with the algorithms, its important to visualize the data set . Many a time, it give subtle hints as

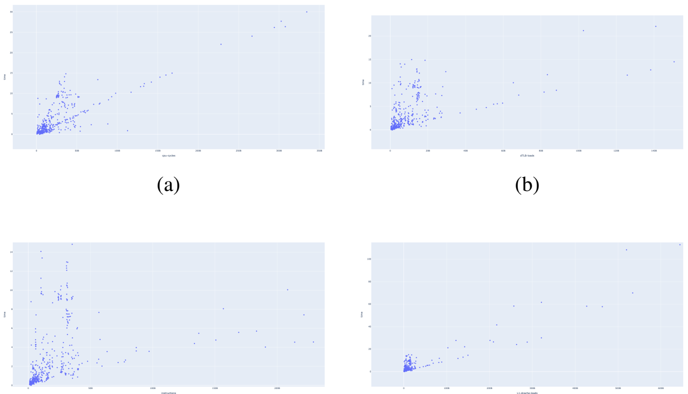


Figure 4: Four selected features correlation with target (Time on Y-axis) (a) Cpu_cycles vs Time , (b) dTLB-loads vs time, (c) Instructions vs Time (d) L1-dcache-loads vs Time

how to proceed with the models design. After selecting top 14 features, we tried to visualize how one feature is scattered in space w.r.t some other feature. We plotted the graphs for some selected features to visualize this relation. For the sole purpose of sample visualization, we have shown below ing fig 4,the relation of cpu_cycles, dTLB-loads, instructions, and L1-dcache-loads with our target 'time' or execution time. From the visualization plots, we can see that most of the data points are clustered together in the bottom region. This makes sense, as in our collected data set, majority of the execution times lie in the range upto 5 seconds. From the Figure 4 (a) cpu-cyles vs Time, we see that as the number of cpu-cycles increases, so does the execution time. Now this holds in real scenarios. We see that upto 50Billion cpu-cycles, the time is fairly upto 10 seconds and it goes as high up to 30 seconds for 350 Billion cpu-cycles. However, there are some outliers which will be taken care of by regression models.

E. Models

e.1) Multiple Linear Regression

Linear regression is a simple statistical model that linearly models the relationship between the target variable (time in our case) and the indicator variables (features). Multiple linear regression differs from the simple linear regression in the number of predicting variables. In the simple version there is usually one single indicator variable and in the multiple it's more than one and hence the name multiple. In our version we have used the normal model without any regularization. We have noted the results and the metric of RMSE, R^2 score, and MAE score for the model's evaluation. We have detailed the results later in the Results section.

e.2) XGBoost

XGBoost is an efficient implementation of gradient boosted trees algorithm. It belongs to the category of supervised learning algorithms in which the model tries to predict the target by combining the estimates of weaker learner models. Tianqi chen et al. in [2] have described the XGBoost in detail for approximate tree learning.It's imperative to note that in XGBoost model, weak learners are regression trees, and each regression tree maps an input data point to one of its leafs that contains a continuous score. It uses $L1$ and $L2$ regularization to minimize the objective function. It's called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models. XGBoost gives remarkable results on our dataset with a very high R^2 score. We detail the results in the Results section.

e.3) Support Vector Regression

Support Vector Regression or SVR is a supervised learning algorithm. It is a regression model that is based on the same principles as Support Vector Machine. Like SVM, the concepts of identifying an optimal hyperplane(decision boundary to classify data points), identifying support vectors(data points closest to the hyperplane) are applicable to SVR as well. Vladimir N. Vapnik in [8] has described the methods to identify an optimal hyperplane in detail. Additionally, in Support Vector Regression we consider two extra lines other than the hyperplane, known as Boundary Lines. Boundary lines create a margin also known as e tube at a specific distance from the hyperplane such that support vectors lie between the hyperplane and the boundary lines. Here e or epsilon is the distance between the boundary line and the hyperplane. Unlike SVM, which is used to predict discrete categorical labels, SVR is used for predicting continuous ordered variables. In SVR we try to fit the error within a certain threshold. Thus, by using SVR we try to predict the best value within a given margin. Here, the e tube corresponds to the margin of tolerance. The SVR model is trained using a symmetric loss function where data points lying outside the margin are penalized whereas the data points lying within the margin having error less than the threshold are accepted without any penalty.

e.4) Ridge Regression

Ridge regression is an extension of linear regression where the loss function is modified by adding a small amount of bias to the cost function. The bias added is known as Ridge Regression penalty. The penalty is calculated by multiplying with the lambda to the squared weight of each individual feature. Ridge regression performs L2 regularization to minimize the cost function. Ridge regression works best for problems where multicollinearity in the data is observed. Multicollinearity occurs when two or more independent variables are highly correlated. Multicollinearity leads to issues while identifying the individual effect of an independent variable on a dependent variable as the high correlation between independent variables can lead to predicting one independent variable from another in a regression model. Adding the ridge regression penalty to the cost function reduces the variance significantly. This modification reduces the complexity of the model. As seen from Fig 3 in section III, the dataset used in our project exhibits multicollinearity. Ridge Regression model outperforms the other models as it is best suited for multicollinear data. We detail the results in the Results section.

e.5) Lasso Regression

Lasso regression is similar to the Ridge Regression. It differs in the way in which the bias/ penalty is calculated. Unlike Ridge regression penalty, in Lasso regression, the penalty is calculated as the absolute weights instead of a square of weights. Lasso regression performs L1 regularization to minimize the cost function. Lasso regression uses a technique known as “shrinkage” where the regression coefficients are shrunk or regularized towards a central value such as zero or mean. Like Ridge regression technique, Lasso regression also works well with datasets where multicollinearity is observed. Lasso regression technique results in a simple model with fewer parameters and also enables feature selection or elimination. For the dataset selected for this project, Lasso regression gives promising results. We detail the results in the Results section.

IV. Experimental Setup

The major part of this project depends upon collecting qualitative real world data to train the machine learning algorithms on. We also needed to find diverse applications which can leverage parallelism with multiple threads. There are quite a few benchmark suites available which already contain well written parallel applications with both pthreads and openmp framework. We selected PARSEC3.0 benchmark suite which contains 13 applications out of which we selected 12 (excluding dedup) and 11 SPLASH2x applications. PARSEC3.0 benchmark is easy to install and begin working with. All we need to do is, download the tar file from the Princeton repository site and extract all the files. All the commands to build and run the workloads are there in the readme section of the repository. We wrote a python script for automating the process of running these workloads and collecting the data in a csv format.

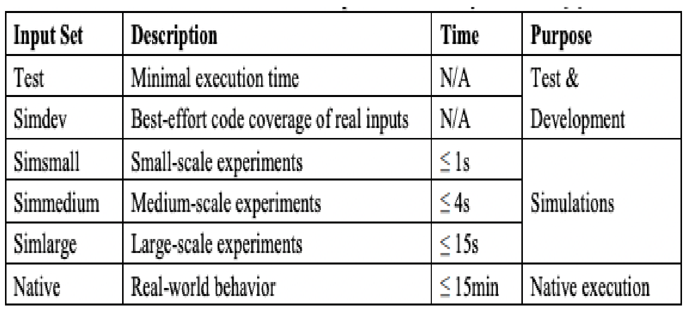
The PARSEC3.0 native workloads have been listed in Figure 1 for reference. In Table 3, we have listed the SPLASH2x workloads that we used for data collection.

Application	Benchmark Suite
splash2x.fft	PARSEC3.0
splash2x.raytrace	PARSEC3.0
splash2x.barnes	PARSEC3.0
splash2x.fmm	PARSEC3.0
splash2x.lu_cb	PARSEC3.0
splash2x.lu_ncb	PARSEC3.0
splash2x.ocean_cp	PARSEC3.0
splash2x.radiosity	PARSEC3.0
splash2x.radix	PARSEC3.0
splash2x.volrend	PARSEC3.0
splash2x.water_spatial	PARSEC3.0

Table III: SPlASH2x Workloads

PARSEC3.0 provides already prepared input sets to work with these workloads. It has a total of 6 input sets to work with ranging from small to large to native real world input sets. These input sets tests the workloads rigorously to make efficient use of parallelism.Figure 5, shows the input sets and their details (taken from Princeton repository for PARSEC3.0).

Figure 5: The six standardized input sets offered by PARSEC src:https://parsec.cs.princeton.edu/doc/memo-splash2x-input.pdf



Input Set	Description	Time	Purpose
Test	Minimal execution time	N/A	Test & Development
Simdev	Best-effort code coverage of real inputs	N/A	
Simsmall	Small-scale experiments	≤ 1s	Simulations
Simmedium	Medium-scale experiments	≤ 4s	
Simlarge	Large-scale experiments	≤ 15s	
Native	Real-world behavior	≤ 15min	Native execution

Out of the 6 input sets offered by PARSEC3.0 benchmark suites, we picked to work on 3 of them - $Simsmall$, $Simmedium$ and $Simlarge$ which take upto 1 second, 4 seconds and 15 seconds of execution time roughly. We didn’t include native input set for technical reasons related to the space issues on the server we have been working with. These input sets worked for both PARSEC3.0 native workloads as well as for SPLASH2x workloads.

Besides PARSEC3.0 we also used CRONO benchmark suite to collect data for our experiments.

Running these workloads was not tough, but the tougher part was to profile these applications and collect profiling data from their simulated execution. There are a few profiling tool already available on Linux kernel such as ”gprof” and ”perf”. ”gprof” didn’t come handy since it didn’t give insights to the hardware level counters so we went ahead with ”perf” suite and it contained a variety of performance counters as shown in Table 2 with their description. We run the workloads with the modified perf command appended in the very beginning to extract values for required (selected) performance counters. We run this command for every workload, for 3 input sizes, and a total of 8 threads, giving a total of 552 data points (23 workloads * 3 input sets * 8 threads), out of which some data points were corrupted and finally we got 545 total data points. This is solely from PARSEC3.0. We got another set of data points from CRONO benchmark suite. Figure 6, shows all available performance counters we extracted from the workload simulation.

CRONO is a benchmark suite composed of multi-threaded graph algorithms for shared memory multicore processors [1]. The requirements for the multicore machine on which the workloads were to be run as specified in the documentation were Linux operating system, g++ 4.6 compiler and pthread Library. These requirements were satisfied by the multicore machines on the CIMS server such as Crunchy1, Crunchy3, etc. and hence running the CRONO benchmark workloads did not require any additional setup. The CRONO benchmark suite provides 10 graph analytic applications out of which 6 graph applications were chosen for this project. The CRONO native workloads used for data collection in this project have been listed in Table IV.

Application	Benchmark Suite
All Pairs Shortest Path	CRONO
Betweenness Centrality	CRONO
Connected Components	CRONO
PageRank	CRONO
Community Detection	CRONO
Single Source Shortest Path	CRONO

Table IV: CRONO Workloads

To build the applications, a common Makefile was provided in the documentation which was executed successfully on CIMS server. For running the workloads a python script was created. The run commands for each workload had the format: executable file P N DEG where, P: Number of threads, N:Number of vertices and DEG: Number of edges per vertex.

Further, we considered 3 problem sizes, details of which are provided in Table V

Problem Size	N	DEG
simSmall	4096	16
simMedium	8192	32
simLarge	16384	64

Table V: CRONO Problem Size

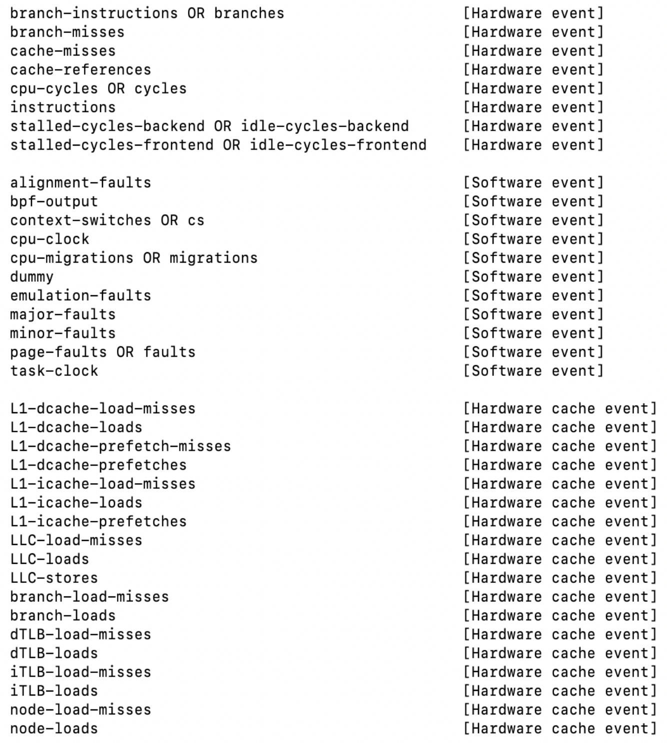
It's worth noting that not all listed performance counters are useful. Many contain a value of zero. We, however, collected all the data and then later during data pre-processing phase we took care of zero values and dropped those from the feature set. We all removed all those performance counters for which the number of unique values across the workload simulation was less than 1%.

Further, after collecting raw data, we run our parser over the output for each workload simulation to extract useful information such as execution time and mapped numerical values to each performance counter in a neat csv database. Later, we run our data pre-processing on this data to remove redundant and useless features, like, correlation coefficient to find multi-collinear features and remove them as explained in Methodology section. We then wrote separate machine learning algorithms over this clean set of data. We also normalized the features to make them Normal distribution, with mean 0 and standard deviation 1, before applying the algorithms.

V. Results & Analysis

After data collection, data pre-processing we applied various machine learning algorithms as described above on the final data set. We split the data set into train and test sets with a train size of 92%.After training for each algorithm, we tested the model on the test set and gathered the results. We observed three metrics for each model, $RMSE$ (Root Mean Squared Error) , R^2 score and MAE (Mean Absolute Error) Table IV

Figure 6: Available performance counters (extracted features from workload simulation)



```
branch-instructions OR branches                    [Hardware event]
branch-misses                                      [Hardware event]
cache-misses                                       [Hardware event]
cache-references                                   [Hardware event]
cpu-cycles OR cycles                               [Hardware event]
instructions                                       [Hardware event]
stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]

alignment-faults                                   [Software event]
bpf-output                                         [Software event]
context-switches OR cs                             [Software event]
cpu-clock                                          [Software event]
cpu-migrations OR migrations                       [Software event]
dummy                                              [Software event]
emulation-faults                                   [Software event]
major-faults                                       [Software event]
minor-faults                                       [Software event]
page-faults OR faults                              [Software event]
task-clock                                         [Software event]

L1-dcache-load-misses                              [Hardware cache event]
L1-dcache-loads                                    [Hardware cache event]
L1-dcache-prefetch-misses                          [Hardware cache event]
L1-dcache-prefetches                               [Hardware cache event]
L1-icache-load-misses                              [Hardware cache event]
L1-icache-loads                                    [Hardware cache event]
L1-icache-prefetches                               [Hardware cache event]
LLC-load-misses                                    [Hardware cache event]
LLC-loads                                          [Hardware cache event]
LLC-stores                                         [Hardware cache event]
branch-load-misses                                 [Hardware cache event]
branch-loads                                       [Hardware cache event]
dTLB-load-misses                                   [Hardware cache event]
dTLB-loads                                         [Hardware cache event]
iTLB-load-misses                                   [Hardware cache event]
iTLB-loads                                         [Hardware cache event]
node-load-misses                                   [Hardware cache event]
node-loads                                         [Hardware cache event]
```

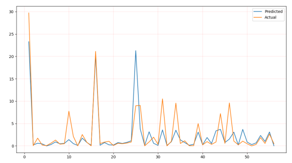
shows the model evaluation results on the test set. It is noted that for XGBoost, ridge and lasso we also used k-fold cross validation to improve the results and observed that prediction works significantly better with k-fold cross validation.

Model	RMSE	R^2 score	MAE
Multiple Linear Regression	8.2375	0.7040	1.4651
Support Vector Regression	3.2166	0.9063	1.6853
XGBoost	5.0186	0.9483	1.2136
Ridge Regression	1.5521	0.9610	1.0318
Lasso Regression	3.8045	0.9298	1.9412

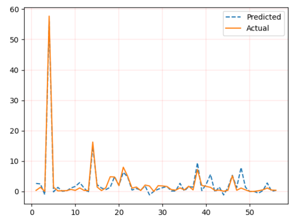
Table VI: Models Evaluation Metric

From Table IV, we see that in our scenario, Ridge Regression has performed the best followed by XGBoost and Lasso regression. In the Multiple linear regression model, the R^2 score is the least and it's RMSE is the greatest among all. Conversely, for Ridge regression, it has the highest R^2 score capped at around 96% with the least RMSE error at 1.5. Generally speaking, R^2 score, pronounced as R squared score is the best metric among above. It is also said to be the coefficient of determination and is the proportion of the variation in the dependent variable that is predictable from the independent variable(s) [source : wiki].

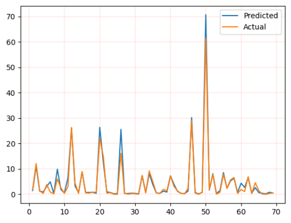
We have submitted separate models (python code) for each regression algorithm that individually trains the model on the data set and then predicts on the test set. We show their prediction for each model in below graphs in figure 7.



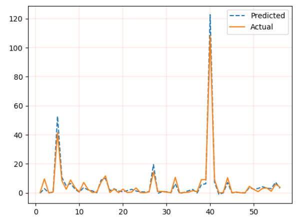
(a) Multiple Linear Regression



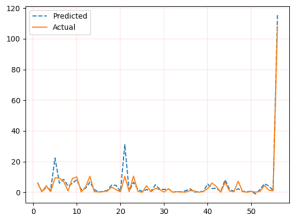
(b) Support Vector Regression



(c) XGBoost



(d) Ridge



(e) Lasso

Figure 7: Actual Vs Predicted graphs

In figure 7, we see the actual vs predicted graphs for all 5 models. The blue line represents the predicted value and the orange line is the actual execution time. The prediction is made on a total of around 70 data points. From the graphs, we observe that the Ridge matches the actual values with a great approximation. XGBoost (Gradient Boosted Decision Trees) also made nice prediction values with k-fold cross validation. Multiple Linear Regression predictions don't follow a good prediction trend on the whole.

On an average, we have observed that all these models do a better prediction on larger execution time with a relatively smaller margin error rate of around 15% on average. For smaller values, predictions are better for Ridge, lasso and XGBoost, although they have better predictions for larger run time as well.

VI. Conclusion

In this paper we have presented common regression machine learning algorithms to predict execution time of various workloads with varying number of threads. We have used a single platform as our target and the purpose of this paper is to give an idea of predictive speedup gained with varying number of threads. This should give us an optimal number of threads that should help us achieve the best performance on a target platform.

However, we have used just a single platform in our experiments, it helps to note that this method can indeed be extended to use over other target platforms. All we need to do is collect data for the target platform and re-train the models. People across the globe have been trying to accurately predict performance on cross-platforms and have already achieved remarkable results.

We observed that, there are a lot of features that can be extremely helpful in achieving our purpose and many features just correlate with other features. We have worked with a very large dimension, which we later tried to reduce with the help of correlation coefficient technique. We observed that some features such as number of instructions, cpu-clock speed, cpu-cycles greatly affect the execution time. However, we assumed beforehand that cache-misses and cache-references must be greatly affecting the target as well but surprisingly, while data pre-processing phase we found their correlation value with the target to be very low and had to remove them from our data set. We verified this manually with the data set and found that for majority of the data collected this value was zero. This might be due to some underlying errors while profiling the workloads. Therefore, in the future, it would be more useful to select a more robust profiler to collect data and generate more robust models.

Moreover, we see that in our scenario, the regularized regression algorithms such as Ridge and lasso work better than un-regularized versions such as multiple linear regression. Gradient Boosted Decision trees work remarkably in predicting the performance with the given data.

One major potential candidate, the Artificial Neural Network has been devoid from our experiments. The sole reason for that is the lack of enough data points. In the future, we hope to come up with enough data points, in range of thousands, for the neural networks to properly train with.

VII. Acknowledgement

We would like to thank Professor Zahran for this project opportunity and his guidance. We have taken references from tutorial sites for implementing the machine learning algorithms such as sklearn repository and other useful resources.

References

[1] Masab Ahmad et al. "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 44–55.

[2] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[3] Sander De Pestel et al. "RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors". In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 257–267. DOI: 10.1109/ISPASS.2019.00038.

[4] Kenneth Hoste et al. “Performance prediction based on inherent program similarity”. In: *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2006, pp. 114–122.

[5] Engin Ipek et al. “An approach to performance prediction for parallel applications”. In: *European Conference on Parallel Processing*. Springer. 2005, pp. 196–205.

[6] Yunus Kologlu et al. “A multiple linear regression approach for estimating the market value of football players in forward position”. In: *arXiv preprint arXiv:1807.01104* (2018).

[7] Geoff V Merrett et al. “PRiME: Applications and Benchmarks”. In: ().

[8] Ashkan Tousimojarad and Wim Vanderbauwhede. “Number of Tasks, not Threads, is Key”. In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015, pp. 128–136. DOI: 10.1109/PDP.2015.81.

[9] L.T. Yang, Xiaosong Ma, and F. Mueller. “Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution”. In: *SC ’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 2005, pp. 40–40. DOI: 10.1109/SC.2005.20.