Performance Prediction of Multithreaded Applications

Nancy Nigam
Department of Computer Science
New York University
New York, NY, USA
nn2163@nyu.edu

Astha Gupta
Department of Computer Science
New York University
New York, NY, USA
ag7982@nyu.edu

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

***Abstract*—Multiple-core processors have been widely available and adopted as these designs smartly addressed the problems of single cores hitting the ceiling of their physical limitations and solved it by moving extra cores on a single processor chip. This helped in running numerous processes at the same time with greater ease and increasing application performance while multitasking. However, all these perks do not come for free. Numerous overheads are associated in order to achieve increased parallelism such as inter-process communication, dependency, resource sharing, scheduling, level of parallelism and synchronization. This shows that simply increasing threads/cores won't always result in better performance while scaling applications and hence we need new means and methodology to effectively and accurately predict the performance of applications. Looking at the complexity and scale at which the software industry is growing, performance prediction of parallel programs has become topic of importance in the domain. One way to predict the perfect combination of hardware resources for a particular application is to run it on different machines with varying parameters and find the best performing set. This however is an unreasonable use of time and resources. In this paper, we try to estimate the performance of numerous state-of-the-art applications by taking the advantage of various machine learning models. We are fixing the underlying hardware and varying different levels of parallelism to accurately predict the speedups for performance on different number of threads relative to single threaded execution. Our evaluation is based on benchmark suit PARSEC 3.0 for applications parallelized using p-threads in C. We use various different machine learning algorithms to predict the speedup of an application. The best algorithm is the Random Forest model which gives us an error of 0.40. We conclude that instruction count, number of page faults, branch misses, CPU migrations and the number of threads affect the performance prediction the most, as analysed using the three best models used.**

I. Introduction

Since the 1960's the semiconductor industry has diligently followed the norms given by Gordon Moore and Robert Dennard. As over the previous five decades, the trend of increasing computation speed could be attributed to faster transistors (thanks to Dennard Scaling) but that is not the case anymore. In this modern era of technological growth, It is commonly recognized that following the simple scaling rules described by Dennard and his team back in 1974 is now no longer a sufficient strategy to meet future transistor density, performance, and power requirements. Nevertheless, thanks to advent of multicore processors, complex tasks which deemed impossible decades ago seems reasonable today. With horizontal scaling and clusters of nodes working together, we are in a better position to solve highly complex problems without compromising on the performance. In spite of the many advantages that multi-core processors come with, there are a few major challenges the technology is facing. One main issue seen is with regard to software programs which run slower on multi-core processors when compared to single core processors. It has been correctly pointed out that "Applications on multi-core systems don't get faster automatically as cores are increased" [1]. Programmers must write applications that exploit the increasing number of processors in a multi-core environment without stretching the time needed to develop software [2].

Moreover, we need to carefully consider the efficient level of parallelism for the application at hand which gives best performance gains for the current hardware architecture. As we have mentioned above, it is not feasible to run the application on different machines with varying thread counts to determine the optimal setup. Hence we are trying to build a system which can help in estimating the performance of the application on a specific hardware for different thread counts by extracting features from single thread execution only.Our goal here is to determine which parallelization setup (here number of threads) will give optimal performance for a particular application without physically running it for different number of threads. We plan to use machine learning models which will extract the application features which have direct impact on parallelizability, speedup, and performance from single thread execution and based on these features, we estimate the speedup that can be achieved by this application for different thread counts. Being able to predict performances of applications in early stages of development can be a beneficial in achieving high performance standards.

In this paper, we propose a learning-based approach accurately predicting the performance of a workload on a target platform from various performance statistics obtained directly on a host platform. Our proposed framework is as follows :

- ***Performance Benchmark*** : PARSEC 3.0 is used as the performance benchmark suite for running various state-of-the-art multithreaded application on emerging workload. Our focus is on applications parallelized using p-

threads in C.

- ***Profiling tool*** : We have leveraged the Perf tool in linux for performance analysis to gather profiling statistics [12]
- ***Machine Learning Models*** : We have used various machine learning (ML) algorithms like Random Forests, Linear Regression and Gaussian Process Regression to find the best fit analytical model for performance prediction
- ***Feature Selection*** : Using the ML models we are able to identify the features which have direct impact on the performance

The remainder of the paper is organized as follows: Section II (Literature Survey) surveys related work in this area which paved the way for selection of certain models and profiling tools and benchmarks. Section III discusses the proposed idea for program performance prediction using machine learning algorithms and Section IV discusses the experiment setup in brief, machine configuration, application tools used to extract features in order to construct the training and testing data sets. This is followed by evaluation and discussion of our experiments, empirical results in Section V. Finally, Section VI concludes with a conclusion of this work.

II. Literature Survey

The problem of performance prediction has seen extensive research in the last few years. Flores-Contreras et al[3] perform an extensive survey analyzing the various use cases wherein performance prediction is performed for parallel programs. The set of methods used for prediction include manual analytic methods that use mathematical equations for performance prediction. However, a drawback of such a technique is that it requires a very deep level of understanding of the application as well as the hardware architecture. In our study, we aim to predict the speedup of an application using regression-based analytical tools and non-analytical machine learning algorithms.

In order to build a prediction system, we first need to identify the features which will be required for accurate prediction of an application's speedup. Both [4] and [5], show the importance of features like cache misses, branches, instruction cycles and number of instructions on the performance of the application.In [4], Zheng et. al develop an analytic model that predicts the cross-platform performance of a program. They show the importance of features like cache misses, branches, number of instructions on the performance of the application. Our goal is different from their work, however, these features are of equal importance in our study. The predictive features used are extracted with the help of tools like PARSEC[11] and Perf [12].

Recently, many studies have used machine learning and deep learning based models for predictive analysis. The choice of algorithms include Linear Regression, Ridge Regression [6] Gaussian Processes [5], [7] and Neural Networks ([9],[8]). Even though neural networks promise high performance, they often require a large amount of data. Furthermore, [10] has shown that even with a large data set boosting-based machine learning algorithms still outperform them.

III. Proposed Idea

Our model takes application features extracted from profiling tools as the input. Selection and extraction of these features is discussed in the following section in detail. The output obtained from the model are the speedups for different number of threads for the application of interest.

A. Feature Selection and Extraction

We extracted the features using profiling tool Perf and single-threaded execution statistics. The extracted features are categorized into following categories :

Application size - Applications with larger input size will benefit more from parallelization as they'll be able to overcome the communication and thread management overheads when compared to smaller input size. Features selected to validate this category are *ipc* (instructions per clock cycle), *instructions* (number of instructions) , *cycles*.

Cache - Execution time in many programs is dominated by memory access time, not compute time. This is becoming increasingly true with higher instruction-level parallelism. Cache hit rates at L1 and cache miss rate at LLC have a direct impact on the performance. In a cache hierarchy, the last-level cache miss is 100x or more expensive than a first-level cache hit. This happens because now the data has to be fetched from memory which is very expensive. It also incurs the overheads associated with cache coherency. Features selected to validate this category are *cache-miss-rate, cache-misses, cache-references, L1-dcache-loads, L1-dcache-load-misses, L1-icache-loads, L1-icache-load-misses, LLC-loads, LLC-load-misses*

Branch Instructions - Branch instruction in a program can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order. Conditional branches can cause "stalls" in which the pipeline has to be restarted on a different part of the program thereby degrading the performance. Hence branch predictors play a critical role in the performance of modern processors, and the prediction accuracy is known to be one of the most important attribute of such predictors.[16] Features selected to validate this category are *branch-instructions, branch-miss-rate, branch-misses, branch-load-misses*

CPU-Migrations - Migration is when a thread, get scheduled on a different CPU than it was scheduled before. This can happen due to several reasons like calling exec(), fork(), thread wake-up event or to balance out the workload. Resuming execution on another core adds a lot of overheads and has direct impact on the performance. Feature selected to validate this category is *cpu-migration*.

Page Faults - When we start using a memory page that has not yet been mapped to a physical location, we get a page fault that is handled by the kernel. Increase in Page faults generally degrades performance and can cause thrashing

which is an expensive operation. Feature selected to validate this category is *page-fault*.

Context Switches - Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. Context-switch time is pure overhead, because the system does no useful work while switching. Hence it attributes to performance loss. Feature selected to validate this category is *context-switches*.

After identifying the features with maximum impact on the performance, we used PARSEC 3.0 Benchmark Suite to run our experiments [11]. We operated on different workloads provided by the suite namely *simsmall, simmedium, simlarge*]. Hardware configuration of current system is listed in Table I. Parallel Applications used for training and evaluation are listed in Table II.

Correlation study is an important step of data preprocessing in any machine learning or statistical task. Features that are highly correlated are dependent on each other. This implies the presence of superfluous features present in the set of input variables. For some algorithms like Linear Regression, this can lead to a negative impact on performance. So, for our model, we perform a correlation test between all the input features and remove one set of features which have a correlation greater than 0.95 to any other features.

B. Model

We make use of a number of popular machine learning algorithms. We decide against the use of Artificial Neural Networks as our data set contains less than a 1000 data points, which would make use of neural networks an overkill. Furthermore, gradient boosting based machine learning algorithms have shown to outperform neural network-based models

The following set of algorithms give us better performance compared to the others used:

Regression-based algorithms: This is the most fundamental technique used to predict a target variable using the input. The algorithms assumes a linear relationship between the speedup and the input features. Ridge regression is a form of linear regression with added regularization.

Decision trees: They predict a regression variable in a supervised manner using a tree-like structure of choices and all the possible outcomes of those decisions and choices. At each successive step, a feature is chosen and decisions are made based on a split of the value of that feature.

Random Forests: Random forests [14], Gradient Boost [13] and XGBoost [15] are enhanced ensemble algorithms that use multiple decision trees and bagging and boosting techniques on top to make predictions. Random forests use multiple different decision trees and predict outcomes based on the majority vote of all the trees. Gradient boosting algorithm uses an iterative approach wherein consecutive trees learn from the mistakes of the previous trees and a set of these weak trees are used to develop the final model. XGBoost (extreme gradient boosting) and AdaBoost (adaptive gradient boosting) are two enhancements of gradient boosting techniques.

IV. Experimental Setup and Experiments

A. Hardware Setup

Hardware configuration of current system used for simulation is listed in Table I.

TABLE I
MACHINE CONFIGURATION (LSCPU)

Spec	Value
Vendor ID	AuthenticAMD
Byte Order	Little Endian
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
CPU Clock Speed (MHz)	2099.983
CPU(s)	64
Thread(s) per core	2
L1d cache	16K
L1i cache	64K
L2 cache	2048K
L3 cache	6144K

B. Benchmark

Program execution time is the only accurate way to measure performance. Without a program selection that provides a representative snapshot of the target application space, performance results can be misleading and no valid conclusions may be drawn from an experiment outcome. The PARSEC 3.0 benchmark[11] features state-of-the art, computationally intensive algorithms and diverse workloads from different areas of computing. We finalized 11 applications which are mention in (Table II) and ran each of them on *simsmall*, *simmedium* and *simlarge* [19] input sets for our experiments. Number of threads ranged from *1 to 32* with a step size of 2. Reason for this selection was to gather enough data points.

TABLE II
PARALLEL APPLICATION USED FOR EXPERIMENTS

Parallel Application	Domain
blackscholes	Financial Analysis
bodytrack	Computer Vision
canneal	Engineering
facesim	Animation
ferret	Similarity Search
fluidanimate	Animation
raytrace	Visualization
streamcluster	Data Mining
swaptions	Financial Analysis
vips	Media Processing
x264	Media Processing

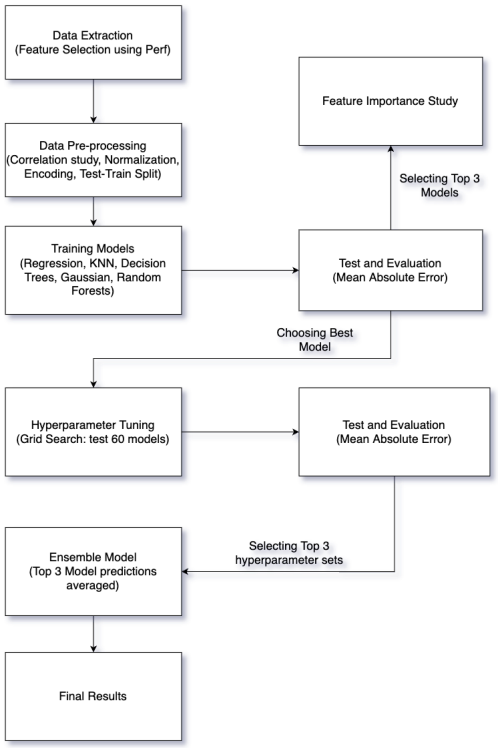

Fig. 1. A high-level overview of the prediction pipeline

C. Tools and Scripts

We used the Linux Perf tool to collect the performance counters data or the features[10]. We used the the *perf* tool owing to its seamless integration, rich set of available metrics, and a myriad of flags to tune simulation with ease. Each simulation is run for 5 times using the perf *r* flag, and average is taken to reduce fluctuations and anomalous data points. We automated the process of feature generation and data creation using python. The script generated a csv file (comma separated) which was given as input to the models.

D. Training and Prediction Setup

Before we begin model training, it is important to perform some pre-processing and data cleaning steps. For all steps from hereon, we use Python and popular machine learning and mathematical libraries like Sklearn, Pandas, Seaborn, Matpltlib and Numpy.

As the first pre-processing step, we analyze our data and perform the aforementioned correlation study to drop redundant features as shown in Fig. 2. As a data cleaning step, we remove any outliers from the data. For the purpose of our study, we define as outlier as any data record which has speedup greater than 100. Since, the maximum number of threads in our dataset is only 32, a speedup values of greater than hundred are characterized as anomalies.

Further, we convert any categorical feature into an ordinal type, as most machine learning models connote work with

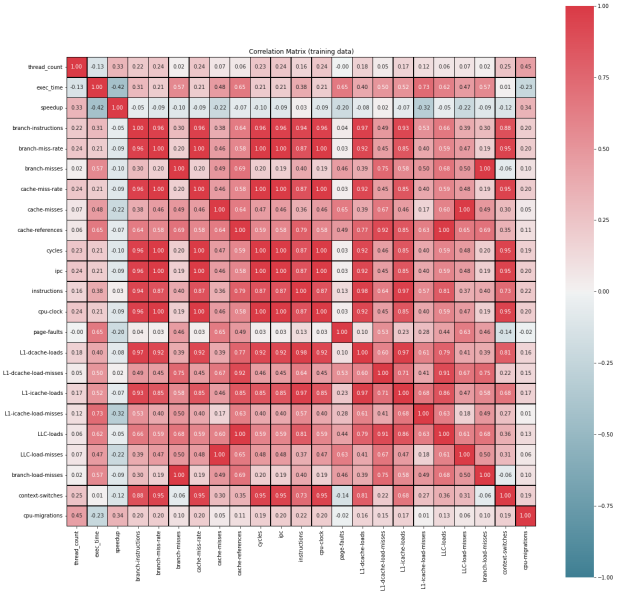

Fig. 2. Correlation between input features

string labels. In our data, the only categorical feature was application size, and we assigned 1,2,3 to the three different categorical labels. Finally, we perform data normalization to the input variables. This ensures that all features are scaled down to values between 0 and 1. We use the Min-max Scaler in Sklearn which performs as following:

$$X\_std = (X - X.min)/(X.max - X.min) \quad (1)$$

$$X\_scaled = X\_std * (max - min) + min \quad (2)$$

Thus, our final data set consists of 464 data points. The number of threads present in our data set are from 1 to 32, in a multiples of 2. The number of applications explored are 11.Finally, we split the data set into train and test sets in a ratio of 80 to 20. Thus, there are 371 data points in the train set and 93 data points in the test set. For the training part, we use a number of machine learning models provided by the Sklearn library. The models trained include Linear Regression,Gaussian Process Regression [5], Random Forest, Ridge Regression, Ada Boost [18], KNN [17], Decision Tree, Gradient Boosting, XGBoost and .

The performance of each model is evaluated on the test set using the Mean Absolute Error (MAE) metric which is calculated between the test set points and predictions. MAE is the absolute differences between the actual values and the predicted values, averaged over the total number of data points, and is calculated as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}(x_i - x_{pred}) \quad (3)$$

Here, n is the number of data points, x_i is predicted value and x_pred is ground truth value for the data point.

For each model, we display a graph showcasing the predicted values and the ground truth values. Since the Random Forest model is the best performing model, we perform Sklearn's Grid Search hyper-parameter optimization to determine the best values of the parameters for the model. The hyper-parameters tuned are the number of trees explored in the random forest model, the minimum number of samples required to split a given node and the minimum number of samples required at each leaf node of the decision trees.

We traverse over the following values of these hyper-parameters.

- num_estimators = (100,200,300,500,1000)
- min_samples_split = (2, 5, 10)
- min_samples_leaf = (1, 2, 5,10)

The grid search thus tests a set of 60 different random forest models and ranks them based on their mean absolute error values.

Finally, we train a final ensemble model as our final model. Using the results of the grid search, we identify the top three performing sets of hyper parameters, and use three random forest models with those hyper parameters. The predicted values of the three models are collected and their respective average values are used as the final predicted values. The MAE is then calculated over these final predictions. Our ensemble model performs the best among all the individual models.

V. Result and Analysis

Table III displays the performance of all the models used. We see that the top three performing models among the base models are the Random Forest model, XGBoost model and the Gradient Boosting model. Random Forest is the best performing model with an MAE of 0.4117.

TABLE III
PERFORMANCE COMPARISON OF ALL MODELS

Model Name	**MAE**
Gaussian Process Regression	1.8566
Ridge Regression	1.7604
Linear Regression	1.7311
Ada Boost	1.3828
KNN	0.7930
Decision Tree	0.6830
Gradient Boosting	0.6463
XGBoost	0.6204
Random Forest	0.4117
Random Forest Best Grid Search	0.4115
Ensemble	**0.4010**

The results of our grid search are displayed in Table III. The best model had the number of trees as 1000, minimum number of samples required to split a node was 2 and the minimum number of samples at the leaf node was found to be 1. The best performing model had a performance better than the base Random Forest model, with a performance improvement of 4.38. Finally, the ensemble model gives us the best performance with an MAE of 0.40.

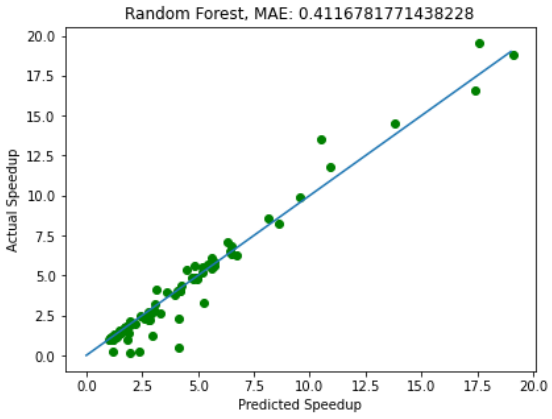

Fig. 3. Actual vs Predicted Speedup base Gradient Boosting Model

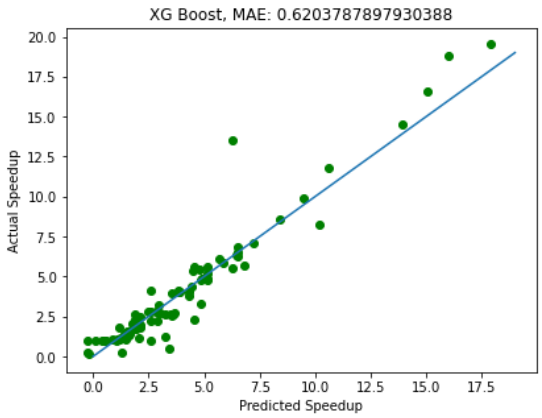

Fig. 4. Actual vs Predicted Speedup XGBoost

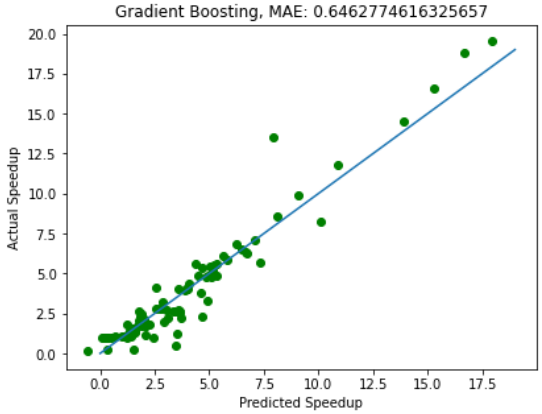

Fig. 5. Actual vs Predicted Speedup Gradient Boosting

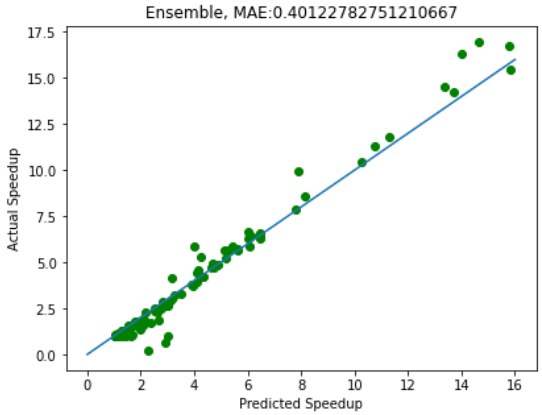

Fig. 6. Actual vs Predicted Speedup Random Forest Ensemble

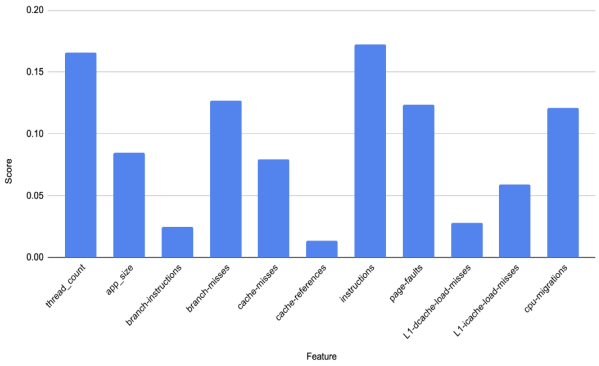

Fig. 7. Features affecting the performance most as scored by XGBoost

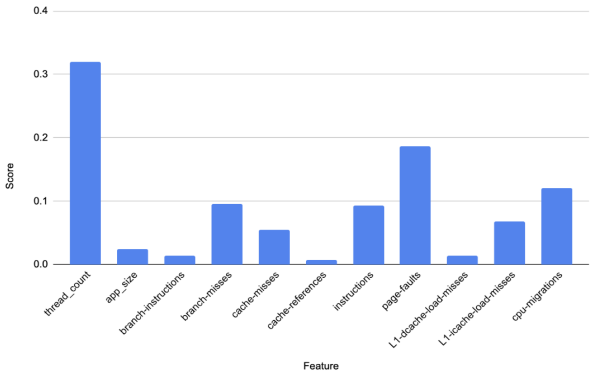

Fig. 8. Features affecting the performance most as scored by Gradient Boosting

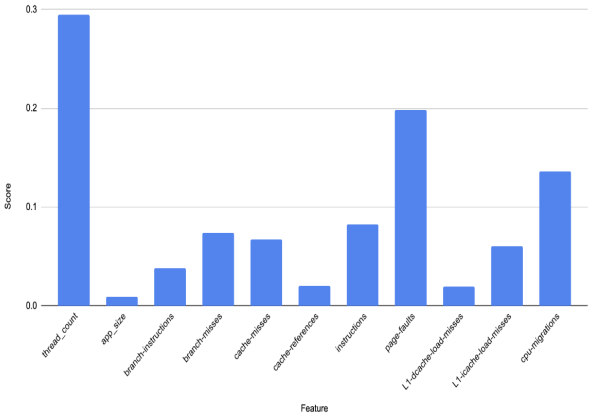

Fig. 9. Features affecting the performance most as scored by Random Forest

Figures 3-6 display the scatter plots for the predicted values and actual values of speedup for the top 3 base models as well as our final ensemble model. As we can see, the predicted values were not far-off from the actual values.

In order to determine the most important features among our chosen 11, we use the top 3 performing base models to display feature importance (figure 7, 8 and 9). We observe that each of the three models give us the same top 6 most important features. Thus, we conclude that for the prediction of the speedup of any application, the most important features are:

- Thread Count
- Page Faults
- Branch Misses
- Number of Instructions
- CPU Migrations
- Cache Misses

VI. Conclusion and Future Work

In this study, we propose a learning-based approach for accurately predicting the parallel speedup compared to single threaded execution on a target platform. As part of our experimets, we studied the relationship between application features like cache performance(cache misses, LLC-load-misses, cache references), branch prediction(branch misses, branch instructions), application size(ipc, instruction, cycles), cpu migration, page faults and context switches with the speedup. We compared performance of 10 machine learning models and found out that the top three performing models are the Random Forest model, XGBoost model and the Gradient Boosting model. Using hyperparamter optimization and ensembling, our best Random Forest model gave us the most accurate predictions, with an MAE of 0.40.

Based on the results obtained from our ML models, we conclude that for the prediction of the speedup of any application, the most important features are the number of threads, page faults, branch misses, number of instructions, cpu migrations, and cache misses. These results are in line with our expectations, and validate our understanding of the factors that can affect the performance of parallel applications.

In our study, we predicted the performance speedup on a specific target machine. In the future, we can extend this study to make cross-platform predictions and include other parallel paradigms like OpenMP, MPI, etc. Furthermore, owing to the limited size of dataset, we were not able to train separate models for different thread sizes. With the availability of a larger cohort, we can use our study to predict the optimal number of threads for an application that would give the best performance.

References

[1] Cass, S. (2010). Multicore Processors Create Software Headaches. Technology Review, 113(3), 74-75

[2] Patterson, D. 2010, "The trouble with multi-core", Spectrum, IEEE, vol. 47, no. 7, pp. 28-32, 53

[3] Flores-Contreras, Jesus & Duran-Limon, Hector & Chavoya, Arturo & Almanza, Sergio. (2021). Performance prediction of parallel applications: a systematic literature review. *The Journal of Supercomputing*. 77. 1-42. 10.1007/s11227-020-03417-5.

[4] 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, doi: 10.1109/SAMOS.2015.7363659.

[5] T. Jain, N. Agarwal, and M. Zahran, Performance prediction for multi-threaded applications,' 2019 *International Workshop on AI-assisted Design for Architecture (AIDArc)*, 2019.

[6] Eric A. Brewer. 1995. High-level optimization via automated statistical modeling. *SIGPLAN Not. 30, 8* (Aug. 1995), 80–91.

[7] F. Hutter, Y. Hamadi, H.H. Hoos, K. Leyton-Brown Performance prediction and automated tuning of randomized and parametric algorithms *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06), LNCS, vol. 4204, Springer-Verlag* (2006), pp. 213-228. pp. 96-103 (2008)

[8] Ipek E., de Supinski B.R., Schulz M., McKee S.A. (2005) An Approach to Performance Prediction for Parallel Applications. In: Cunha J.C., Medeiros P.D. (eds) *Euro-Par 2005 Parallel Processing. Euro-Par 2005. Lecture Notes in Computer Science*, vol 3648. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11549468_24

[9] Marcio Seiji Oyamada, Felipe Zschornack, Flávio Rech Wagner, Applying neural networks to performance estimation of embedded software, *Journal of Systems Architecture, Volume 54, Issues 1–2,* 2008, Pages 224-240.

[10] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, Gjergji Kasneci Deep Neural Networks and Tabular Data: A Survey.*CoRR abs/2110.01889* (2021) 2020

[11] 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.

[12] A. C. De Melo, "The new linuxperftools," in *Slides from Linux Kongress*, vol. 18, 2010.

[13] Jerome H Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics,* pages 1189–1232, 2001.

[14] Ho, T.K., 1995. Random decision forests. *In Proceedings of 3rd international conference on document analysis and recognition*. pp. 278–282.

[15] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). *Association for Computing Machinery, New York, NY, USA, 785–794* . DOI:https://doi.org/10.1145/2939672.2939785

[16] G. H. Loh, "Revisiting the performance impact of branch predictor latencies," 2006 IEEE International Symposium on Performance Analysis of Systems and Software, 2006, pp. 59-69, doi: 10.1109/ISPASS.2006.1620790.

[17] K. Taunk, S. De, S. Verma and A. Swetapadma, "A Brief Review of Nearest Neighbor Algorithm for Learning and Classification," 2019 International Conference on Intelligent Computing and Control Systems (ICCS), 2019, pp. 1255-1260, doi: 10.1109/ICCS45141.2019.9065747.

[18] Schapire RE. Explaining adaboost. In: Empirical inference. *Springer;* 2013. p. 37–52.

[19] Bienia, C. & Li, K. Fidelity and scaling of the PARSEC benchmark inputs. *IEEE International Symposium On Workload Characterization (IISWC'10)*. pp. 1-10 (2010)