Performance Prediction Using Machine Learning for Multi-threaded Applications

Samvid Avinash Zare
New York University
New York, US
sz3369@nyu.edu

Jorge Roldan
New York University
New York, US
jlr9718@nyu.edu

Aditya Walvekar
New York University
New York, US
aw4496@nyu.edu

***Abstract*—Multi-core processors, in 2022 are the default choice for a variety of computing needs from small Internet-of-Things devices to large super-computing clusters. While developing applications and algorithms are primarily done with the needs of a single-threaded or single-core processor at the forefront, running these applications or algorithms in a multi-threaded setting is increasingly important with the limits of the processor manufacturing nearing the edge of quantum physics. Running the programs in a multi-threaded setting poses a set of challenges and prediction of its performance is increasingly complex with the various parameters involved when moving from a single-threaded execution environment to a multi-threaded environment. Thus having a simulation and prediction model for performance gain without actually running it on the often-expensive hardware is of growing importance.**

In this paper, we propose a learning based approach for predicting the performance of standard multi-threaded benchmarks. We attempt to achieve this by collecting various underlying parameters of the application execution and under various input parameters. We then evaluate various machine learning models and to estimate the expected performance.

To achieve this, we use industry standard, publicly available benchmark suites, Parsec 3.0 and Splash 3.0 and use a standard profiler called perf, which is included in the Linux kernel.

We found that the best performing model to predict the speedup to be the K-Neighbors regressor with an average mean absolute error of 0.8.

I. Introduction

The usage of multi-core systems has drastically increased over the last couple of decades as a way to fulfill the computing needs of the ever increasing demands of current applications, where parallel computing plays a key role. Furthermore, due to physical constrains related to energy dissipation, the speed of single processors has reached a limit, and the need for multi-core systems will only become more relevant in the future [1].

To take full advantage of the potential of multi-core systems, it is imperative to be able the predict the performance of parallel applications when using different number of threads. This is specially truth because of the excessive amount of resources that are required to run these applications. Therefore, building predictive models for this purpose can save significant time, and computing resources, as well as making the development of parallel applications more agile and effective.

The complexity of multi-core systems have increased along the popularity and development of these systems over time. For this reason, the analytical models developed for predicting their performance usually have low accuracy, they are inflexible, and are difficult to implement. For this reason, machine learning based models have been shown to perform better, be more versatile, and we will using these learning-based methods in our work.

II. Literature Review

The ability to predict the performance of parallel applications has been a topic of increasing interest over the last couple of decades. A systematic overview of the research done in this area is presented in [7]. In this work, the authors propose a framework to classify the methods of performance prediction in two main categories, namely, prediction domain and prediction methods.

The prediction domain specifies the domain of the application, that is, whether the predictions are dependent or independent with respect to the application. It also denotes whether the prediction are platform dependent [2] or independent, as well as the type of platform use for the predictions, including distributed systems, simulation environments, graphics processing unit [4], multicore systems, and others.

The other category, the prediction methods are divided in three main areas, namely, analytic methods, non-analytic, and characterization. Some of the analytic methods include manually and automatically derived, statistical methods such as linear [3][4] and stochastic methods, quadratic programming and polynomial approximation. The non-analytic methods include Linear regression Support Vector Machines (SVM) [2][3][4][5], Artificial Neural Networks (ANN), classification methods such as classification and regression trees (CART), K-nearest neighbors (KNN) [5], Random forests [3].

Other work on performance prediction include [6], where the authors use statistical methods to predict the execution times of applications using empirical analyses of the same application for small input sizes. Other work done in [12] focuses on cross-platform performance prediction using only partial execution of the application by taking advantage of the repetitive and predictive behavior of parallel applications.

Authors in [13] synthesize analytical models using a learning-based approach, with the goal of predicting the performance of a workload on an specific platform. They accomplish this using performance statistics from the host

platform. Another approach for performance prediction with models such as principal component analysis, normalization, and generic algorithms uses inherent program similarities as in [10].

One of the works that is more closely related to our investigation is [1]. In this work, the authors use learning based approaches to predict the performance of applications while varying amount of threads used to execute the applications on a specific hardware. The authors use the benchmarks suites Parsec-3.0, and Splash-3.0 to obtain the source code of the applications. The best performing model in this work were Gaussian processes regressor, where they achieve a correlation coefficient of 0.67 to 0.82 for the predicted versus the true speedups.

III. Methodology

A. Data generation

The aim for gathering data, apart from the raw execution time and indirectly the speedup compared to one thread, is also to acquire the underlying operating system related metrics. We proposed that multiple factors affect the overall performance of a multi-threaded application. In order to find the metrics which have a higher correlation to the performance, we aim to gather all the available metrics that may change with changing thread count. We achieve this by using a standard, thread-scalable, and workload-varied suite of benchmarks, Parsec-3.0 and Splash-3.0.

Perf is a CPU profiler which is a part of the Linux kernel that will aid in recording most of the required parameters. Based on preliminary testing, we found that the parameters that change for a given benchmark and input combination include: branch instructions, total instructions count, branch misses, L3 cache references, L3 cache misses, CPU Cycles, Instructions per cycle, CPU Clock, page faults, L1 cache data loads, L1 cache instruction loads, L1 cache instruction misses, Last Level Cache load misses, and execution time [8] [9].

B. Problem formulation

Based on the data that can be generated in section Data generation, we proposed the hypothesis that there exist a relation between speed-up of an application and memory access patterns and instruction cycles. Thus we formulated the problem statement that given the instruction set cycles and parameters related to memory access pattern, can we predict the speed-up of an unknown application?.

To have a granularity of predicting performance of unknown application at different number of threads, we decided to train different models for each thread configuration present in our data. But the benchmarks used support only threads which are power of 2, we trained all models for thread count 2, 4, 8, 16, 32, 64, 128 to figure out the best model for the given thread count.

C. Feature Selection

For the task of predicting the speedup, we decided to include the following features provided by the Perf profiling tool. The features used from the Perf profiling tool were Branch instructions, Branch instructions rate, Branch misses, Branch miss percentage, L3 cache misses, L3 cache miss percentage, L3 cache references, CPU cycles, Total instructions, IPC, CPU clock, Page faults, L1 data cache loads, L1 instruction cache load misses, LLC load misses.

We plotted the heat-map for these features and found relatively high correlation between the features and speedup. Thus we decided to use these features for training the models.These features provide statistics about following categories:

- How memory is accessed across different levels of caches such as L3 cache miss percentage, L3 cache references etc.
- Instruction level information such as Total instructions, Branch miss percentage, Branch instruction miss etc.
- Number of cycles involved for example CPU cycles, IPC, etc.

D. Models

According to the problem statement formulated in section III-B, our task was to predict speedup for the given set of features as in section III-C. Since the speedup is a continuous float variable and we had labeled data for different speedups, we decided to train some the following supervised learning models to find the best fit model for the given thread count.

1) Linear regression: Linear regression is the basic model used for task of regression. This model works well if there is a linear relation between input features and output target variable (speedup in our analysis). One common problem faced during training a linear regression model is the over-fitting where model performs well on the training data, but fails to generalize well on the test data. We handled this problem by introducing L2-regularization in the equation of linear regression model. Thus, the learning objective function for linear regression becomes

$$L = ||y - X \cdot w||_2^2 + alpha * ||w||_2^2$$

where X is input feature vector, w is vector of learned weights including the intercept $alpha$ which is the regularization coefficient. $||w||_2$ is the L-2 norm of the weight vector.

2) Gaussian Process Regression: In a Bayesian setting, to solve the problem of regression, the Gaussian Process Regression model is used. In this view, we assume some prior probability distribution and compute the posterior distribution based on the likelihood of the conditional probability of $y|X$, where y is a target variable (speedup in our case) and X is an input feature vector. Thus, as per [11], a regression function with Multivariate Gaussian Prior is given as,

$$P(f|X) = N(f|\mu, K)$$

where $X = [x_1, ..., x_n]$, $f = [f(x_1), ..., f(x_n)]$, f is a regression function, $\mu = [m(x_1), ..., m(x_n)]$, $m()$ is a

TABLE I
SYSTEM SPECIFICATIONS FOR DATA GENERATION

System specifications	
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	64
On-line CPU(s) list	0-63
Thread(s) per core	2
Core(s) per socket	8
Socket(s)	4
NUMA node(s)	8
Vendor ID	AuthenticAMD
CPU family	21
Model	1
Model name	AMD Opteron(TM) Processor 6272
Stepping	2
CPU MHz	2100.011
BogoMIPS	4200.02
Virtualization	AMD-V
L1d cache	16K
L1i cache	64K
L2 cache	2048K
L3 cache	6144K
NUMA node0 CPU(s)	0-7
NUMA node1 CPU(s)	8-15
NUMA node2 CPU(s)	32-39
NUMA node3 CPU(s)	40-47
NUMA node4 CPU(s)	48-55
NUMA node5 CPU(s)	56-63
NUMA node6 CPU(s)	16-23
NUMA node7 CPU(s)	24-31
CPU Flags	(See Appendix I)
Memory (GB)	263.940556
Operating System Name and Version	CentOS Linux 7 (Core)

mean function and K is a kernel function between two data points. Thus, a prediction for a new point X_* is given as

$$y_* = K_*^T[K + \sigma_n^2]^{-1}y$$

where y is vector of target variables in training set and σ_n^2 is the variance of Gaussian noise distributed across input training points.

3) K-nearest Neighbours Regression: This method attempts to predict the target variable by considering the points in the neighbourhood in the hyperplane of the given test point. The size of neighbourhood and voting contribution of each neighbour in the final prediction are considered as hyper-parameters for this model. This works well when the points in the training set with similar characteristics of target variable (speedup in our case) are near to each other in the hyperplane formed by the input features.

Apart from the above models, we tried to fit Random Forest Regression model, Support Vector Regression model for each thread configuration in order to find best fit model that gives least absolute mean error with higher confidence score.

IV. EXPERIMENTAL SETUP

A. System Setup

The performance metrics were recorded on a CentOS 7 based Linux machine running on a multi-threaded, multi-core, multi-socket system with Non Uniform Memory Access(NUMA) and Non Uniform Cache Access (NUCA) . The exact system specifications are mentioned in Table I. We chose this system to enable data collection for Parsec-3.0 and Splash-3.0 with up to 128 threads. Additionally, this system also enables us to examine division of workloads across threads, cores, CPU dies, and sockets.With NUMA, we can also examine the effects of off-chip memory access on the performance of a multi-threaded workload. NUCA adds an additional gradient of performance with increasing thread-count as there is a difference between on and off-core cache access latencies.

B. Benchmarks and Tools

We chose Parsec and Splash as our benchmarks and used perf to record the metrics. We chose to run the benchmark to get the raw execution without perf, and then subsequently run the same benchmark with perf to record the metrics. We did this to avoid any interference of execution time by perf.

While we initially set out to run both Parsec and Splash, we chose to add Splash as a part of our future work as Splash

does not have a benchmark management tool like Parsec's parsecmgmt. Each of the benchmarks in the Splash-3.0 suite had only a fixed number of available inputs - thread count combinations. This made it unsuitable for automation. While the addition of Splash-3.0 would increase the accuracy of our models, it doesn't add too much in terms of variety of workloads. Most of the various categories of workloads such as Financial Analysis, Computer Vision, Engineering, Animation, Similarity Search, Animation, Data Mining, and Media processing [1] are covered by Parsec-3.0.

Coming to the individual benchmarks under Parsec-3.0, we chose to run Black-Scholes, Body Track, Canneal, Face Sim, Ferret, Fluid Animate, Stream Cluster, Swaptions, VIPS and x264. The inputs we chose were simsmall, simmedium and simlarge for a range of execution times and other metrics. We also ran these workloads across a various number of threads (1-128).

We achieved this by writing a Python Script and using the subprocess library to enable command-line execution of each of the benchmarks. We chose to run the benchmarks a total of five times and differing background workloads and averaged the relevant numbers to account for the performance variations. We used the time commands to understand the real time elapsed, as well as parsecmgmt with it's various options to change thread count, input size and benchmarks. Additionally we used perf with options to extract certian features in a TXT File and collated the results in a CSV File as the input to the training suite.

To run the python, we first installed the Parsec-3.0 Benchmark suite on the sytem. This was achieved by running the installation included in the parsecmgmt tool. The parsecmgmt tool can be loaded by sourcing the env.sh file included in the Parsec-3.0 package.

To setup Splash, the process is a bit more involved. We installed gcc-9.2 and make-3.8.5 to install the various benchmarks. We updated the appropriate path in the Makefile.config file and ran the make command to build the benchmarks.

The final step of the data gathering process was running the python script which generated a simple data.csv file.

C. Training and Prediction Setup

1) Training: The task of training the data was accomplished through the following steps.

- Anomaly detection: Due to various reasons such as effect of other running applications on the benchmark, load imbalance, memory limitations, we may observe some odd behaviour in the performance analysis of various benchmarks. This can lead to noise in the input data. We attempted to reduce this noise by plotting the histogram of speed-ups to detect the anomalies. These anomalies are then filtered in the training data to have good fit data while training. As this approach can cause model to overfit, we used regularization while training to avoid overfitting.
- Normalization: As each feature in the given data has different ranges of values possible, for example, branch miss percentage is between 0-100 while branch misses are a few hundreds. Thus in order to fit a better model, all features were normalized using the StandardScalar API provided in the SKlearn module.
- Training model per thread count: The data obtained then was split into multiple datasets on the basis of thread count. Then for each dataset for the given thread count, the data was split into training data and testing data with a ratio of 3:1. For each training data, we trained trained the following models: Support Vector Regressor model, Linear regression model wih L2 regularization, Guassian Process Regression model, Random forest model, K-Neighbors Regressor model. These models were used from python based module Sklearn.
- Hyper-parameter tuning: In order to find the best hyper-parameters for each model, we performed hyper-parameter tuning with the GridSearchCV API in the Sklearn module using the scoring objective to minimize the mean absolute error. Additionally, we performed three fold cross-validation for each model ensuring similar performance characteristics across different subsets of the data.

2) Prediction: The models trained via the process mentioned in section IV-C were tested using the testing data. Each trained model were exposed to unseen testing data to predict speed-up for each of the points in the testing data. For example, figure 1 shows the predictions made by a Support Vector Regression model for a thread count of 2. Similarly, Figure 2 shows predictions made by Gaussian Process Regression model for a thread count of 64.

V. Results

The data gathering was successfully completed to collect various parameters affecting the performance of multi-threaded applications. The output of this process was a simple CSV file with various benchmarks, input sizes and thread counts along with the previously mentioned operating system related metrics.

A. Discussion on Mean Absolute error and Pearson's Coefficients

From all the predictions made by the various models we constructed Table **??**, which computes the mean absolute error obtained by different models across different thread configurations. The Mean absolute error is a measure of how good is the model performing on the testing data. The lower values of error shows better fitness of the model on the testing data.

The additional factor to measure the performance of the model is the Pearson's coefficient between the predicted speed up and the actual speed-up. Pearson's coefficient is a measure of the strength of linearity between two variables and its range is from -1 to +1. The Pearson's coefficient more closer to 1 represents the strong linearity between predicted speed up and the actual speed up. Thus, a higher value of a Pearson's coefficient implies higher stability of the model. Table II shows

the Pearson's coefficient computed for different models across different thread configurations.

We observe that almost all the models were able to perform well with the thread count 2 and 4 as illustrated in figure 4. We suspect this is the case because the benchmarks might have similar patterns in memory access patterns from single threaded application. Additionally, for thread count 4, Support Vector Regression Model was able to fit the RBF kernel on the training data and achieved one of the best performance as presented in figure 3.

For thread count 64, the Gaussian Process Regressor tends to perform better as cache pattern becomes constant due to 64 threads and thus cache related feature values become symmetric about the Gaussian prior mean, making the Gaussian Regressor to fit better. Figure 4 shows that the best model for thread count of 64 is indeed the Gaussian Process Regressor. The Pearson's coefficient suggests the higher confidence across the best models for the each thread configuration obtained from minimizing the mean absolute error.

For higher number of threads - 128, the K-Neighbor regression model performs better as the features obtained from Perf analysis follows similar patterns due to availability of the actual threads for the parallel processing saturates. Thus, it becomes relatively easy to predict the speed up using nearest neighbours in the hyperplane of the input features. Furthermore, we can see from figure 3, that K-neighbors regressor model performs well for threads 2, 4, 8, 16, 64, and 128, making it one of the most accurate and more versatile models. This is confirmed in figure 5, which shows K-Neighbors regressor with an average mean absolute error of **0.8**. The next best models are Support vector regressor and Random forests with **0.95**, Gaussian process regression with **1.01**, and Linear regression with **1.18**.

B. Discussion on Error plots

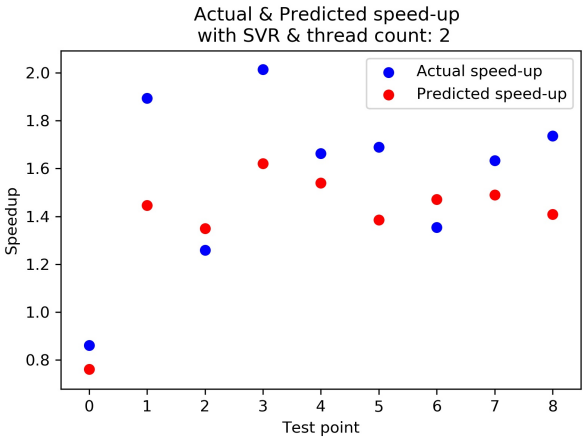

Fig. 1. Prediction by Support Vector Regressor for thread count 2

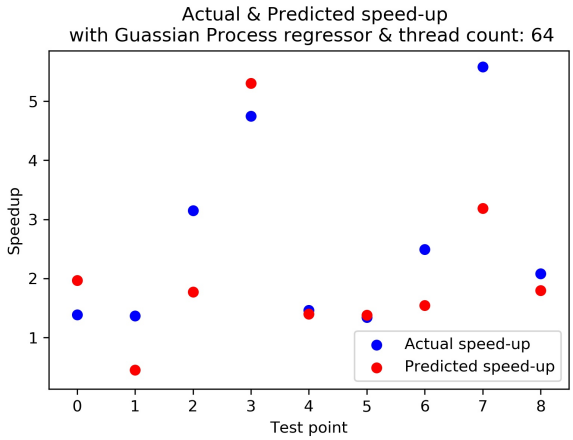

Fig. 2. Prediction by Gaussian Process Regressor for thread count 64

VI. Conclusion and Future Work

A. Conclusion

From the data gathering process, we see that the speedup often has a sweet spot of thread count for a given set of benchmark and input-size. This can be explained as a result of the trade offs between thread-setup overhead and amount of parallelism achieved.

We can conclude that given certain performance statistics from an applications related to memory access patterns and instruction cycles such as the ones mentioned in section III-C, we can train different machine learning models that can accurately predict the speed-up of an unknown application using different number of threads.

We have found that the best performing, accurate, and versatile model to predict the speedup to be the K-Neighbors regressor with an average mean absolute error of **0.8** for threads 2, 4, 8, 16, 32, 64, and 128. The next best models are Support vector regressor and Random forests with **0.95**, Gaussian process regression with **1.01**, and Linear regression with **1.18**. We believe that gathering more data with the Benchmarks Parsec-3.0 and Splash-3.0 with the scripts included in the source code, we could reduce even more the error of these models.

B. Future Work

- Gather more data using the Splash 3-0 benchmark.
- Try on applications that supports thread which are not power of two, because, this will give better intuition of memory access patterns as there will be lot of false sharing.
- Detect anomalies on different basis rather than by just observing the odd speed-ups
- Fit a Neural network for the regression task to enhance accuracy

TABLE II
PEARSON'S COEFFICIENT ON TEST DATA

Number of threads							
Models	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Support Vector Regressor	**0.832**	0.358	0.493	-0.09	0.271	**0.779**	0.56
Linear regression model with L2 regularization	**0.833**	0.56	0.43	0.342	0.455	0.367	0.411
Gaussian Process Regression model	0.609	**0.618**	0.481	**0.777**	**0.585**	**0.794**	0.62
Random forest model	0.661	0.436	0.149	0.674	0.374	0.237	**0.835**
K-Neighbors Regressor model	0.352	-0.434	**0.787**	0.538	0.478	0.67	0.644

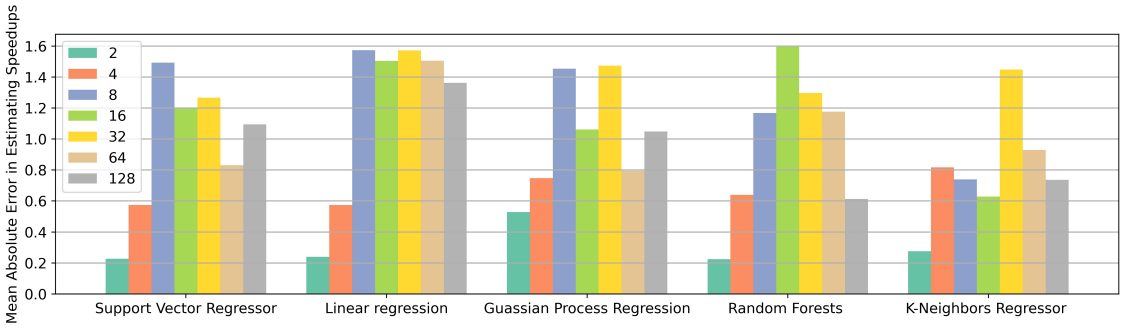

Fig. 3. Mean Absolute Error grouped by model

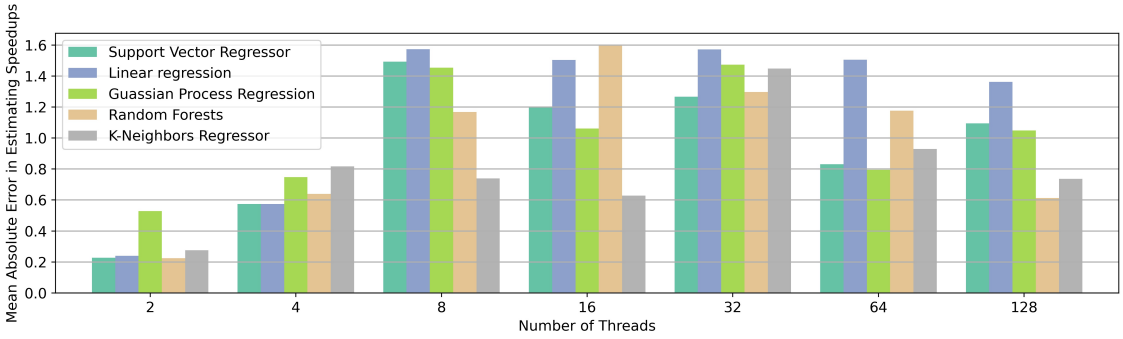

Fig. 4. Mean Absolute Error grouped By Number of threads

ACKNOWLEDGMENT

We would like to thank Dr. Mohamed Zahran for his support and guidance. We would also like to thank the staff at the Courant Institute of Mathematical Sciences for providing the computing resources to complete this work.

APPENDIX I

CPU Flags enabled : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc art rep_good nopl nonstop_tsc extd_apicid amd_dcm aperfmperf pni pclmulqdq monitor ssse3 cx16 sse4_1 sse4_2 popcnt aes xsave avx lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs xop skinit wdt fma4 nodeid_msr topoext perfctr_core perfctr_nb cpb hw_pstate retpoline_amd ssbd ibpb vmmcall arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold

REFERENCES

[1] Nitish Agarwal, Tulsi Jain, and Mohamed Zahran. *Performance prediction for multi-threaded applications*. URL: https://eecs.oregonstate.edu/aidarc/paper/PP.pdf.

[2] Mehmet Akay, Cigdem Aci, and Fatih Abut. "Predicting the Performance Measures of a 2-Dimensional Message Passing Multiprocessor Architecture by Using Machine Learning Methods". In: *Neural Network World* 71 (Mar. 2015), pp. 1907–1931. DOI: 10.14311/NNW.2015.25.013.

[3] Marcos Amarís et al. "A comparison of GPU execution time prediction using machine learning and analytical modeling". In: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. 2016, pp. 326–333. DOI: 10.1109/NCA.2016.7778637.

[4] Thanh Tuan Dao et al. "A Performance Model for GPUs with Caches". In: *IEEE Transactions on Parallel and*

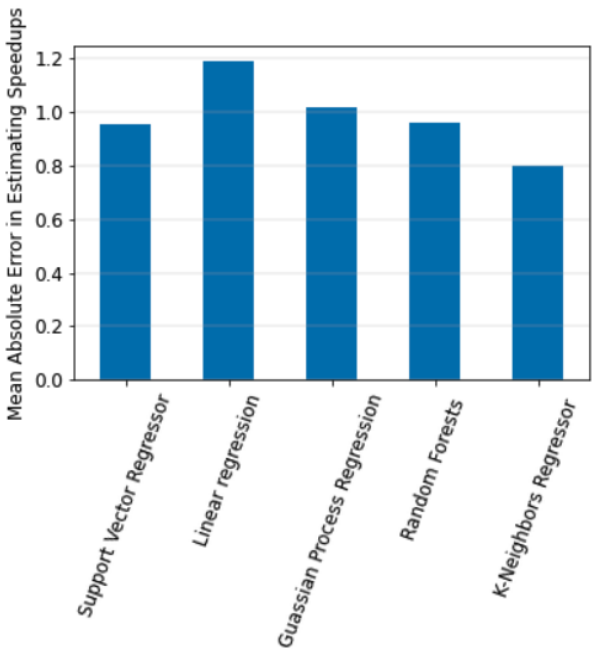

Fig. 5. Average of Mean Absolute Error for each Model

Distributed Systems 26.7 (2015), pp. 1800–1813. DOI: 10.1109/TPDS.2014.2333526.

[5] Tri Doan and Jugal Kalita. *Predicting run time of classification algorithms using meta-learning - international journal of machine learning and cybernetics*. July 2016. URL: https://link.springer.com/article/10.1007/s13042-016-0571-6.

[6] Rodrigo Escobar and Rajendra V. Boppana. "Performance Prediction of Parallel Applications Based on Small-Scale Executions". In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 2016, pp. 362–371. DOI: 10.1109/HiPC.2016.049.

[7] Jesus Flores-Contreras et al. *Performance prediction of parallel applications: A systematic literature review - the journal of supercomputing*. Sept. 2020. URL: https://link.springer.com/article/10.1007/s11227-020-03417-5#citeas.

[8] Markus Frank et al. "Performance-influencing Factors for Parallel and Algorithmic Problems in Multicore Environments: Work-In-Progress Paper". In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 21–24.

[9] Ashif S Harji, Peter A Buhr, and Tim Brecht. "Comparing high-performance multi-core web-server architectures". In: *Proceedings of the 5th Annual International Systems and Storage Conference*. 2012, pp. 1–12.

[10] 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.

[11] Jie Wang. *An Intuitive Tutorial to Gaussian Processes Regression*. Sept. 2020.

[12] 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.

[13] Xinnian Zheng et al. "Learning-based analytical cross-platform performance prediction". In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2015, pp. 52–59. DOI: 10.1109/SAMOS.2015.7363659.