Performance Prediction of Multithreaded Applications: Version a

Author: Ying Liu, Group: 20

Abstract

Running applications in a multi-threaded setting becomes increasingly popular to achieve better performance. Naturally, it introduces the questions of how better we can improve the performance compared to the single thread. Thus having a prediction method for performance gain becomes necessary.

This report works on the prediction based approach for predicting the speedup of standard multi-threaded benchmarks under different threads settings on different machines. First I collected various underlying parameters of the application execution and machine information as well. Then I evaluated various prediction models to estimate the expected performance.

The training and test data came from the multiple runs of programs in the benchmark suites Parsec 3.0 and Rodinia. And the best performing model to predict the speedup is XGBoost with an average mean absolute error of 0.335.

Introduction

The utilization of multithreaded is increasing as the current applications tend to be more data intensive. On the other hand, the computing servers nowadays, from Cloud Cluster to home desktop, mostly come with multicore which promotes multi-thread computing.

To take advantage of multicore, a commonly asked question is, how many threads are optimal to harvest the best performance gain on a certain machine. If we can answer the question, we can always schedule the optimal number of threads to achieve better performance while avoiding overhead and waste.

Hence, building the prediction method is important, but it's also hard. Running the programs in a multi-threaded setting is more complicated than single thread as it introduces factors that are indefinite. To overcome this complexity, in this report, I proposed to use multiple machine learning methods to explore the prediction problem.

Literature Survey

For the cross platform performance prediction, it mainly falls into 2 categories: (1) prediction based on the program similarity[1][2] (2) prediction based on small-scale executions based[3][4].

In the first category, the prediction is done offline. Multiple benchmark programs are executed to create a benchmark space, and when a new program arrives, we will first find the most similar program to it and infer the expected performance. Figure 1 shows the general ideas. The advantage of this approach is, as the benchmark space becomes sufficient, it's more likely to get a similar program and achieve accurate prediction. However, this approach relies heavily on the prediction method and how to find a good prediction model is the main challenge.

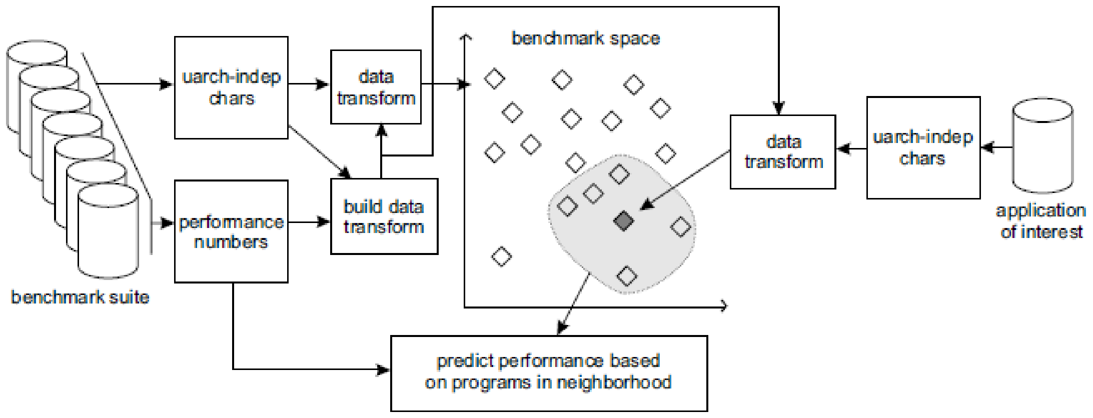

Figure 1: Ideas of prediction based approach from [1]

In the second category, the prediction depends more on the small sample run, and the result would be adjusted by the sample run. This is good if the sample part of the program is representative enough and we may not even require a sophisticated model for prediction. But it will fail when the computation pattern changes during the run. Also it requires a solid infrastructure to support the reference if running and predicting in real time.

One of the works that is more closely related to this report is [5]. In this work, the authors use a prediction based approach for predicting the performance of multi-threaded benchmarks.They achieved this by collecting various underlying parameters of the application execution on one machine and evaluating various models to estimate the expected performance. In this report, I will expand it to multiple machines and explore more machine learning algorithms.

Proposed Idea

Problem Overview

Let's reiterate the problem definition: given there are inputs of

- Specs of a multithreaded program.
- Specs of the machine that will be used to run the program.
- Problem size.
- Speedup on some machines.

What would the speedup (relative to a one thread) of an application be on a different machine? In order to achieve this goal, multiple questions need to be answered:

1. Which specs from the program are of interest and how to collect them.
2. What specs from the host are important and how to collect them.
3. How to get problem size.
4. What are our prediction models?

And the following sessions in this chapter would propose the solution to above questions.

Data Generation

From the problem definition, it can be safely assumed that multiple characteristics from both the program and machine affect the performance of the multi-thread application. Hence the goal here is to run a variety of multithreaded programs and collect their performance metrics as much as possible.

Multiple multithreaded programs can be gained from benchmark suits. There are several popular benchmark suites available. In this report, Parsec-3.0 and Rodinia (only the CPU part) are used because they are both thread scalable and they each include different kinds of workload for CPU.

In terms of how to collect performance metrics from the program, Linux perf, a performance analyzing tool in Linux available from Linux kernel, was picked. To be more specific, perf stat command was used to collect multiple Hardware events, Software events, Hardware cache events and kernel PMU events. At this step, I proposed to collect as many events as possible and some statistical analysis to run later to pick the highly correlated ones.

Regarding the host spec, three Linux commands are used: 1) lscpu to display information about the CPU architecture, 2) inxi to retrieve the memory information of the machine, 3) sar to get the current host memory and CPU utilization status.

Prediction Model

According to the problem definition, the task is to predict speedup from the given set of features. As the predicted variable, speedup, is continuous,the prediction model falls into the categorial of supervised learning and regression model. Here are some proposed prediction models:

1. Traditional linear regression.
2. Linear regression with L2 penalty.
3. K Nearest Neighbors Regression.
4. Bagging method: Random forest.
5. Boosting method: Gradient boosting and XGBoost.

Experimental Setup

System Setup

Host

The experiments were run on NYU HPC. Those are a group of CentOS 7 Linux machines running on a multi-core multi-socket system with Non Uniform Memory Access(NUMA) and Non Uniform Cache Access (NUCA). The picked hosts are in Table I.

Hostname	**CPU**	**Memory**	**OS**	**Purpose**
crunchy1	Four AMD Opteron 6272 (2.1 GHz) (64 cores)	256 GB	CentOS 7	CPU and memory intensive processes
crunchy5	Four AMD Opteron 6272 (2.1 GHz) (64 cores)	256 GB	CentOS 7	CPU and memory intensive processes
crunchy6	Four AMD Opteron 6272 (2.1 GHz) (64 cores)	256 GB	CentOS 7	CPU and memory intensive processes
crackle1	Two Intel Xeon E5630 (2.53 GHz) (16 cores)	64 GB	CentOS 7	generic number crunching
crackle3	Two Intel Xeon E5630 (2.53 GHz) (16 cores)	64 GB	CentOS 7	generic number crunching
crackle5	Two Intel Xeon E5630 (2.53 GHz) (16 cores)	16 GB	CentOS 7	generic number crunching
snappy1	Two Intel Xeon E5-2680 (2.80 GHz) (20 cores)	128 GB	CentOS 7	generic number crunching

snappy4	Two Intel Xeon E5-2680 (2.80 GHz) (20 cores)	128 GB	CentOS 7	generic number crunching
cuda1	Two Intel Xeon E5-2680 (2.50 GHz) (24 cores)	256 GB	CentOS 7	(In this report, we use it only for CPU computation.)

Table 1: Hosts to run benchmarks.

Benchmark

I picked multiple programs from Parsec-3.0 and Rodinia with the hope to introduce various categories of workloads. The details are in Table 2.

Benchmark Suite	**Program**
Psrsec-3.0	blackscholes
	bodytrack
	facesim
	fluidanimate
	swaptions
	canneal
	streamcluster
	freqmine
Rodinia	bfs
	kmeans
	lavaMD
	myocyte

Table 2: Programs from benmark suits.

Even though benchmark suite Parsec-3.0 allows to specify input size as simsmall, simmedium and simlarge, Rodinia doesn't have this optional. This becomes a challenge as problem size is an input of the prediction model. To overcome this, a proxy metrics is used, that is the computational time in seconds if executing this program with a single thread. Moreover, Rodinia also includes GPU workload which is not the goal of this report. I went through the program in Rodinia and picked the multi-thread programs that are executed on CPUs.

Data Collection

To start with, benchmark suites Parsec-3.0 and Rodinia are installed on each of the experiment machines. To install Parsec-3.0, I utilized the parsecmgmt tool that can be loaded by sourcing the env.sh file after downloading the Parsec-3.0 package. For Rodinia, after downloading the package, I went to each eligible program folder and ran the make command to build the benchmarks. And gcc-12.2 is used.

Then, a Python script was implemented to execute multiple command-line instructions. Define one round as running one benchmark program with one certain number of threads. For each round, the python script executed the commands in this order: 1) get host spec and current utilization status, 2) run the benchmark with certain threads with command perf, 3) postprocess the output from all the command line instructions and save to the output folder in csv format.

Each program ran with a various number of threads (1, 2, 4, 8, 16, 32, 64, 128). To deal with the cold start, the performance metrics from the first several runs in each round are discarded. Also each benchmark run would be executed 5 times and the average computational time was used for calculating speedup.

Modeling

A python colab was implemented for statistical analysis and forecasting. This analysis followed the steps of data loading, anomaly and missing data detention, feature exploration, preprocessing, training and evaluation.

Preprocessing

Missing data: due to access or other reasons, the perf command fetched different sets of metrics on different machines. As a mitigation, the program and host specs whose empty count are larger than 20% were filtered out.

Anomaly: because of load imbalance, memory limitations and so on, the speedup number can be abnormally large. This would lead to the skewness in prediction. After plotting the histogram of speedup, the training entries with speedup > 100 are filtered out.

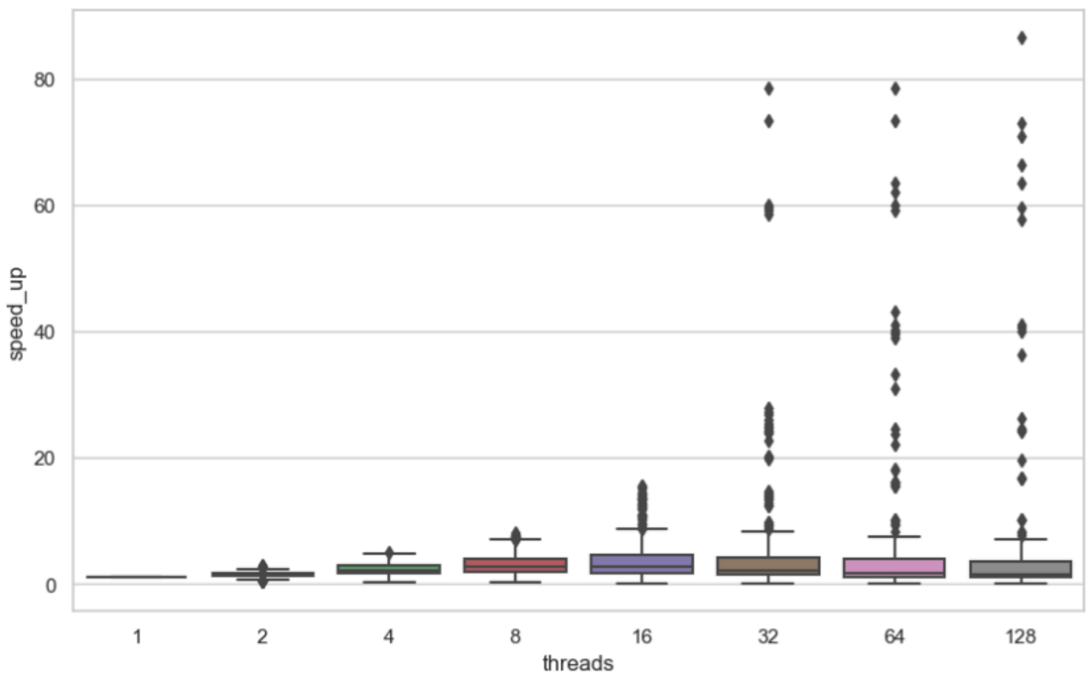

Figure 2: speed up distribution per thread (after filtering out anomaly)

Train test split:The speedups in machine crunchy6 were treated as test sets, and all the other were considered as the train set.

Training

For each training data, the models proposed above were trained using Python package sklearn and xgboost. Note that there were more than 50 features that remained unselected, and applying all those blindly to the model would lead to overfitting. Luckily, Random forest and Boosting models are able to perform feature selection while training, and generate the feature importance score as one of the output. For linear regression with L2 penalty, regularization can be added to avoid overfitting. Hence, I started with model

1. Traditional linear regression
2. Linear regression with L2 penalty.
3. Bagging method: Random forest.
4. Boosting method: Gradient boosting and XGBoost.

And I manually picked the important features from the result of traditional linear regression, random forest model and boosting models. Then I trained K Nearest Neighbors Regression with the selected features. Table 3 is the feature space:

Category	**Sub Category**	**Metrics that are correlated with speedup**

Program Metrics	Hardware Events	IPC, IPS, cache-references, stalled-cycles-frontend, branch-instructions, branch-misses, cpu-cycles, instructions
	Hardware Cache Event	LLC-loads, LLC-stores, LLC-load-misses, branch-loads, branch-load-misses, L1-dcache-load-misses, L1-dcache-loads, L1-icache-load-misses, L1-dcache-prefetch-misses, dTLB-load-misses, iTLB-load-misses, dTLB-loads, iTLB-loads
	Software Event	minor-faults, cpu-clock, page-faults, task-clock, context-switches, cpu-migrations
	Kernel PMU Event	cache-misses, msr/aperf/, msr/tsc/, msr/mperf/
	problem size	runtime_serial (as proxy of problem size or problem complexity)
	number of threads	number of threads
Machine Metrics	Host Spec	BogoMIPS, Number of CPUs, Number of NUMA node(s), Core(s) per socket, Thread(s) per core, Number of Socket(s), Stepping, Model
	Host Utilization	host_memused, cpu_utilization_from_user

Table 3: features space

All the above 4 models require hyper-parameter tuning. That was done by the GridSearchCV in the Sklearn module to minimize the mean square error and also I used five fold cross-validation ensuring similar perform

Measure Performance

Two measurements Mean Square Error and Mean absolute error were used in test data to measure how good the model performed. The lower values of error are better.

Experiments & Analysis

Data Exploration

Figure 2 shows for each program, the value of speed up with respect to different threads. It is pretty clear that different programs experience different speedup growth. For example, programs like bfs benefit less or even harm from multithread. This is due to the nature of the bfs algorithm, frequent synchronization is required when traveling to the next layer in the graph. On the other hand, for programs such as kmeans, they are much easier to benefit from multi thread due to less barrier in the program.

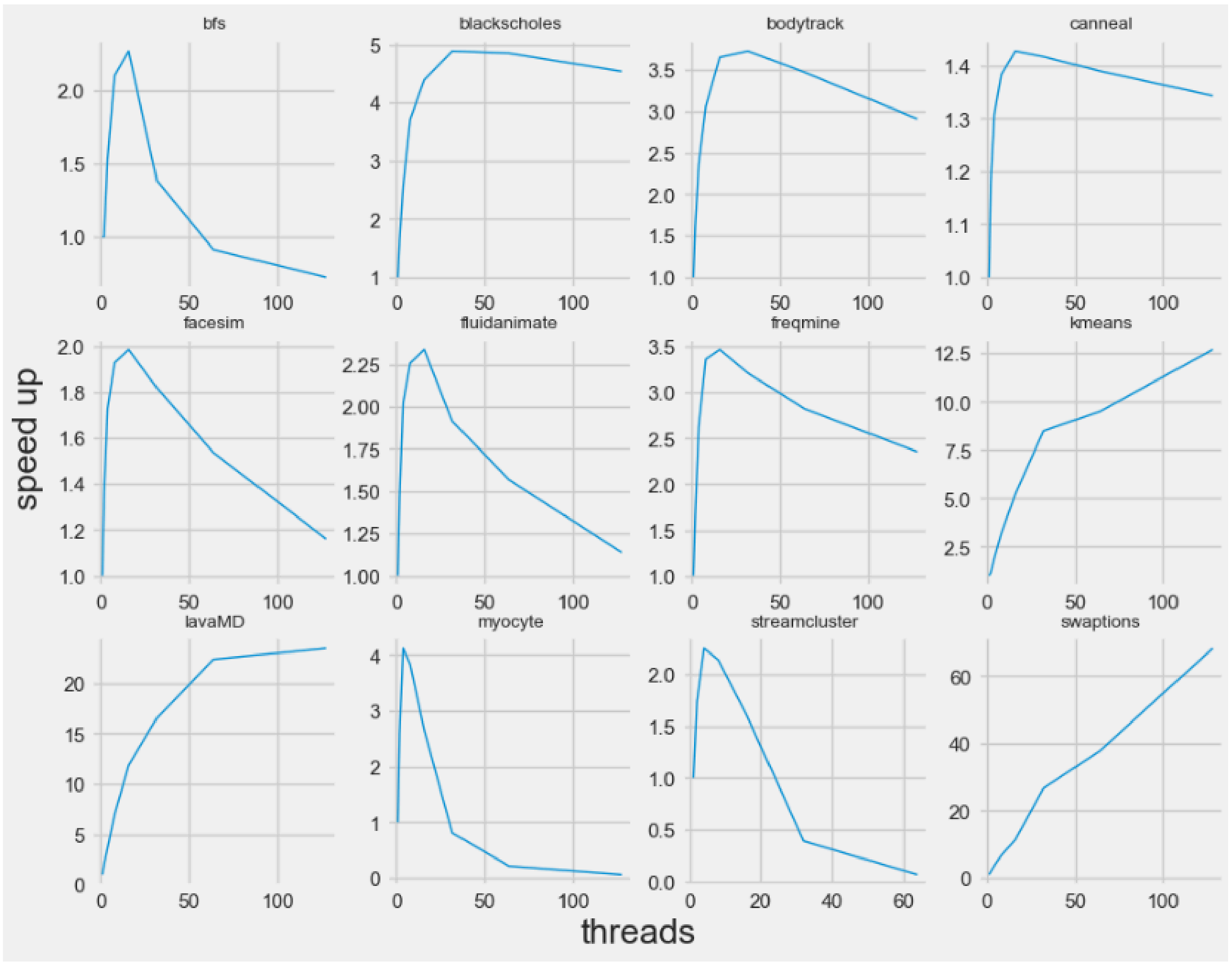

Figure 3: avg(speed up) w.r.t. threads per program

Figure 3 shows for each host, the value of speed up with respect to different threads. From table 2, we can group 9 hosts into 4 groups based on the general machine spec: crackle, crunchy, cuda and snappy. And it's observed that each machine group performs the similar speedup pattern, which is expected.

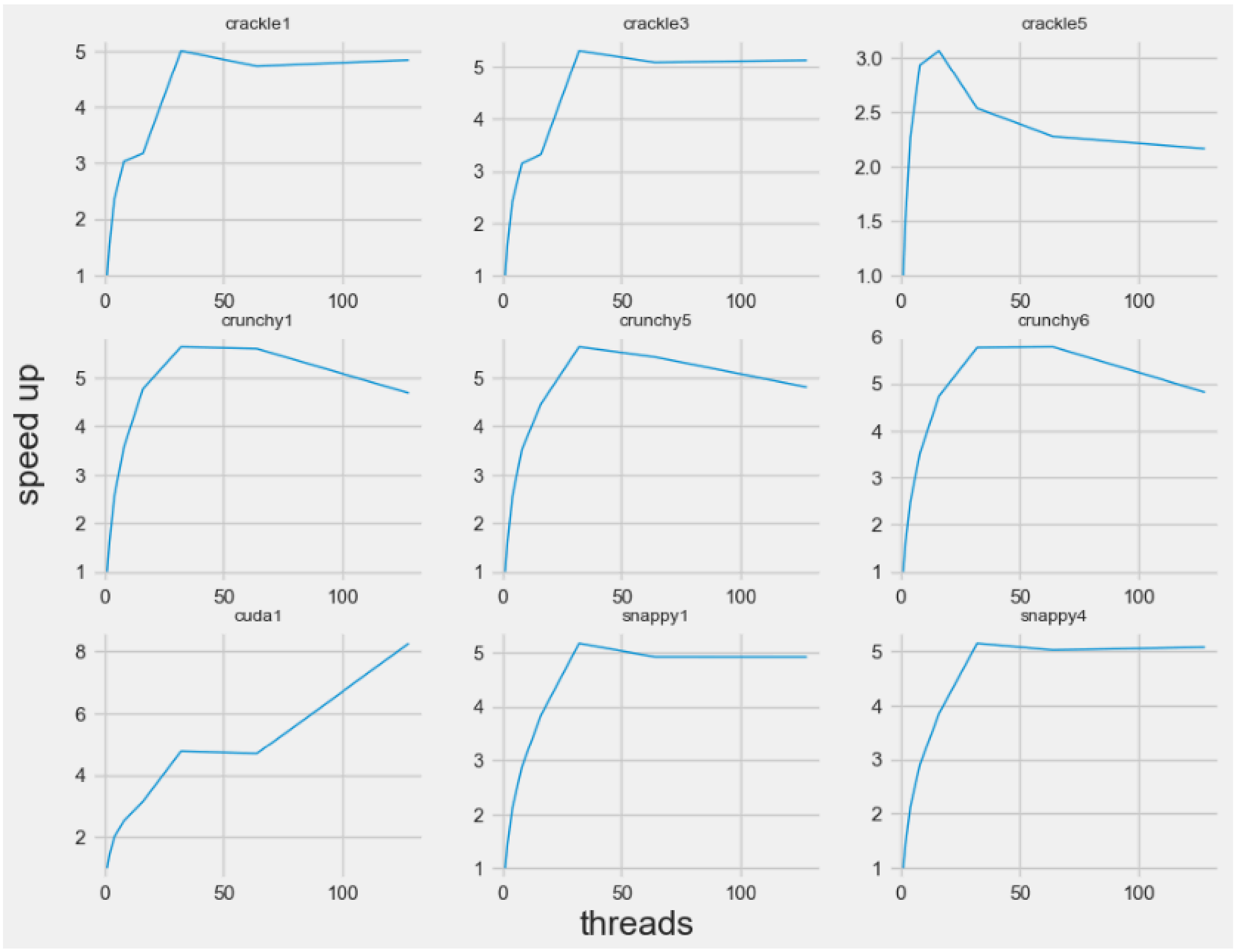

Figure 4: avg(speed up) w.r.t. threads per host

Training and Prediction

Table 4 and 5 record all the predictions measurements made by the various models in test data.

Threads	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**Overall**
Linear Regression with L2 regularization	7.844	5.130	5.012	12.133	33.203	44.082	31.581	19.512
KNN	0.928	0.393	1.372	8.508	16.807	25.168	16.001	9.703
Random Forest	0.116	0.041	0.061	0.178	0.365	1.890	1.194	0.532
Gradient Boosting	0.084	0.104	0.406	0.237	1.292	0.861	1.166	0.577
XGBoost	0.047	0.071	0.224	0.216	0.593	0.176	0.977	0.314

Table 4: mean square error in test data per model per thread.

Threads	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**Overall**
Linear Regression with L2 regularization	1.754	1.376	1.632	2.504	3.427	3.810	4.465	2.668
KNN	0.568	0.448	0.825	1.701	1.930	2.328	1.646	1.341
Random Forest	0.133	0.137	0.181	0.267	0.387	0.572	0.435	0.298
Gradient Boosting	0.171	0.224	0.458	0.386	0.707	0.530	0.693	0.446
XGBoost	0.160	0.201	0.336	0.354	0.478	0.304	0.546	0.335

Table 5: mean absolute error in test data per model per thread.

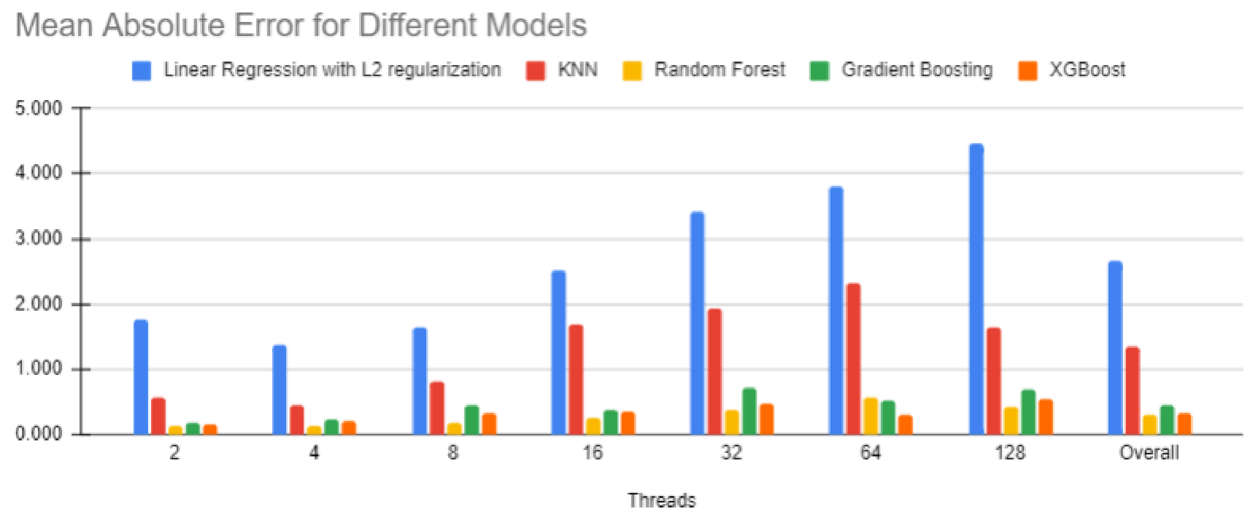

Figure 5: mean absolute error in test data per model per thread

From the observation, linear regression model has the worst performance, KNN model performs decent, random forest and boosting models give the best results. The reason why linear models perform poorly is because the speedup is not growing linearly with respect to predictors as observed from figure 2. The linear model performs fine when thread number is less than 8, and becomes significantly worse as thread number increases. As a matter of fact, as thread number grows, all models get worse due to the increasing uncertainty as threads grow, but other non-linear models such as random forest can handle it better.

The KNN model performs decently, however it can't beat random first and boosting models. My assumption is due to the feature selection. As we have more than 50 features, random forest model and boosting model take care of large feature numbers by selecting them while training the model, however, KNN uses the given feature which makes the model less flexible.

Finally, the algorithms that leveraged the ensemble learning are performing the best. No matter it utilizes the bagging technique (random forest), or boosting technique (gradient boosting and XGBoost). This is because bagging and boosting decrease the variance of a single estimate as they combine several estimates from different weaker models and that helps to reduce high bias. In our case, it means they can better handle large numbers of features.

Feature Important Scores

Both random forest and boosting models generated feature importance scores at the end of training. Figure 6 and 7 shows the feature importance score from XGBoost and random forest.

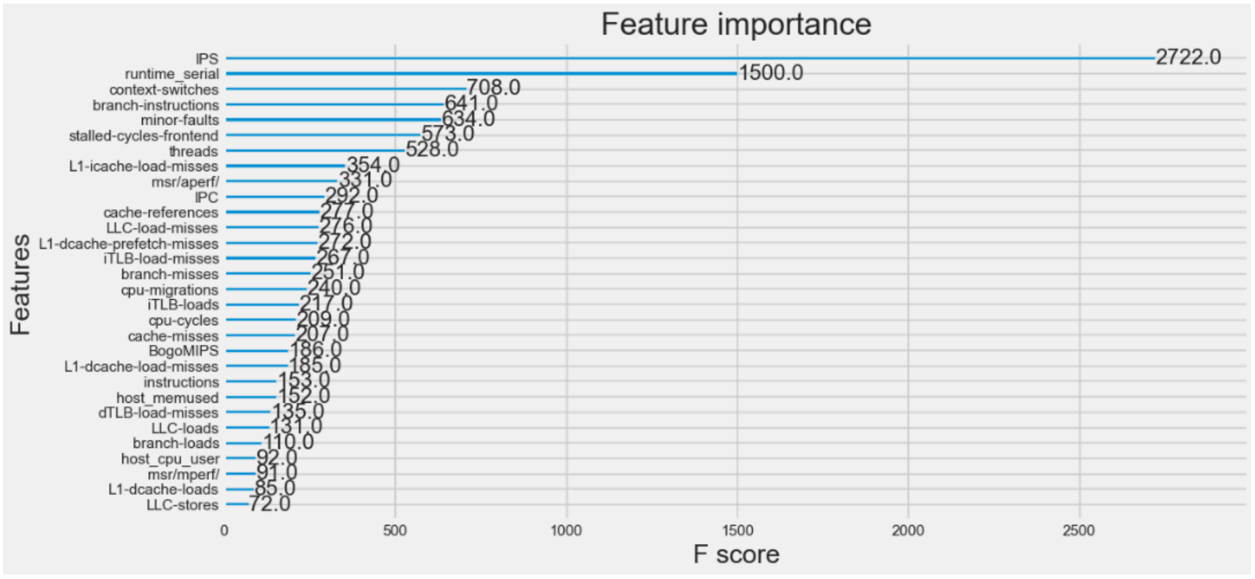

Figure 6: Feature importance score from XGBoost

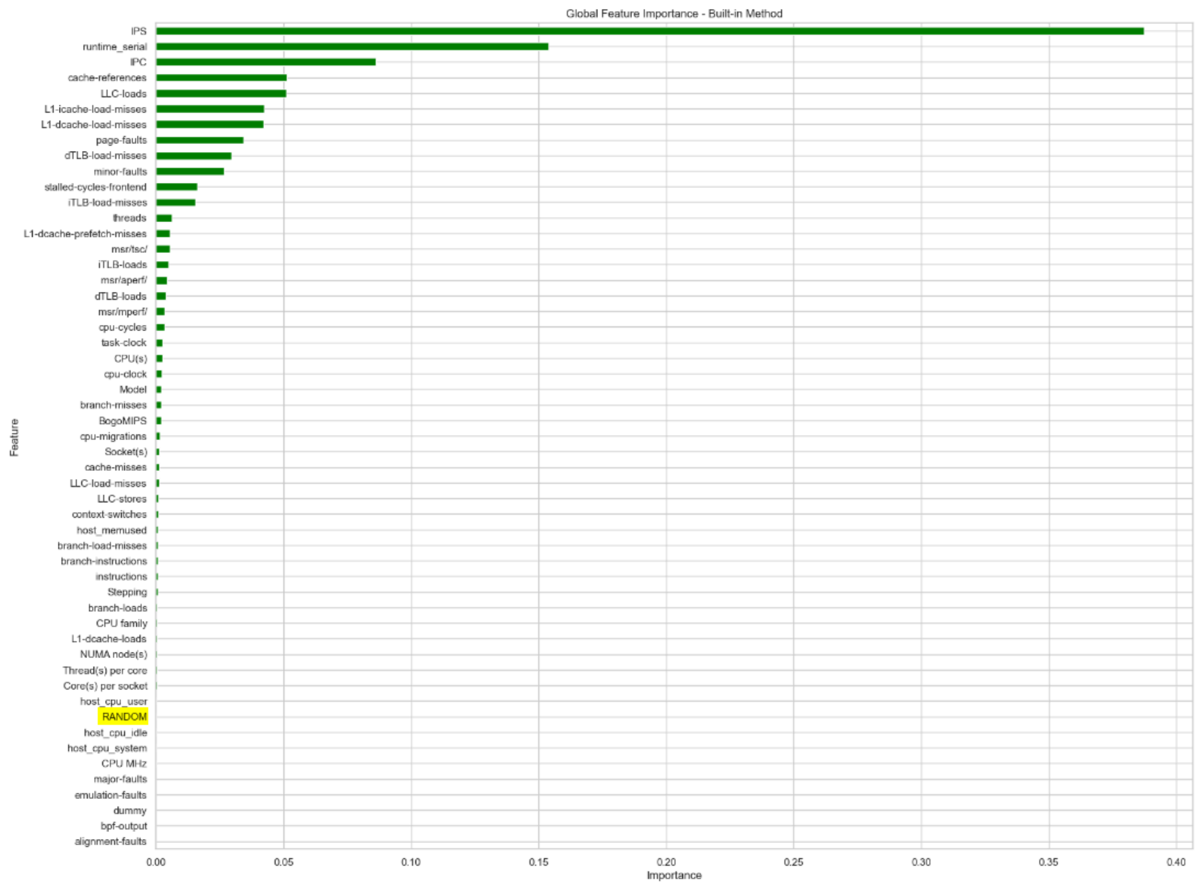

Figure 7: Feature importance score from Random Forest

One surprising finding is, I would expect the speedup is influenced by the host utilization, but this is not the case.

Conclusions

1. Different programs on the same host, and same program on different hosts, are observed to have different speedup increase patterns as thread numbers increase.
2. The speedup can be predicted. And machine learning ensemble methods, either random forest or boosting methods, generate good predictions.
3. There are some machine spec and program spec that are more correlated to the speedup, as shown in table 3, figure 6 and figure 7.

Improvement

Due to the bandwidth, here are some bullet points I would like to investigate but was unable to:

1. The feature selection is a bit rough. If there is more time, I would like to better understand why machine learning models picked these features and feed a better feature space to the KNN model. I would expect it to give better results.
2. The current report only proves that the speedup can be well predicted if we have the full features of the testing data. However, I would like to explore if it's possible to only partially run or not run the program at all to still predict the speedup accurately, i.e. the second method in the Literature Survey. And then compare both methods in the report.

References

1. Performance Prediction based on Inherent Program Similarity
2. Learning-based Analytical Cross-Platform Performance Prediction
3. Performance Prediction of Parallel Applications Based on Small-Scale Executions | IEEE Conference Publication
4. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution | IEEE Conference Publication
5. Performance Prediction Using Machine Learning for Multi-threaded Applications
6. https://courant.nyu.edu/dynamic/systems/resources/computeservers/