Project Overview

In today's day and age, we are constantly trying to optimize our algorithms to improve performance and minimize execution time. One of the way we can achieve those goals is by using loop parallelism. If an algorithm is made of independent iterations, such a technique can be used. Using HPX's for-loop algorithm, the different iterations are assigned to threads and then to different CPU's. One of the key parameters of parrallelization is chunk size which, in that case, represent how much work is assigned to a single thread. The goal of this project is to be able to predict the optimal chunk size for a given task. One method that has proven to be successful to predict such parameter is machine learning. In machine learning, you train a model based on data to get predictions on new situations. The first section will focus on the data used to train the machine learning algorithms.

1 Data Description

The data that must be generated is comprised of features and target values. Features represent the characteristics of a given algorithm. In the case of this work the 6 features are

— <Number of operations per iteration>
— <Number of float operations per iteration>
— <Number of caparison operations per iteration
— <Deepest loop level>
— <Number of iterations>
— <Number of threads>

The first 4 features are extracted at compile time by a ClangTool called Loop-Convert and the 2 other features are collected at runtime. The target values represent what we want to predict for given features. In that case, the target value is the optimal chunk size for a loop. But how does one find the optimal chunk size for a loop ? Well, we have to try many chunk size candidates and measure the execution time. The chunk size with minimal execution time is selected as target value. Such a method may seem time consuming but this is simply to generate data. Once data has been generated and the model has been trained, finding the optimal chunk size for a new algorithm is very fast to compute. The set of chunk size candidates will be described as CS. In the article [0] the following set has been used $CS = \{0.001; 0.01; 0.1; 0.5\}$ Note that here the chunk size is a percentage which represent what fraction of the total number of iterations is given to a thread.

1.1 Data Generation

Data generation consist of running experiments. Each experiment is an example of features and execution times (from which we can extract optimal chunk size).

Here is an example of a data-set comprised of three different experiments.

```
NTop NFop NCop DDL Ntte Nthr 0.001 0.005 0.01 0.05 0.1 0.2 0.5
1223706 320800 10053 2 200 6 0.015632 0.0144048 0.011975 0.0117228 0.012009 0.0221492 0.0248183
7203 1801 901 1 900 8 0.00118291 0.00105788 0.00101698 0.00106958 0.00136634 0.00134487 0.00330694
16163 4041 2021 1 2020 8 0.00514625 0.00478956 0.00474357 0.00529247 0.00697644 0.0143438 0.0171738
```

To generate an given experiment, you need a lambda function,a set CS,a number of iterations and a number of threads. These are all the steps necessary to run an experiment with examples of code. (More details will be added to the wiki and on the repository)

-1 Choose a given lambda function, set CS, number of iterations and number of threads. In my case all experiments are written in a text file.

```
Matrix_Matrix_Mult,3,300,1
Matrix_Matrix_Mult,3,400,1
Matrix_Matrix_Mult,3,500,1
Matrix_Matrix_Mult,3,100,2
Matrix_Matrix_Mult,3,200,2
Matrix_Matrix_Mult,3,300,2
```

The collumn represents :
<Lambda function>, <header file>, <Number of iterations>, <Number of threads>

-2 Extract the features of an algorithm (static and dynamic) Static features are extracted by using the loop-convert ClangTool. It must be executed on the right cpp file.

```
~/compile/cfe-6.0.0.src/build/bin/loop-convert extraction.cpp -- -Iinclude /path/to/hpx /path/to/boost
```

-3 For a given chunk size candidates, run an HPX for loop on the lambda function and measure the execution times. Repeat this step many times and take the mean of execution times.

-4 Repeat the previous step with on all chunk size candidates.

```
using namespace hpx::parallel;

double time_now;
double N_rep=10;
double mean_time=0;
double elasped_time;
for(int j(0);j<N_rep+1;j++){
    time_now=mysecond();
    for_each(execution::par.with(dynamic_chunk_size(vector_size*chunk_candidate)), range.begin(), range.end(), f);
    elapsed_time= mysecond() - time_now;
    if(j!=0){
        mean_time+=elapsed_time;
    }
}
std::cout<<mean_time/N_rep;
```

2 Matrix Multiplication

In this section, I will focus on the Matrix Multiplication algorithm. All experiments shown will have been run with a matrix multiplication lambda function in a HPX for-loop. The goal is to start analyzing a smaller data-set before slowly expanding it by adding new lambda functions. In the case of a matrix multiplication algorithm the feature <Number of iterations> can be renamed <Matrix Size>.

2.1 Variance of execution times for a given chunk-size

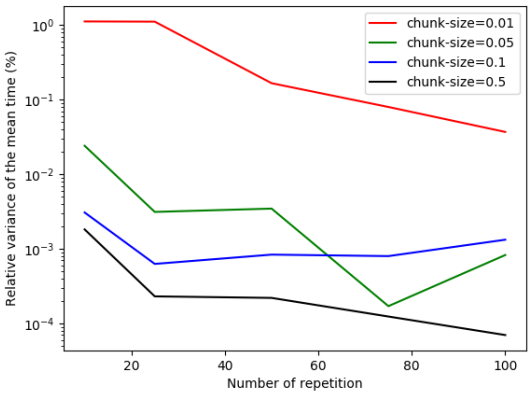
For a given experiment, you expect to get similar execution times for a given chunk size value every time you run it. There will always be some noise in the data since there are many processes that the user cannot directly control. To correct that, every for-loop is run N_{rep} times and the execution time is given by the mean of these repetitions. Each repetition can be represented by an index $i = 0, 1, 2, 3, 4, ...$So for an experiment j the execution time outputted is

$$\bar{t_j} = \frac{1}{N_{rep}} \sum_{i=1}^{N_{rep}} t_i$$

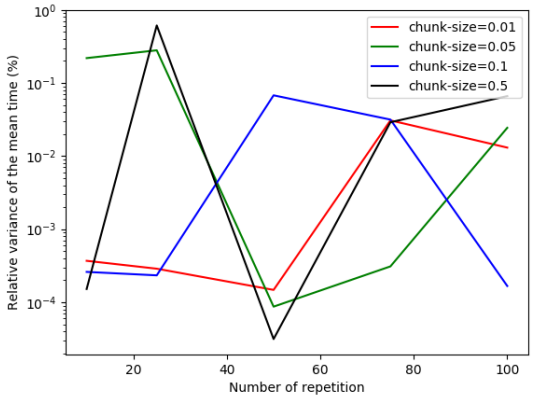
Note that the first time the function is run is ignored so this repetition could be given an index $i = 0$. This is done to ignore the time where the cache memory is being filed. One would expect the variance of the mean to decrease as the number of repetitions get bigger. If this could be confirmed experimentally, this would mean that by taking more and more repetitions, one can get more precise in execution time measurement and therefore the data generation process described earlier can be reliable. Studying the variance is a great tool to measure the variations of execution time between repetitions but it doesn't inform on how much the values vary relative to their size. To solve this issue, the relative variance is used :

$$RelVAR(\bar{t_j}) = \frac{VAR(\bar{t_j})}{\bar{t_j}^2} \times 100\%$$

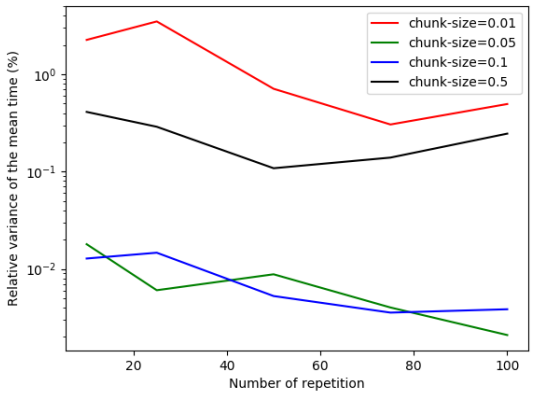
To test this assertion, the relative variance of the mean of execution times has been measured for values of $N_{rep} = 10, 25, 50, 75, 100$. these measurements have been done on matrix multiplications algorithms with different number of threads and different matrix sizes



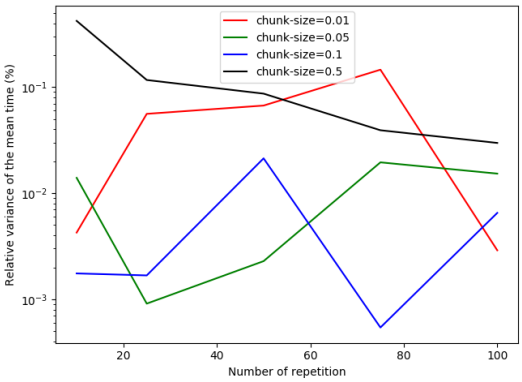
(a) 100x100 on 2 threads



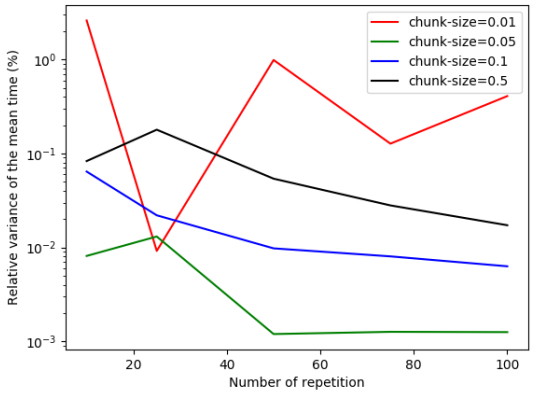
(b) 500x500 on 2 threads



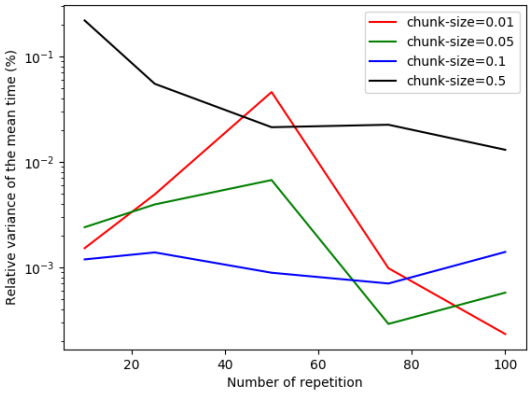
(c) 100x100 on 4 threads



(d) 500x500 on 4 threads



(e) 100x100 on 8 threads



(f) 500x500 on 8 threads

Figure 1 – The Variances of the mean of execution times for different experiments on Matrix Multiplication

From this analysis, we can see that in most cases for a 100x100 matrix the variance of the mean get smaller when N_{rep} get bigger which ultimately means that the more repetitions you do when generating your data, the more accurate your execution time measurement becomes. There are oscillations in some cases, mostly with 500x500 matrices, but the relative variances is always smaller or close to 1% (the maximal relative variance measure was 2%). Since the relative variance is always smaller or close to 1%,even for 10 repetitions, it was decided that 10 repetitions would be enough for this work. This means that every time I run any experiment, I measure the execution time of a HPX for-loop 10 times and output the mean. This choice was made to accelerate the data generation process but a larger number of repetitions can be used to generate more precise data at the cost of time.

2.2 Variations of optimal chunk size for a given experiment

Now that it has been demonstrated that you can accurately measure execution time by having more and more repetitions, one would expect the optimal chunk size to be the same each time you run the same experiment. This is important because the goal of the project is to predict the optimal chunk-size based on the features of a loop. If you run an experiment twice and get 2 different optimal chunk sizes, than it would mean that there is no function

$$f : \{Features\} \rightarrow Optimal\,Chunk\,Size$$

. It is important to be sure that such a function can exist because this is the function that will be approximated with machine learning. To analyze the variations in optimal chunk size, the same experiments as previous section have been run. Chunk sizes of {0.01,0.05,0.1,0.5} have once again been used on loops with 100 and 500 iterations with 2,4 and 8 threads. Here are the optimal chunk sizes for each experiment.Each collumn represents an experiment and each row represents a repetition on the given experiment.

Tableau 1 – Optimal Chunk size for each experiment

2 Threads		4 Threads		8 Threads	
100x100	500x500	100x100	500x500	100x100	500x500
0.5	0.5	0.5	0.01	0.05	0.5
0.5	0.5	0.1	0.05	0.05	0.05
0.5	0.5	0.1	0.05	0.05	0.5
0.5	0.5	0.1	0.05	0.01	0.01
0.5	0.5	0.1	0.05	0.05	0.01
0.5	0.5	0.1	0.05	0.05	0.01
0.5	0.5	0.1	0.05	0.05	0.01
0.5	0.5	0.5	0.05	0.05	0.01
0.5	0.5	0.1	0.01	0.01	0.01
0.5	0.5	0.1	0.5	0.05	0.01

As we can see there are some variations in optimal chunk size when you run the same

experiment different times and therefore we are not guaranteed to find the same target values when re-generating data. However, for these tests, it is obvious that those variations are not huge and we can clearly identify a value for chunk size that is selected more often than the other values. To conclude, the function $f : \{Features\} \rightarrow Optimal\,Chunk\,Size$ does, at first glance, seem to exist even if we may see some errors due to background noise or not optimal candidates.

2.3 Comparison of executions times for all chunk size candidates

As seen previously, when you run the same experiment multiple times, there is some variation in the optimal chunk size obtained. To understand this in more details, I decided to analyze the variations of execution times with respect to different chunk size candidates. In fact, for experiments where there is very little variations in executions times for different chunk-sizes, we should expect huge variations in optimal chunk-size since the times are so close that the optimal chunk size keeps changing.

To study the variations between execution times for different chunk sizes, the variance is once again used. Here the variance is calculated by the following for an experiment j :

$$VAR(t_j) = \frac{1}{Card(CS) - 1} \sum_{cs \in CS} (t(cs) - \bar{t}_j)^2$$

with

$$\bar{t}_j = \frac{1}{Card(CS)} \sum_{cs \in CS} t(cs)$$

Here time is a function of chunk-size so $t(cs)$ means the time for the chunk-size cs. $Card(CS)$ is the cardinality of the set CS which means how many candidates are being tested. Also, in that case the mean $\bar{t}_j$ is the average of executions times between all chunk sizes candidates. This variance can be interpreted as how much chunk size affect the execution time of an experiment. Small values of variance mean that there is no significant impact on execution times and a huge variance means that chunk size has a significant impact on the execution time.

The variance has been calculated on a Matrix Multiplication algorithm with 100,200,300,400, 500,600,700,800 iterations and 1,2,4,6,8,10,12,14 threads and $CS = \{0.01; 0.05; 0.1; 0.5\}$ Here the variance is shown using a color map.

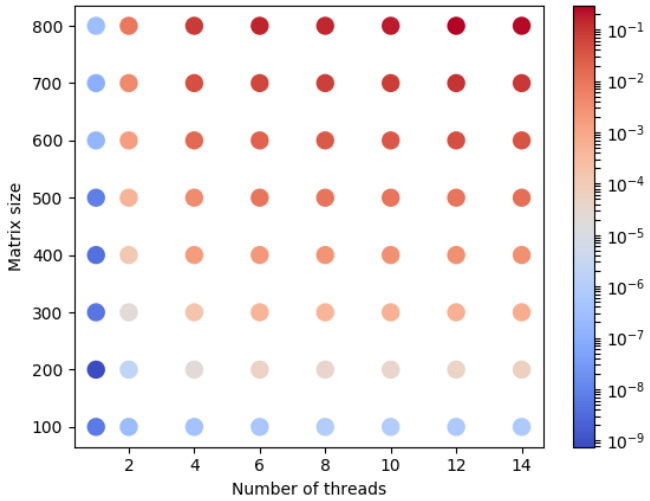


Figure 2 – Variance of execution times

It would seem, according to this result, that when you add more threads and more iterations, the more variations you observe when measuring execution time for different chunk-sizes. However there is a problem with this result, the execution times are not all on the same scale. In fact the more iterations you have, the bigger the execution time. In that case, the execution time ranges from 0.005 to 2 seconds depending on the experiment. To solve this problem, the relative variance must once again be used.

Here is a plot of the relative variance for all the same experiments :

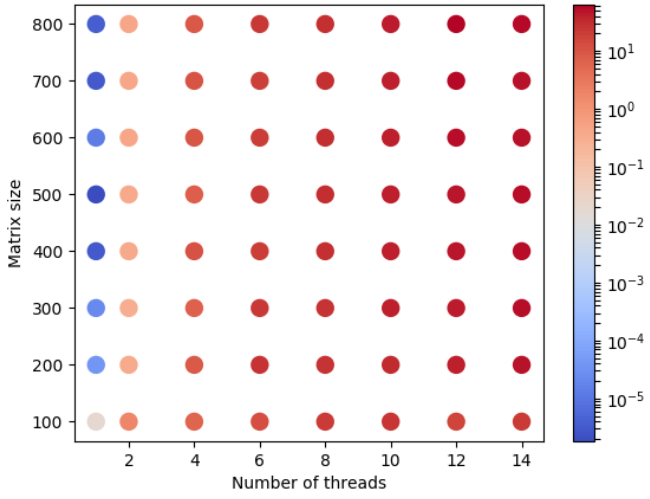


Figure 3 – Relative Variance (%) of execution times

As we can see, the relative variance depends highly on the number of threads. It appears

that the lowest variance is obtained when using only one threads which makes sense since 1 thread is equivalent to sequential execution. Therefore, it has been decided that the number of threads would never be put to 1 because of the low relative variance between execution times. Here is the same graph without the experiments with 1 thread.

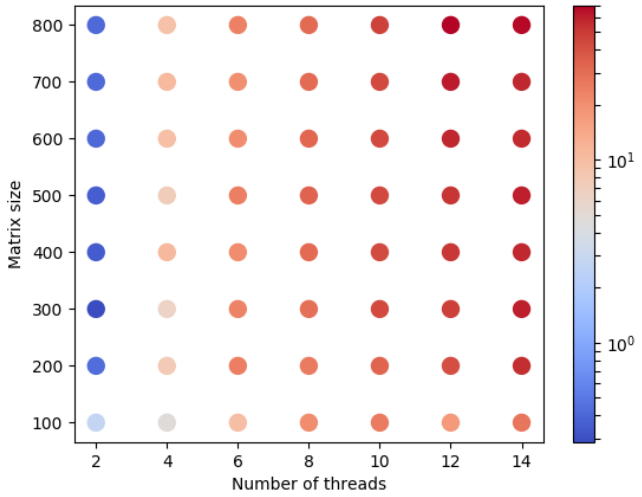


Figure 4 – Relative Variance of execution times

In this graph, we can see some slight dependence of relative variance with respect to Matrix size but it is still clear that number of threads has the biggest impact on relative variance.

2.4 Function analysis

In this section, the focus will be on analyzing the function that will be approximated by machine learning

$$f : \{Features\} \rightarrow Optimal\,Chunk\,Size$$

The data used to visualize this function is comprised 64 experiments of matrix multiplication with 100,200,300,400, 500,600,700,800 iterations and 1,2,4,6,8,10,12,14 threads. The chunk sizes candidates were $CS = \{0.005; 0.01; 0.05; 0.1; 0.2; 0.5\}$

First of all, it is important to note that since when generating data with only a matrix multiplication function, the features <Number of Operations> <Number of Float Operations> and <Number of Comparison Operations> are a polynomial function of <Number of Iterations> Here is an example where a second degree polynomial can be fitted with no error :

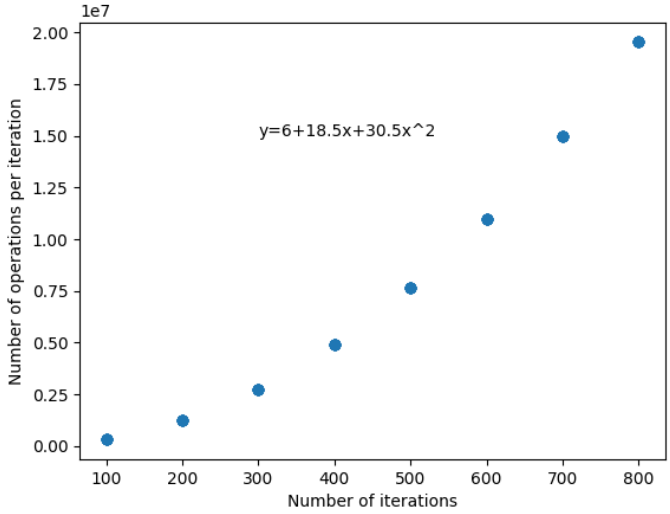


Figure 5 – Optimal Chunk Size with respect to Number of iterations and threads

This means that of all 6 features, only 2 can be used (Deepest loop level is not necessary because it is the same for each matrix multiplication experiment). Here is a punctual color-map of the function with the current data.

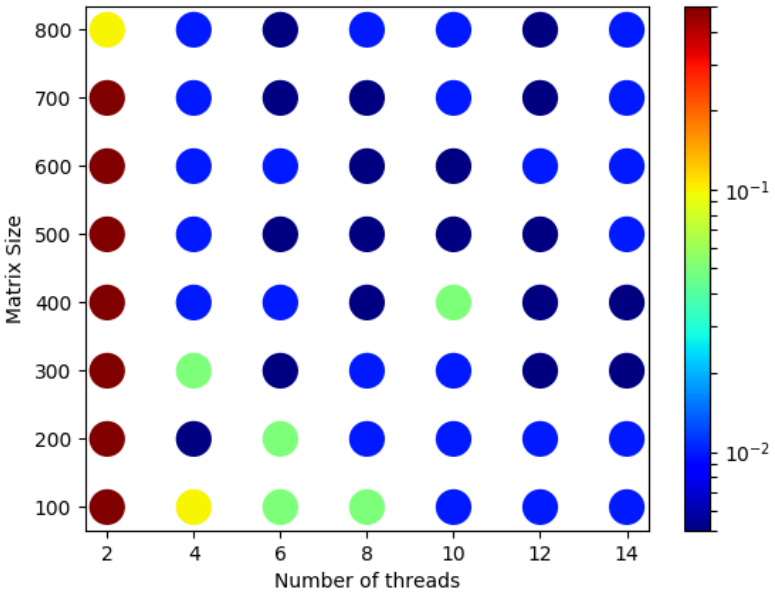


Figure 6 – Optimal chunk size

Number of threads has an impact on optimal chunk size For 100 iterations where we can see a steady decrease of chunk size with respect to number of treads. However for bigger

matrices, the function oscillates. These oscillations are caused by the fact that execution times between 0.005 and 0.01 are very close. Here is a table of some execution times to show my point. (Note 0.2 was remove because it was never selected)

Tableau 2 – Execution times for 5 experiments

0.005	0.01	0.05	0.1	0.5
0.221386	0.219532	0.247906	0.253246	0.629754
0.229569	0.229047	0.238892	0.422549	1.02871
0.168765	0.166738	0.2141	0.4157	1.0294
0.122448	0.121981	0.136015	0.260424	0.709034
0.0179531	0.0177467	0.0181951	0.0337951	0.082139

Here we can see than in all experiments, the execution times for 0.01 and 0.005 are very close. In fact the relative difference if around 1% in average. The oscillations could be caused by the fact that the small difference 0.005 0.01 is smaller than the noise. To have a better looking function, the noise must be reduced.

Here are some way we try to remove some noise.

1- 10 repetitions is not enough to have good results on execution times and therefore on optimal chunk size.

2-Matrices size is too small and therefore execution times for certain chunk sizes are too close to be reliably distinguishable.

3-There is a missing dynamic feature that is causing the variations which seem uncorrelated to the 2 current features.

4- Some chunk size candidates are too close to be reliably distinguished so the choice of candidates could be optimized to maximize the difference between their execution times.

2.4.1 First approach

The second hypothesis has been tested by doing 50 repetitions instead of 10.

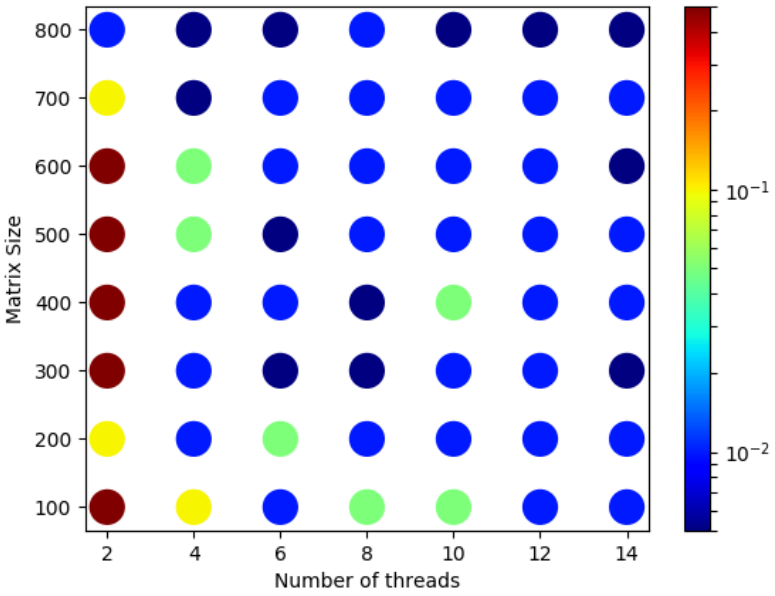


Figure 7 – Optimal Chunk Size with 50 repetitions

This seemed to help reduce some of the oscillations between 0.005 and 0.01 but they are still present.

2.4.2 Second approach

The third hypothesis was tested by using bigger matrices of sizes : 100,200,400,600,800,1000,1500, 2000,2500 with chunk-size candidates $CS = \{0.0005; 0.001; 0.005; 0.01; 0.05; 0.1; 0.5\}$

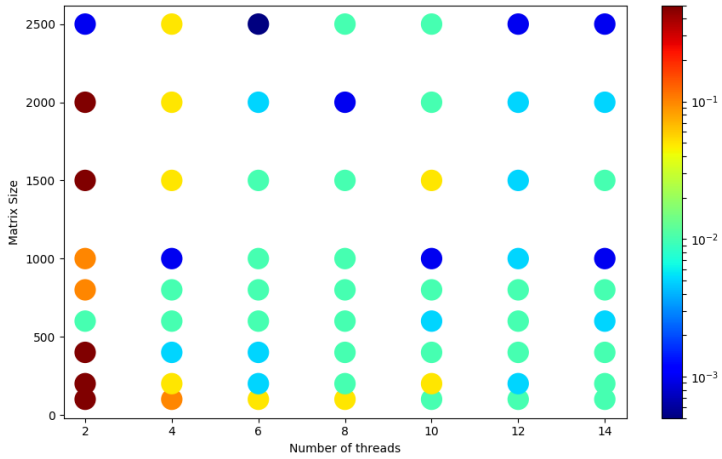


Figure 8 – Optimal Chunk Size with bigger matrices

We still observe oscillations so I decided to examine the execution times once more.

Tableau 3 – Execution times for 1000x1000, 1500x1500, 2000x2000 and 2500x2500

0.0005	0.001	0.005	0.01	0.05	0.1	0.5
NaN	1.06075	1.10638	1.01317	1.13454	1.15139	3.35216
NaN	3.33313	3.34309	3.33242	3.88524	3.93604	9.56364
7.91752	7.84541	7.8376	7.88513	9.22963	9.44594	22.4268
15.4564	15.5674	15.4928	15.4807	18.0824	18.2549	43.5778

It appears like once again, the execution times for the smaller chunk-sizes are very close, which explains the oscillations in the function. Having bigger matrices doesn't solve the problem because there will always be oscillations for very close chunk sizes.

2.4.3 Third approach

The idea would be to add the idle rate of the CPU's as a dynamic feature in the hope of having a better function. This hasn't been done yet.

2.4.4 Fourth approach

2.5 Machine-Learning

2.5.1 Arguments to use Regression

Now that the function has been analyzed, it is time to fit it using machine learning algorithms. There are many algorithms to fit functions which are divided into two categories (Classification and Regression)

Classification algorithms can output any value from a finite set of values. Using a classification algorithm to fit the function would mean that the algorithm can output any chunk-size from the set CS. This method was used in [0] and has proven to be successful on large data sets with $CS = \{0,001; 0,01; 0,1; 0,5\}$.

Regression algorithms can output any value from an infinite set which is usually a sub set of the Real Number line. For chunk size prediction, a well trained regression could output any real number from 0 to 0.5.

The advantage of regression over classification in chunk size prediction is that technically, you can output any value for chunk size. If classification is used, then you may possibly miss the best chunk size value if it is not in the set CS. As an example, Imagine an experiment that has an optimal chunk size of 0.025. If a classification algorithm is used, you run the risk

that 0.025 is not in your set of candidates CS and therefore you will not predict it. You will simply predict the best chunk size within your set which is not the overall best chunk size. A solution would be to have a very large set CS but this can quickly become unpractical if ...

By nature, a regression could be able to predict this value 0.025. In reality it would output something like 0.0249854687 which can than be manually rounded to something realistic like 0.025.

Another advantage of regression with respect to classification is that classification interprets the target values as classes and not numbers. Therefore the chunk sizes are treated as categorical instead of numerical. Regression would interpret the chunk sizes as a numerical variable which is closer to reality.

2.5.2 Regression Results

The 4 regression algorithms that were studied are Support Vector Regression, Neural Network Regression, k-Nearest-Neighbors Regression and Random Forest Regression. To compare the performance of these algorithms, the k-fold cross validation has been used. In this method, you divide your data-set into k subsets and than you train the regression on (k-1) subsets and use the last subset as a testing set on which the regression will be tested. The measure of the error will be done by scikit's $MeanSquaredError()$ which outputs the mean of the squared error between the test subset and the predictions.

$$MSE(f) = \frac{1}{N_{data}} \sum_{i=1}^{N_{data}} (y_i - f(x_i))^2$$

Where f represent the regression function,x_i represent the features for an experiment,y_i represents the target values and N_{data} represents the number of experiments in the data set. It is important to note that to train regression algorithms, you need target values which are on the same scale. To solve this issue a logarithmic scaling is used on the targets values. The chunk-size is obtained by applying the exponential function to the result of the regression. Here are some results when using the data used to generate figure [5]. A set of 64 experiments of matrix multiplication with sizes of 100,200,300,400, 500,600,700,800 and 1,2,4,6,8,10,12,14 threads.$CS = \{0.005; 0.01; 0.05; 0.1; 0.5\}$

Tableau 4 – Results on 64 experiments with logarithmic scaling and $CS = \{0.005; 0.01; 0.05; 0.1; 0.5\}$

	SVR	Neural Network	K-Nearest-Neighbors	Random Forest
k=4	0.0317+-0.038	0.032+-0.045	0.0186+-0.027	0.010+- 0.017
k=5	0.0318+-0.038	0.018+-0.045	0.014+-0.027	0.010+-0.017
k=6	0.0319+-0.038	0.0159+-0.045	0.019+-0.027	0.012+-0.017

By using this metric, we can see that the k-Nearest-Neighbors and the Random Forest are the two best algorithms. I believe this is cause by the fact that these functions are

piece-wise constant instead of being continuous like SVR and Neural-Network regression. However, the MSE is not a good metric to understand what happens locally when fitting the data. To better understand what happens locally, a graph of Ydata vs Yprediction can be used.

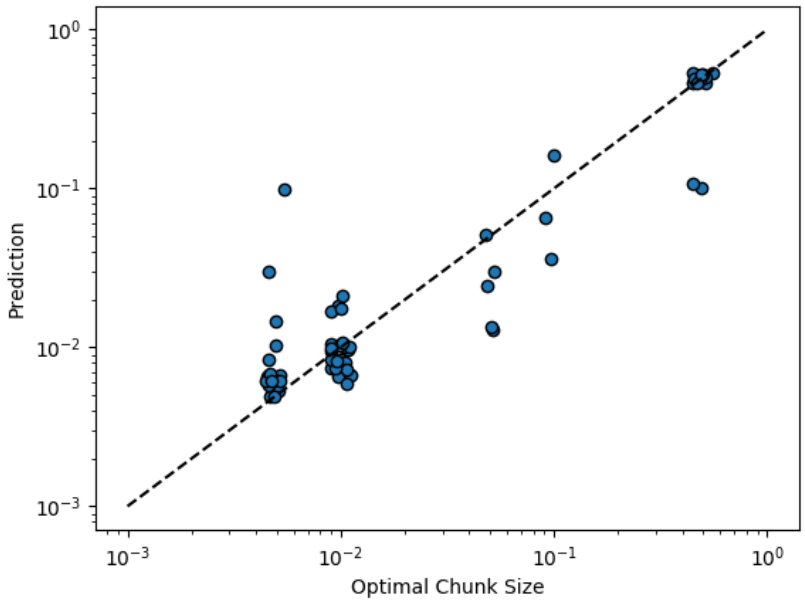


Figure 9 – Evaluation of Nearest-Neighbors

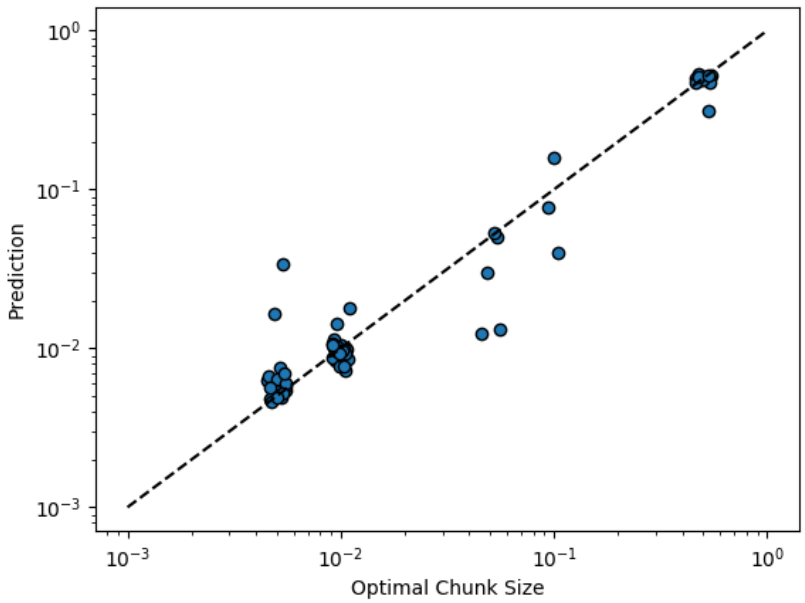


Figure 10 – Evaluation of random forest

We can see with these graphs that both algorithms are actually doing fine. The reason why the mean error is so high is because there are few instances where the prediction error is very huge, but these are not the majority of cases. In the majority of cases the prediction is pretty close to the actual value.

2.5.3 Bigger Matrices

Here are similar results with bigger matrices.

Tableau 5 – Results on Bigger Matrices with logarithmic scaling and $CS = \{0.0005; 0.001; 0.005; 0.01; 0.05; 0.1; 0.5\}$

	K-Nearest-Neighbors	Random Forest
k=4	0.0196+-0.086	0.018+- 0.078
k=5	0.020+-0.086	0.019+-0.078
k=6	0.02+-0.086	0.020+-0.078

Here we can see that the results for cross validation are almost identical for both algorithms. To study the result, here are the graphics of Reality vs Prediction

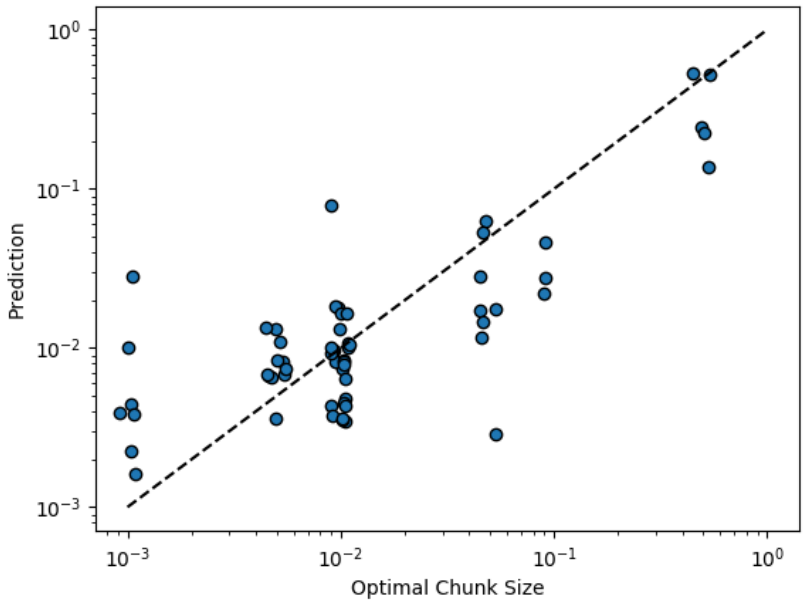


Figure 11 – Evaluation of Nearest-Neighbors on Bigger Matrices

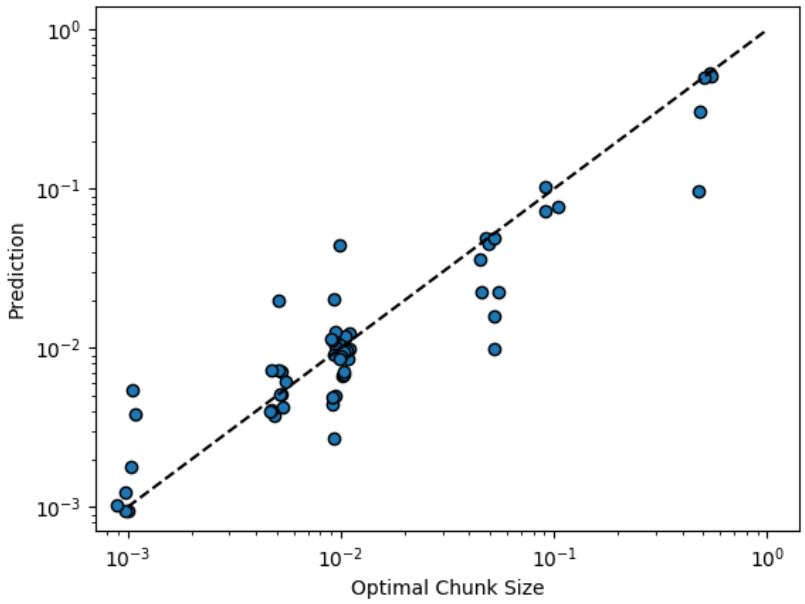


Figure 12 – Evaluation of random forest on Bigger Matrices

2.5.4 artificial noise reduction

I think that one way to improve prediction would be to get rid of the oscillations that were observed in the function at figures 6,7,8,9. This can artificially be done by removing candidates. A study of execution times shows that in figure [6], chunk sizes candidates 0.005 0.01 always have close execution times so we can try removing one of them.

Figures Soon

Even though random forest seems to perform a bit better than Nearest-Neighbors, I'm leaning toward Nearest-Neighbors because It's easier to implement and more intuitive.

...