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 ways we can achieve those goals is by using loop-level parallelism. If an algorithm is made of independent iterations, such a technique can be used. Using HPX's $for\_each()$ algorithm, the different iterations are assigned to threads and then to different CPU's. One of the key parameters of parallelization 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 all the notation used during the document.

1 Mathematical Notation

The ultimate goal of this research is to minimize the execution time of a parallel loop. Here is an example of a hpx for loop.

code example goes there

In such loops, a lambda function is applied over a range. The lambda function could be any algorithm like matrix multiplication per example. Each instance of a loop being run will be refered to as an experiment. Each experiment has an unique set of features which will be called X_i for the ith experiment. The following 6 features are used :

1 <Total Number of operations per iteration>

2 <Number of float operations per iteration>

3 <Number of comparison operations per iteration>

4 <Deepest loop level>

5 <Input size (range)>

6 <Number of threads>

The first 4 features will be referred as static since they only depend on the lambda function internal structure. They are collected at compile time by a ClangTool called loop convert.

The last 2 are called dynamic since they dont depend on the algorithm and are extracted at runtime.

If we assume that the variance in time measurement is small, we can say that time is a function of features and chunk size. If I describe the set of all possible features as $\{X_i\}$ and the set of all possible chunk sizes as CS than we have :

$$t : \{X_i\} \otimes CS \to \mathbb{R}$$

Our ultimate goal is to find the chunk size that minimizes the execution times for a given set of features. This means that we want to find the function :

$$f : \{X_i\} \to cs_i \tag{1}$$

where

$$cs_i = \underset{cs \in CS}{\arg\min}\ t(X_i, cs) \tag{2}$$

The value cs_i is known as optimal chunk size as it minimizes the execution time for experiment i. Optimal chunk sizes will be refered to as target values which is the vocabulary used in machine learning for values that we want to predict.

The objective is to use machine-learning algorithms to approximate the function f expressed in (1).

2 Data Description

The data that must be generated is comprised of features and target values. $(X_i, cs_i)\ \forall i$. To generate multiple experiments, one need to define lambda functions and apply them on a range inside a hpx for-loop.

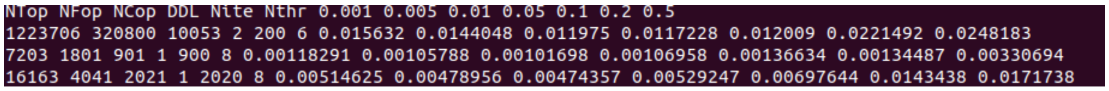
In practice, how does one find the target value for an experiment ? We can find the target value cs_i by applying equation [1], but finding the minima is impossible since the set CS of all possible chunk sizes is very large. The idea that was used in [0] was to restrict CS to a small set $CS = \{0.001; 0.01; 0.1; 0.5\}$. Each value in the set is refered to as a candidate. Having a small set means that you dont have the absolute best chunk size but an approximation of it. Even though its an approximation it will still be refered as cs_i Note that here the chunk size is a percentage which represent what fraction of the total number of iterations is given to a thread. To resume, generating target values consist of choosing a finite set CS of candidates and apply equation [1] which means evaluating the execution time for all candidates and taking the chunk-size with minimal time as target value for the experiment.

2.1 Data Generation

Data generation consist of running experiments and extract the values $(X_i, cs_i)\ \forall i$. the features of loops are extracted at compile and runtime and the execution time are measured for different chunk sizes candidates.

Here is an example of a data-set comprised of three different experiments and $CS = \{0.001; 0.005; 0.01; 0.05; 0.1; 0.2; 0.5\}$

Here is a pseudo code of how the data is generated.



```
NTop NFop NCop DDL Nite 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
```

```
For all experiments
    Extract static features using Loop Convert;
    N_candidates=5;
    For all chunk size candidates
        Time=0;
        For i = 0 to 10
            Time+=Time measured on hpx for-each loop with candidate;
        Time/=N_candidates;
```

3 Data Analysis

3.1 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 <Input-size> can be renamed <Matrix Size>.

3.1.1 Variance of execution times for a given chunk-size

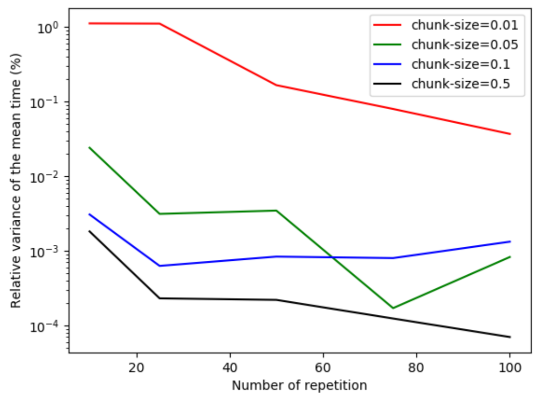
For a given experiment, you expect to get similar execution times for a given chunk size value every time you run it. There will always be some noise in the data since there are many processes that the user cannot directly control. To correct that, every for-loop is run N_{rep} times and the execution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

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

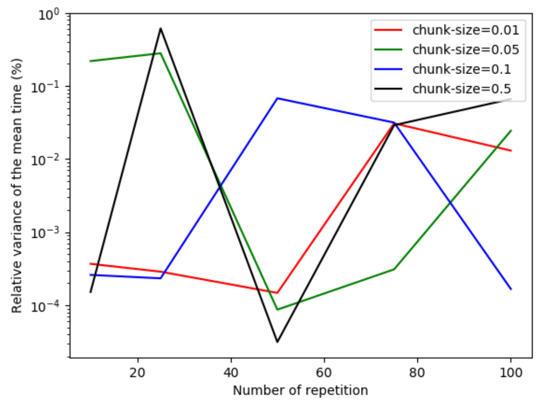
Note that the first repetition of the for-loop is ignored so this repetition could be given an index $j = 0$. This is done to ignore the time where the cache memory is being filled. One would expect the variance of the mean to decrease as the number of repetitions get bigger. If this could be confirmed 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 :

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

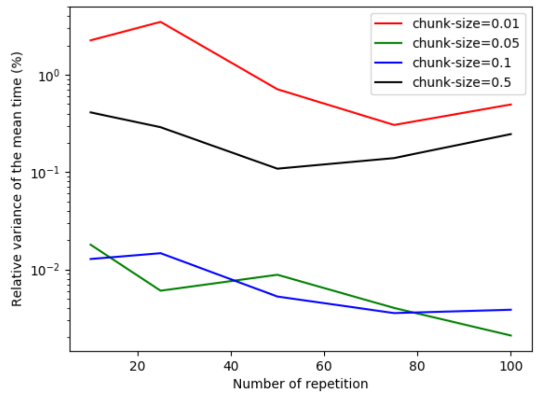
To test this assertion, the relative variance of the mean of execution times has been measured for values of $N_{rep} = 10, 25, 50, 75, 100$. These measurements have been done on matrix multiplications algorithms with different number of threads and different matrix sizes



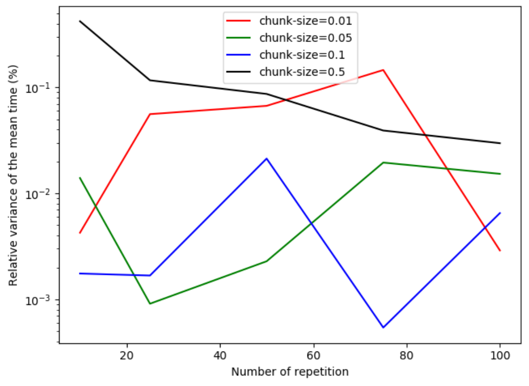
(a) 100x100 on 2 threads



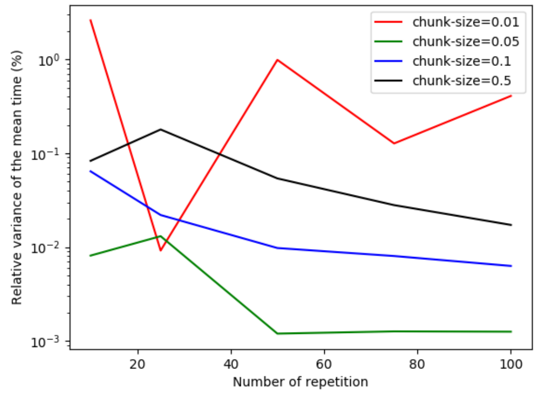
(b) 500x500 on 2 threads



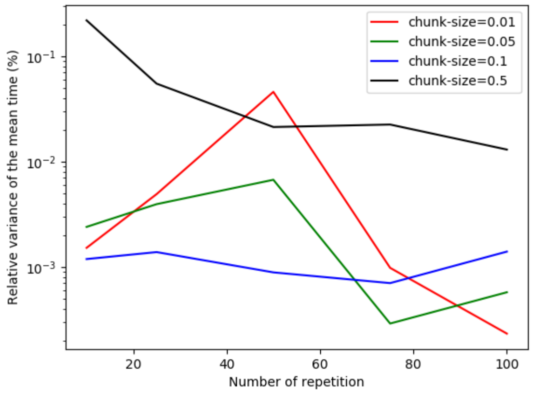
(c) 100x100 on 4 threads



(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.

3.1.2 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

$$f : \{X_i\} \rightarrow cs_i$$

. 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	
experiment	100x100	500x500	100x100	500x500	100x100	500x500
1	0.5	0.5	0.5	0.01	0.05	0.5
2	0.5	0.5	0.1	0.05	0.05	0.05
3	0.5	0.5	0.1	0.05	0.05	0.5
4	0.5	0.5	0.1	0.05	0.01	0.01
5	0.5	0.5	0.1	0.05	0.05	0.01
6	0.5	0.5	0.1	0.05	0.05	0.01
7	0.5	0.5	0.1	0.05	0.05	0.01
8	0.5	0.5	0.5	0.05	0.05	0.01
9	0.5	0.5	0.1	0.01	0.01	0.01
10	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, we can clearly identify a value for chunk size that is selected more often than the other values. To conclude, the function f does seem to exist but we can also observe some background noise.

3.1.3 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 i :

$$\mathrm{VAR}(t_i) = \frac{1}{Card(CS) - 1} \sum_{cs \in CS} (t(X_i, cs) - \bar{t_i})^2$$

with

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

$Card(CS)$ is the cardinality of the set CS which means how many candidates are being tested. Also, in that case the mean $\bar{t_i}$ 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.

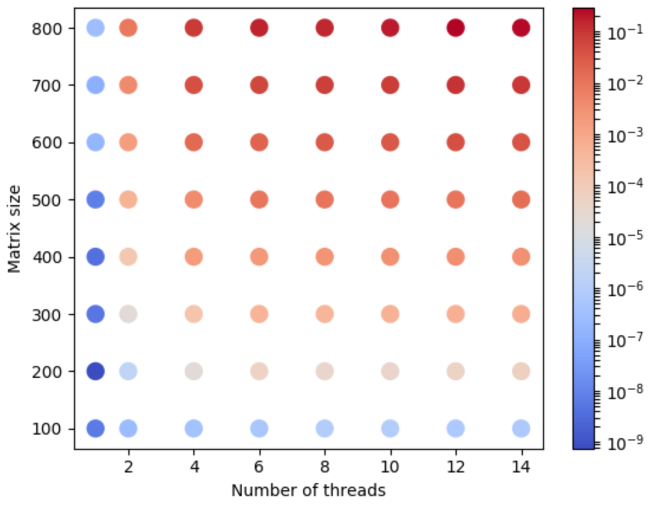


Figure 2 – Variance of execution times

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 :

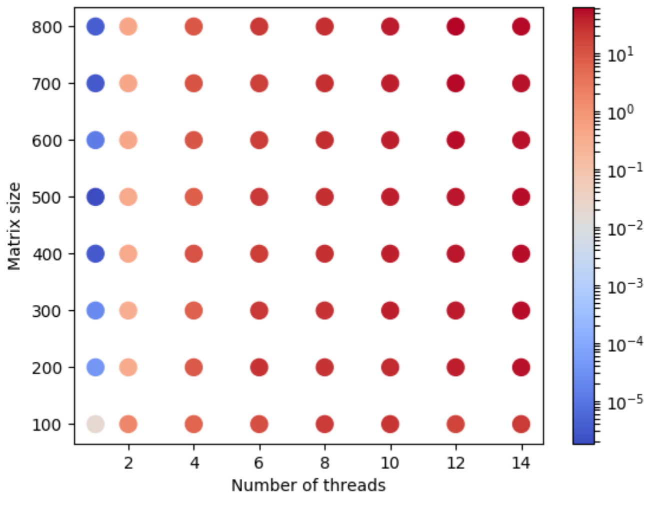


Figure 3 – Relative Variance (%) of execution times

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. We can also see that by adding more threads, we add more variances in the execution times because an efficient use of these threads results in a significantly smaller execution time.

3.1.4 Function analysis

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

$$f : \{X_i\} \rightarrow cs_i$$

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 :

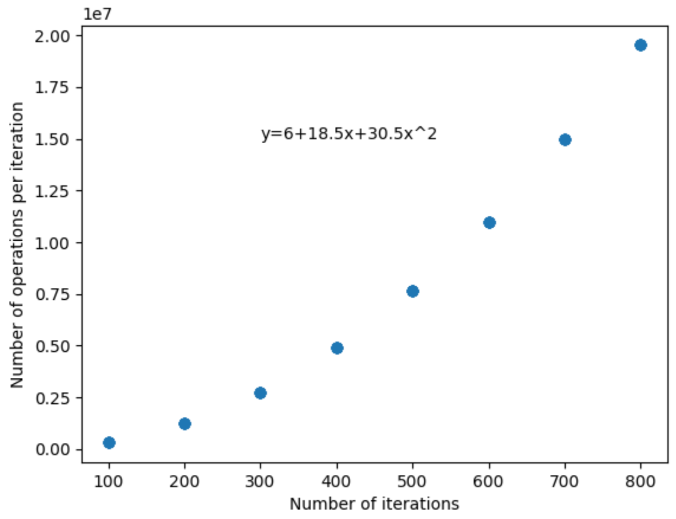


Figure 4 – Optimal Chunk Size with respect to Number of iterations and threads

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.

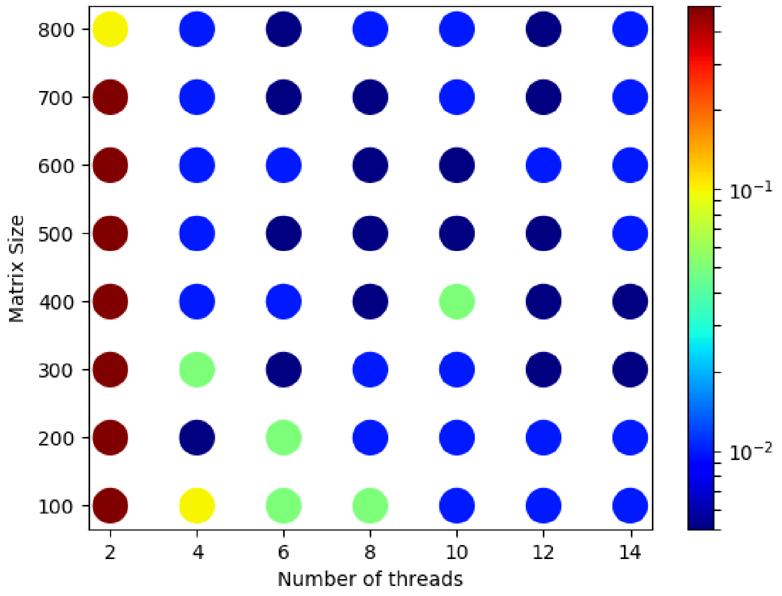


Figure 5 – Optimal chunk size

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 is around 1% in average. The oscillations are caused by the fact that the time difference between 0.005 and 0.01 is smaller than the noise.

Here are some way we try to remove the oscillations

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

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

3- Some chunk size candidates are too close to be reliably distinguished so the choice of candidates could be optimized.

3.1.5 First approach

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

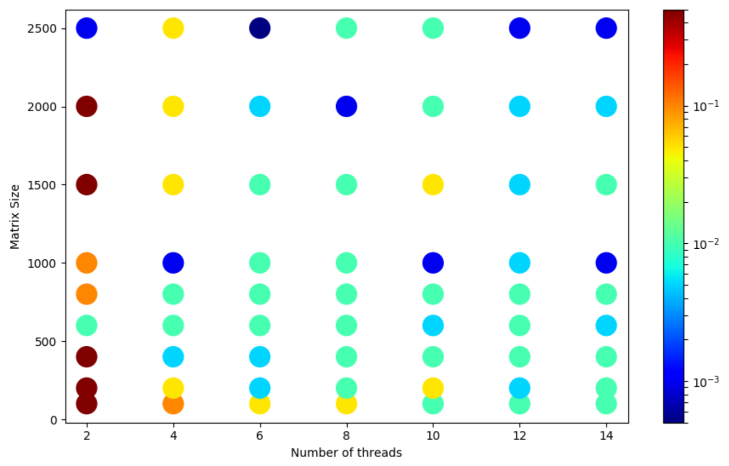


Figure 6 – Optimal Chunk Size with bigger matrices

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 didn't seem to solve the issue.

3.1.6 Second 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.

3.1.7 Third approach

By looking at table 2 and 3, it seems that chunk sizes which are different by a factor of two give very similar execution times while chunk sizes with a factor of 5 or more gave bigger differences. Here I will try to have a set of chunk size candidates where all consecutive chunk sizes are separated by the same multiplicative factor. First let's try with a factor 4. $CS = \{0.5; 0.125; 0.03125; 0.0078125; 0.001953125\}$. This means that the job will be split into 2,8,32,128,512 parts.

Here is for small matrices

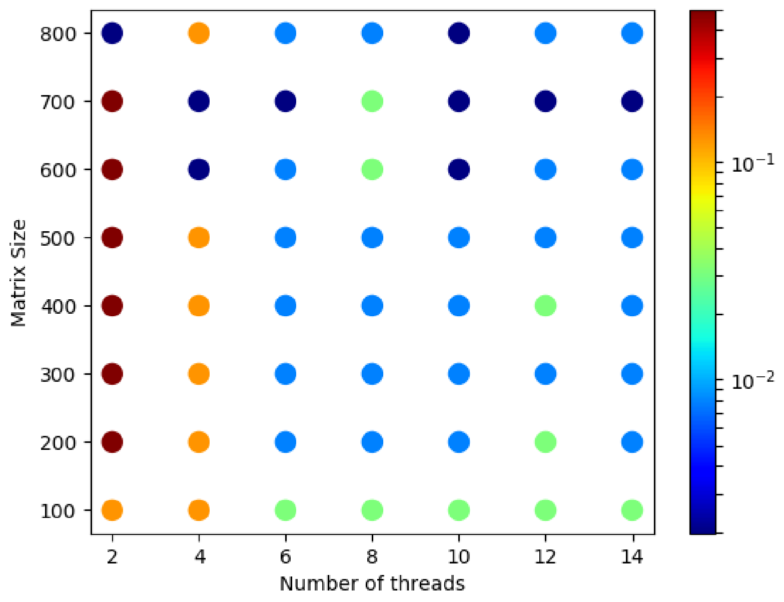


Figure 7 – Optimal Chunk Size for small matrices

And for bigger matrices

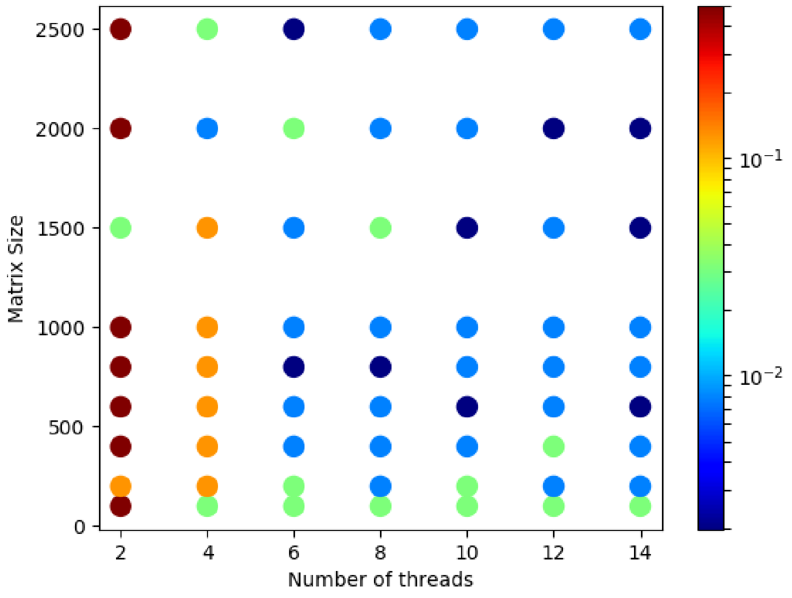


Figure 8 – Optimal Chunk Size with bigger matrices

that result is very encouraging, We can see some reduction of the oscillations. The choice of candidates is still not optimal but we can see that choosing candidates is really important. Here is the results with 50 repetitions

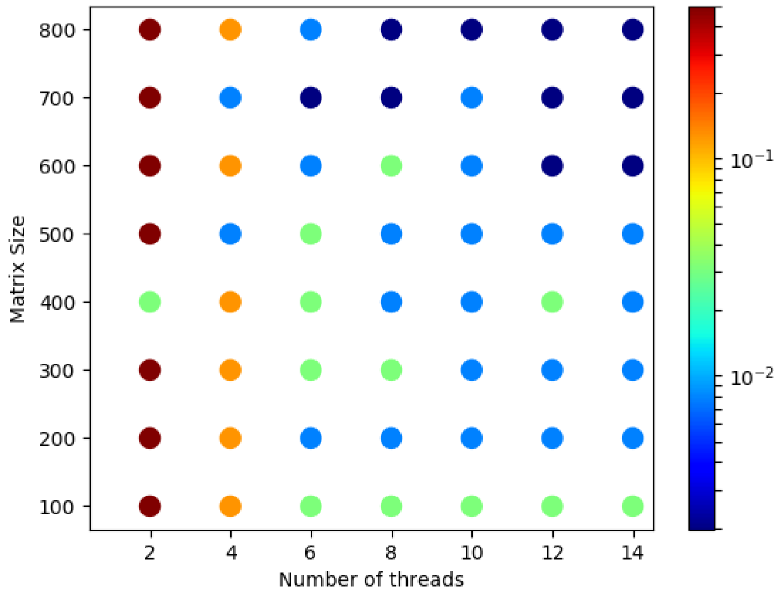


Figure 9 – Optimal Chunk Size with small matrices and 50 repetitions

This result is closer to what you would expect, the more threads and the bigger the Matrix, the more advantageous it becomes to split your job in many chunks. There is still some noise but it is unavoidable at this point. At least we can reduce it by having the right set of candidates.

3.2 Bigger Data Set

In this section, the focus will be on a bigger data set which is comprised of 278 experiments with many lambda functions. Here is a descriptions of all the functions used :

Name	**Description**	**Numbers of itera**
Nothing()	Does Nothing	10^7
Swap()	Swap elements between two vectors	100,1000,1000 100000,100000
Stream()	Basic stream algorithm	1000,10000,1000
Matrix_Vector_Mult()	Matrix Vector multiplication	100,500,1000 5000,10000,200
Dyadic()	Applies a dyadic product on two vectors	100,1000,1000
Cosine()	Applies a cosine function on each element of a given vector using taylor series	100,1000,1000 100000
Matrix_Matrix_Mult()	Matrix Matrix multiplication	100,500,1000,15 2000,2500
Max()	Find the max of each collumn of a 3 dimensionnal array	1000,5000,1000 50000,100000
Tensor_Generator()	Generates a tensor of dimension 4	100,200

Due to the difficulty of analying functions of more than 2-3 dimensions, this function has not been analysed yet. However at some point it should be analysed as this data-set is closer to what will be used in practice.

4 Machine-Learning

4.1 Classification or Regression

In machine learning there are two type of functions approximators, classifications and regressions. Classification can output any value from a finite set and this method was previously used with $CS = \{0.001; 0.01; 0.1; 0.5\}$. Regression can output any number from a infinite set of candidates so it can be seen as classification algorithm where the number of candidates grows to infinity. But one has to wonder, Is it better to use less or more chunk sizes candidates. In [0] chunk sizes of $\{0.01; 0.01; 0.1; 0.5\}$ have proven to be succesfull but can we improve performance by adding more candidates ? If the variance of time is null than we have :

$$CS \subset CS' \Rightarrow \min_{cs \in CS'} t(X_i, cs) \leq \min_{cs \in CS} t(X_i, cs) \ \forall i$$

Which is to say that adding new candidates should always give a smaller or equal minimum of time. However, as seen earlier, there is variance in time measurement but we can hope that the variance is small enough to ensure that the statement is still true. This is my main hypothesis that will be tested :

HYPOTHESIS : Adding more candidates to CS will reduce the execution times of hpx-loops.

One way to test this assertion is to try a classification algorithm and adding more than 4 candidates but this becomes unpractical as generating data for many candidates takes longer. As an example data with 8 candidates takes around twice as long to generate as data with 4 cadidates Also, classification algorithms tends to reduce in accuracy the more candidates you add. However, if a regression algorithm is used, we technically have a set CS which is infinite, even if we have only 4 or 5 candidates in the training set. This is because regression can interpolate between two candidates. For example if the candidates used when training are $\{0.01; 0.1; 0.5\}$ a regression could output any value in between like 0.0245879097 or 0.3333453 instead of being restricted to the finite set $\{0.01; 0.1; 0.5\}$. So let me rephrase the hypothesis :

HYPOTHESIS(rephrase) : Using a regression algorithms will give smaller execution times than classification algorithms.

The goal of this analysis will be to confirm or infirm this hypothesis.

4.2 Matrix Multiplication dataset

The 4 regression algorithms that were studied on the matrix multiplication dataset 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 evaluated. 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.

$$MSE(\hat{f}) = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} (y_i - \hat{f}(X_i))^2$$

Where $\hat{f}$ represent the regression function which is an approximation of f, X_i represent the features for an experiment i, cs_i represents the target values and N_{test} represents the number of experiments in the test 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. This means that the error expressed will be the error of the logarithm. This is very hard to visualize but the goal is to compare models so we simply need to compare the errors. 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 2,4,6,8,10,12,14 threads.$CS = \{0.5; 0.125; 0.03125; 0.0078125; 0.001953125\}$

Tableau 4 – MSE on 64 experiments with logarithmic scaling

	SVR	Neural Network	K-Nearest-Neighbors	Random Forest
k=10	1.78+-1.14	1.72+-0.91	1.28+-1.31	1.55+-1.

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. In such graphs, cross validation is once again used but now the prediction are outputted. Each color represent a different test set.

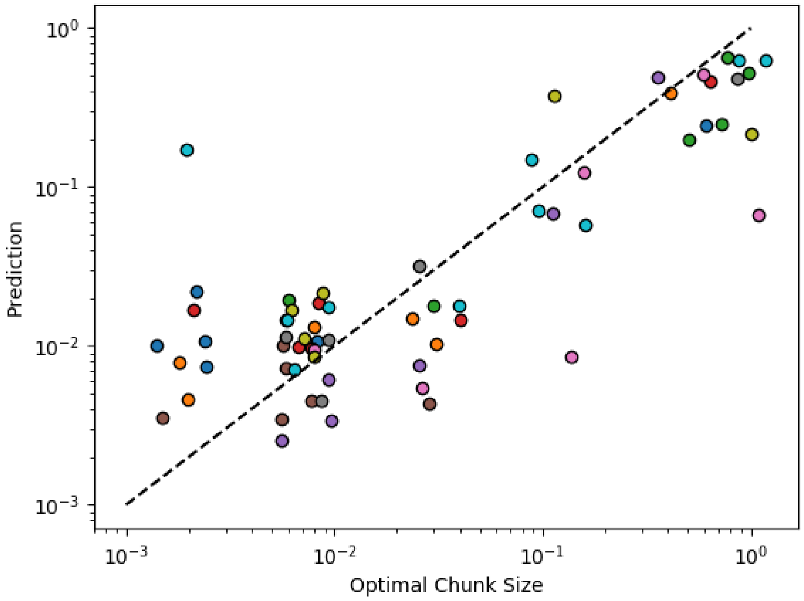


Figure 10 – Evaluation of Nearest-Neighbors

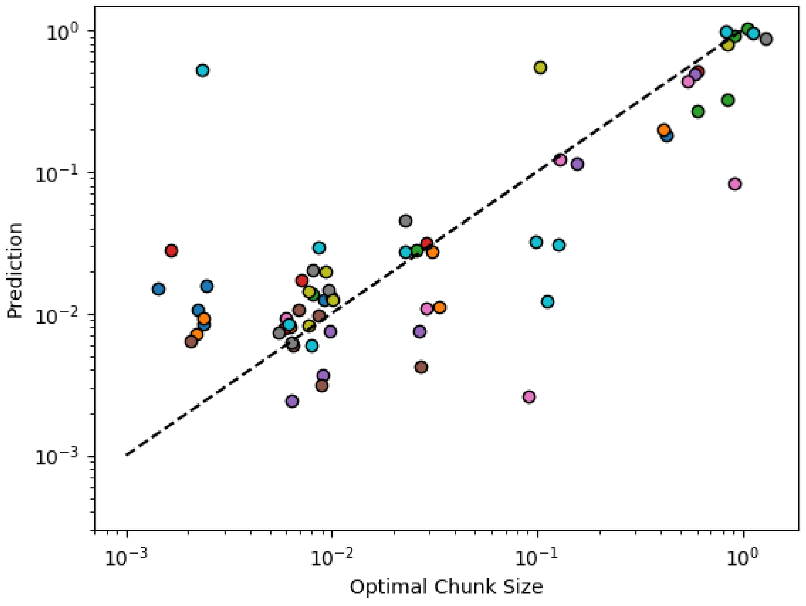


Figure 11 – Evaluation of random forest

We can see with these graphs that both algorithms are having similar results. However, calculating the error on chunk size is one thing but how does this error translate to execution time ? The following measure I will call *Approximated_Total_Time*() will be used :

$$Approximated\_Total\_time(\hat{f}) = \sum_{i=0}^{N_{exp}} t(X_i, \hat{f}(X_i))$$

where i represent a given experiment, $\hat{f}(x_i)$ is the predicted chunk size. To make this measure, the predictions on the experiments are obtained in python using k-fold cross validation, then the predictions are used as chunk-sizes on hpx for-each loops. This is an approximation because it doesn't include the time used to make the predictions and because the regressions are not fully implemented in hpx. Even though it's not fully reprensentative of the performance of these algorthms if they were implemented in hpx, this measure is still a useful tool to see if interpolating between chunk-size candidates really improves performances. To verify the hypothesis, 2 regression algorithms were compared to the multinomial classification algorithm which is already fully implemented in hpx.

Tableau 5 – Total execution time on 64 experiments with logarithmic scaling

	K-Nearest-Neighbors	Random Forest	Multinomial Classification
k=10	10.07+-.16	9.95 s+-0.09	11.99+-0.4

Here we can see that both algorithm regression have similar execution times. In fact, the difference between their execution time is lower than the incertanty of the measurements and therefore we cannot conclude that one is better. However we can see that they both perform better than the Multinomial Classification.

Graph to explain difference between classification and regression. Here the difference in execution time is represented by colors.

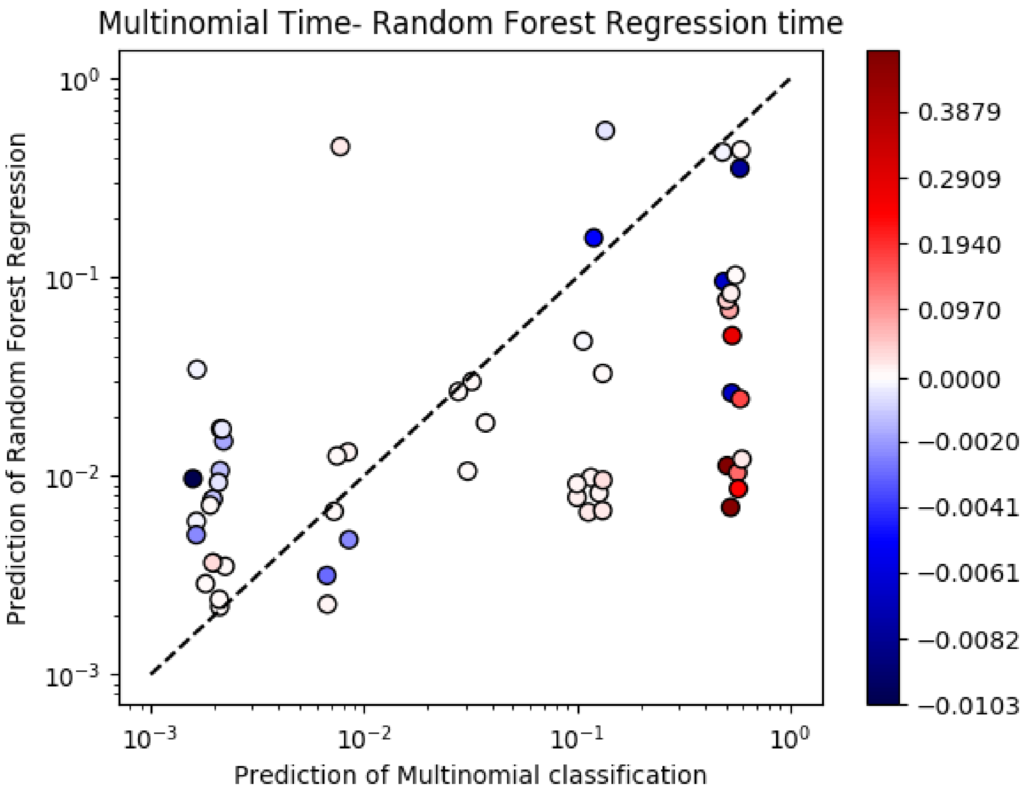


Figure 12

Here we actually see that the reason the regression works better is that, the multinomial predicts 0.5 when the optimal chunk size is actually smaller. In fact, because KNN and Random Forest regression take average over points in the feature space, these algorithms are less likely to predict high chunk sizes like 0.5.

CONCLUSION 1- KNN and Random Forest regressions are less likely to predict high chunk sizes like 0.5. Which improves performance.

Resulting from this analysis, it seems that predicting high chunk sizes like 0.5 is more risky than predicting small ones. This is because of the following statement :

$$cs > \frac{1}{\text{number of threads}} \Rightarrow \text{missclassification error}$$

When you have $cs > \frac{1}{\text{number of threads}}$, it means that some CPU's are inactive since you dont divide your job into enough chunks to split to all available CPU's. Also in figure [3] it is shown that there is a smaller variance with respect to chunk size on 2 threads. This means that the reward for rightly predicting a 0.5 on 2 threads is smaller than the loss for wronly predicting a 0.5 on more than 2 threads. This missclassification can be manually corrected after the multinomial classification prediction by reducing the chunk size until we have $cs \leq \frac{1}{\text{number of threads}}$. The chunk size obtained may not be the best one, but at least all the CPU's will be active. This will be refered to as *Multinomial Classification corrected*

Tableau 6 – Total execution time on 64 experiments with logarithmic scaling

	K-Nearest-Neighbors	Random Forest	Multinomial Classification Corrected
k=10	10.07+-0.16	9.95 s+-0.09	10.18+-0.09

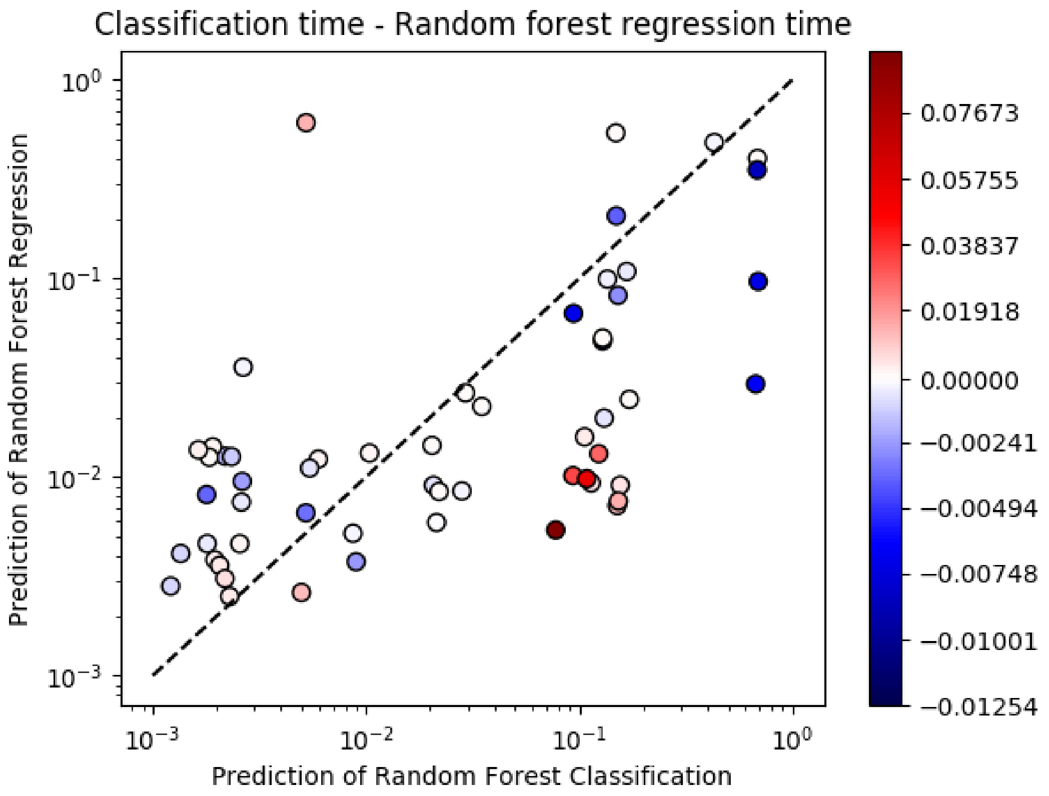


Figure 13 – Evaluation of random forest

Here we can note that by manually reducing the aberant chunk sizes has improved performances but we can still see that the classification has a longuer execution times in some instances. These instance happen to be the ones where the chunk size was manually reduced. However, it cannot be manually reduced any further because we don't know what is the best chunk size. The time difference for these errors is now smaller It is important to note that the color blue is not signiticant in this graph since its magnitude is around the variance of the data. This can be said because there are blue dots on the doted-line.

4.3 Big Data Set

Here on the big data set, the Mean Square Error has once again been measured on the 2 selected regressions :

Tableau 7 – Mean Squared Error on 278 experiments with logarithmic scaling

	K-Nearest-Neighbors	Random Forest
k=10	2.18+-1.1	1.5+- 0.45

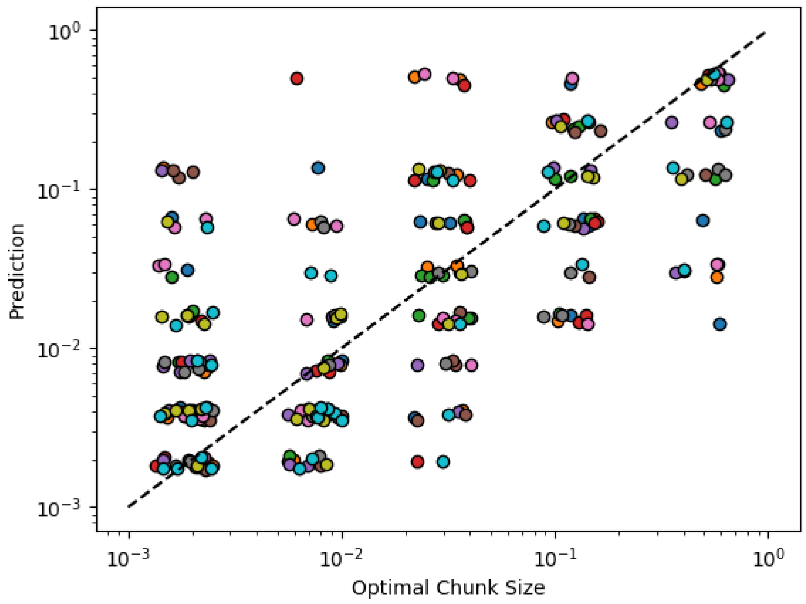


Figure 14 – Evaluation of Nearest-Neighbors with 2 neighbors

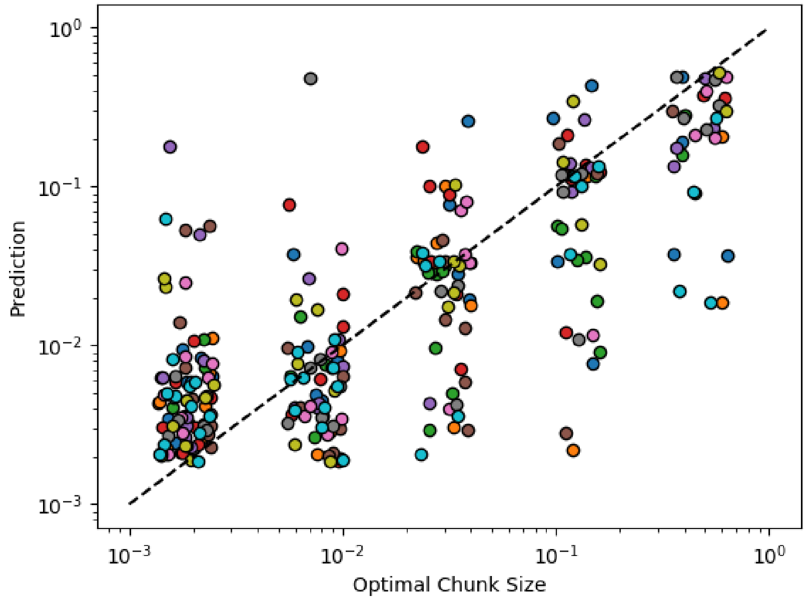


Figure 15 – Evaluation of random forest

The number of neighbors has been set to 2 because it was the value that minimized the MSE. However it is still quite big for k-Nearest-Neighbors relative to Random Forest. I believe it performs worse in that scenario because of the preprocessing on the big data set. In fact not normalizing the features gave a MSE of 1.67. It is normal that kNN is sensible to preprocessing and not RandomForest because it calculates Euclidian distance between points in the feature space. However I have decided to normalize the data nontheless because the features are on different scales. The exact reason why kNN performs better without normalization could be studied on its own.

Now let's see how this error in prediction relates to execution times. Once again the regressions are compared to Multinomial Classification :

Tableau 8 – time Total time on 278 experiments with logarithmic scaling

	K-Nearest-Neighbors	Random Forest	Multinomial Classification
k=10	352+-3.5	341+- 3.6	344+-4.2

We can once again see that kNN is not doing great which is not surprising considering the big MSE is provided. Random Forest is slightly faster than Multinomial Classification but not by a significant ammount as the two intervals of confidences are supperposed.

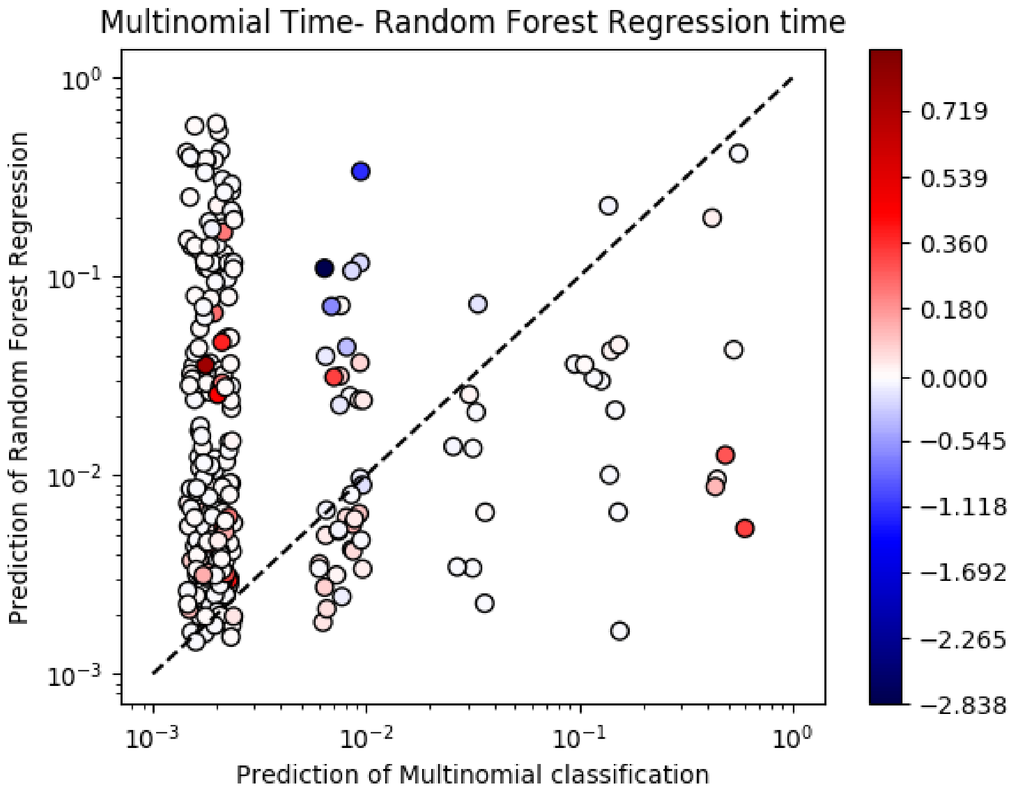


Figure 16 – ...

Once again, we see that the Classification predicted 0.5 when the best chunk size was smaller. The Multinomial Classification Corrected can once again be used :

Tableau 9 – Approximated Total time on 278 experiments with logarithmic scaling

	K-Nearest-Neighbors	Random Forest	Multinomial Classification Corrected
k=10	352+-3.5	341+- 3.6	342+-3.56

graph

we can see that thi data et introduced new types of error.

STENCIL BIG INPUT SIZES put to 10000000 or higher