Prediction of High-Performance Computing Input/Output Variability and Its Application to Optimization for System Configurations.

Edward Eckle

Department of Computer Science
California State University, Los Angeles

May 01, 2024

Table of Contents

Abstract

This study addresses the crucial aspect of performance variability in High-Performance Computing (HPC) systems, exploring its intricate relationship with factors like IO threads and IO scheduler. Despite the complexity, statistical methodologies are employed to predict HPC variability, an underexplored area in engineering HPC systems. This paper simulates tests done in the research paper of the same name written by Li Xu, Thomas Lux, Tyler Chang, Bo Li, Yili Hong, Layne Watson, Ali Butt, Danfeng Yao, and Kirk Cameron. Adjustments have been made to accommodate for the Delta Supercomputer being used as the primary focus of these tests. Through extensive experimentation, a robust data analytic framework is developed, leveraging various predictive models tailored to HPC variability. Performance evaluation across interpolation and extrapolation scenarios underscores the efficacy of these methods, making way for their applicability in practical settings. Furthermore, a systematic methodology for optimizing system configurations based on estimated variability maps is proposed, offering valuable insights for HPC variability management practices. The study's findings enhance one's understanding of statistical methods in HPC variability management, providing valuable tools for optimizing system performance and reliability.

Introduction

High-performance computing (HPC) systems, while immensely powerful, suffer from performance variability, particularly in input/output (IO) throughput. This variability can significantly impact scientific workflows. This research investigates the potential of statistical techniques and experiments using surrogate models to predict and manage IO throughput variability in HPC systems. Our aim is to assess the suitability of these methods for practical application and provide valuable insights for incorporating them into HPC variability management strategies.

This is a significant problem, because the ability to manage and predict performance variability in HPC systems can contribute to many issues. These can include reproducibility of scientific findings. Inaccuracies that result due to unexpected performance fluctuations can be detrimental to scientific progress, therefore consistent IO throughput is crucial for ensuring reproducible results on HPC systems. Unpredictable performance can also lead to inefficient allocation of resources. By managing this variability the optimization of resources can be utilized to potentially reduce overall execution time for tasks. More frugally there's also the cost efficiency of these systems to take into account by minimizing variability in IO throughput. There can be better planning and scheduling of tasks reducing energy and overall computational costs.

In this paper we will be comparing the results of another paper which focused on this issue. We attempt to replicate the results as well as come up with our own unique solutions to their problem. We work around the varying challenges based on our configuration to theirs.

Background

This paper deals with the optimization of system variability and performance regarding file IO. At points the authors discuss concepts such as variability surface and performance surfaces, approximation methods, surrogates models, and so on. We will discuss these here for clarity sake.

Surrogate models are created based on the data collected by the authors to interpolate and extrapolate values that were not collected experimentally. This is done using a number of methods, most simply a linear regression. These surrogate models are based on the same inputs.

Simply written as x, the inputs to the features as described in the paper are the system configurations. x being the system's number of threads, cpu frequency, file size, and record size. It serves as the input to both variability and performance models.

Performance and surface models are based on similar inputs. The performance models are based on the raw inputs and throughput, while the variability models are based on the unique configurations of x, and the variability of those groups various throughput values.

Optimization Problem

In the paper the authors note that the original objective function for this problem can be written as:

$$\min_{\boldsymbol{x}\in\mathcal{D}} \widehat{f}(\boldsymbol{x}), \quad \text{subject to} \quad \widehat{m}(\boldsymbol{x}) \geq m_0.$$

In plain English this can be read as saying, to minimize the variability of the system we are subject to the performance of the system as long as it's greater than or equal to some minimum performance requirement. We will note that the function f is the variability surface and the function m is a performance surface. These are both created with surrogate models based on the data collected, while m_0 is the minimum performance requirement.

$$\min_{\boldsymbol{x}\in\mathcal{D},\ z} \widehat{f}(\boldsymbol{x}), \quad \text{subject to} \quad m_0 - \widehat{m}(\boldsymbol{x}) + z^2 = 0.$$

It's a better organized function. The authors introduce a slack variable to remove the inequality. Using an augmented Lagrange method the function is simplified to.

$$L(\boldsymbol{x}, z, u, c) = \widehat{f}(\boldsymbol{x}) + u\left[m_0 - \widehat{m}(\boldsymbol{x}) + z^2\right] + \frac{c}{2}\left[m_0 - \widehat{m}(\boldsymbol{x}) + z^2\right]^2,$$

$$L_z(\boldsymbol{x}, u, c) = \widehat{f}(\boldsymbol{x}) + \frac{1}{2c}\max\{0, u + c[m_0 - \widehat{m}(\boldsymbol{x})]\}^2 - u^2.$$

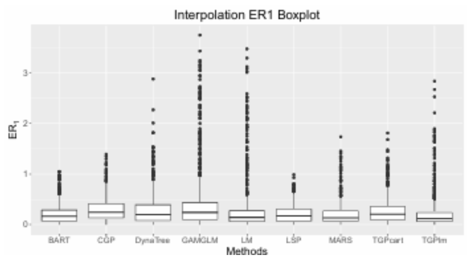
We convert this in Section 6 to python code to optimize directly.

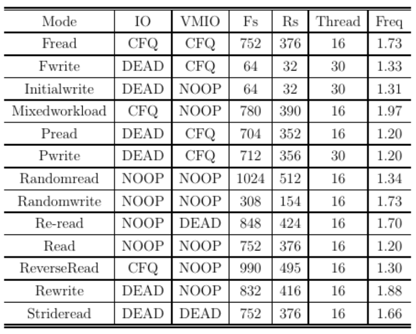
Methodology

The authors set up the VTDIRECT95 algorithm by defining the objective function, specifying variable ranges, as well as setting any constraints. They configured the algorithm parameters such as termination criteria. VTDIRECT95 operates by dividing the search space into smaller regions and evaluating the objective function at designated points. It combines deterministic and stochastic methods to effectively search for the global minimum of the objective function. In HPC systems management, VTDIRECT95 was used to determine the best system configurations that minimize performance variability and satisfy performance standards. The algorithm continues to run until it finds an optimal solution, or until it meets the specified termination criteria.

To achieve this the authors create different models based on different mathematical formulas using collected data from their benchmarks with IOzone. Using the collected data they create models that predict the variability of the systems as well as the performance in throughput. Next, they measure the standard deviation of these models and later, after creating an objective function, they use them to minimize the global maxima to find the most optimal parameters and highest throughput of those models. This is all done so as to ultimately achieve the least variability and highest throughput of those models.

Results and Discussion





Mode	IO	VMIO	Fs	Rs	Thread	Freq
Fread	CFQ	CFQ	752	376	16	1.73
Fwrite	DEAD	CFQ	64	32	30	1.33
Initialwrite	DEAD	NOOP	64	32	30	1.31
Mixedworkload	CFQ	NOOP	780	390	16	1.97
Pread	DEAD	CFQ	704	352	16	1.20
Pwrite	DEAD	CFQ	712	356	30	1.20
Randomread	NOOP	NOOP	1024	512	16	1.34
Randomwrite	NOOP	NOOP	308	154	16	1.73
Re-read	NOOP	DEAD	848	424	16	1.70
Read	NOOP	NOOP	752	376	16	1.20
ReverseRead	CFQ	NOOP	990	495	16	1.30
Rewrite	DEAD	NOOP	832	416	16	1.88
Strideread	DEAD	DEAD	752	376	16	1.66

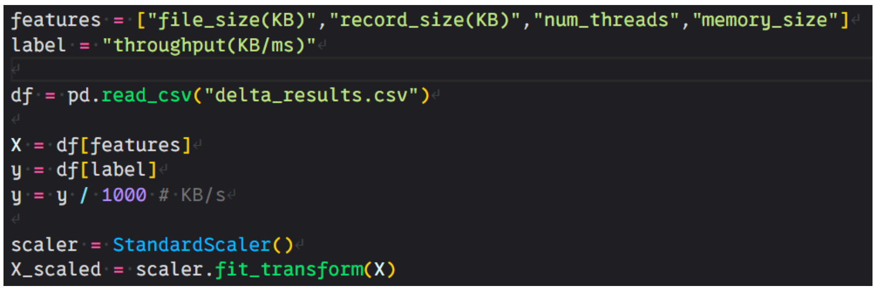
Shown above, the results from the original paper, we can see the box plot of the interpolation results of the variability models and to the other side we can see the results of the optimization problem. These results allow us to see which of the models are most effective when solving for the variability of the model and the results on the right show us our expected values for what we should be getting for our file size, record size, and thread count. This model interpolation box plot allows us to focus on finding implementations in python that are the most accurate, as shown on this plot. Using this form of hindsight we can focus our algorithms and models to more quickly find an accurate solution. The authors note that while they can predict variability on the system they are not quite able to reveal the root cause of the performance variability, so there is room for research in this area.

Your Implementation

Before beginning to implement the optimization we first have to consider how we are going to be collecting the data to be optimized or rather, to create the models that will be optimized. When reading the paper we notice that the four key parameters that are modified to create these models are the file size, the record size, number of threads, and the CPU frequency. We immediately notice that three of these are possible on Delta, however it is not possible to modify the CPU frequency on delta so for this reason we have to discard this parameter. We acknowledge this results in a reduction of the dimensionality of our input, however, we notice that Delta can modify the RAM size, so instead, we introduce this as our 4th parameter. This then solves the first part of our problem. We can create a script which instances an interactive session on Delta with differing numbers of threads and memory and then, in our software, we will modify the file and record size.

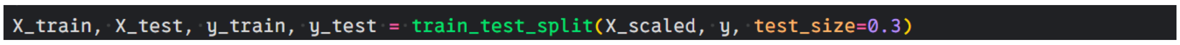
The next problem that occurs is that the original authors use a software called IOzone. This is a benchmarking software that is not available on Delta. We will write our own software that can mimic this functionality. In order to do this we create our own program in C. This is a simple program that reads and writes random data to a file depending on the file and record size. We benchmark the throughput of the operations. It is written in C because we want to reduce as much overhead as possible so that we can achieve similar results to that of the paper. We also consider that the somewhat simple C code is not quite as sophisticated as the benchmarking tool used by the authors so we keep this in mind when we look at our results. This software can be found in our repository under the name “benchmark.c”. After we had finished running this software on Delta in multiple different hardware configurations we received a final file simply called delta_results.csv. This raw file can be found in a repository. This file is a comma separated values file; it contains the following fields: operation, file size, record size, num threads, memory size, time, and throughput. It is important to note that our throughput is measured in kilobytes per millisecond.

With our data collected we can begin to process the data as well as create the models that we will use to optimize and find a solution for our problem. We achieve all of this inside of Python in a Jupyter notebook that can also be found in our code repository. We use a number of libraries, the most important ones to note are the libraries csv, numpy, pandas, scikit-learn, and matplotlib.



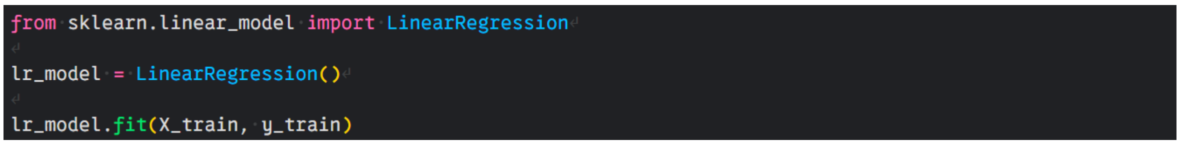
```
features = ["file_size(KB)","record_size(KB)","num_threads","memory_size"]
label = "throughput(KB/ms)"

df = pd.read_csv("delta_results.csv")

X = df[features]
y = df[label]
y = y / 1000 # KB/s

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

We preprocessed the data by separating the features and the labels of the data from the CSV file into a pandas data frame. The specific features being the file size, the record size, the number of threads, and the memory size; while the label is the throughput. We also have to divide our throughput by 1000 since our raw value is in kilobytes per millisecond when we want kilobytes per second. Next we'll take the features and scale them using the scikit-learn standard scaler. The authors choose to take the log directly of the features rather than to use a scaler. In our tests, however, we choose to use a scaler since it automates the process and may ultimately be more precise. Now we will split our data into training testing sets.



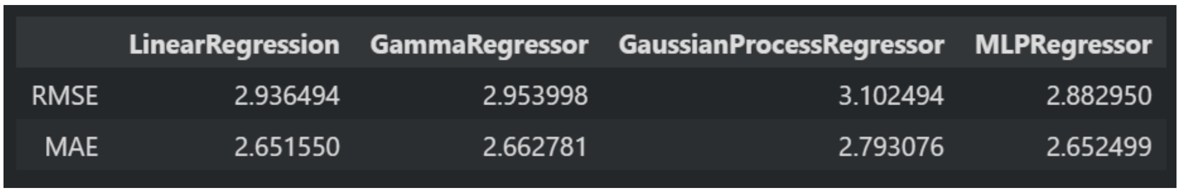
```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3)
```

With our data prepared we can build our first set of models. Keep in mind that currently our data is with respect to the throughput of the system, therefore any model we build will be a performance model. Further on we will discuss building the variability models. In the paper the authors build numerous different models of those we are able to recreate: a Linear Regression model, a Gamma GLM regressor, and a Gaussian process regressor. We also additionally add a model not used by the author's creators. We create a multilayer perceptron regressor more colloquially known as a neural network. Scikit-learn exposes an API for all of these making them very easy to create rapidly. The code below shows the process of creating one of these. The process to create the other 3 are very similar.



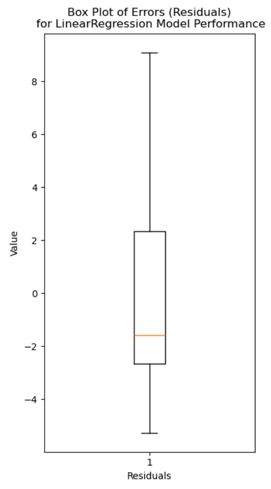
```
from sklearn.linear_model import LinearRegression

lr_model = LinearRegression()

lr_model.fit(X_train, y_train)
```

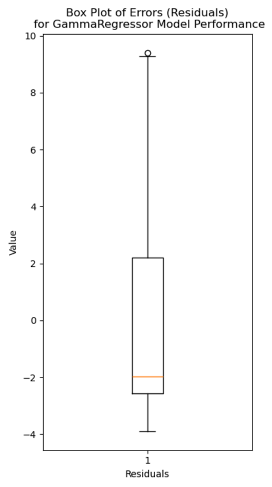
Next we will evaluate the models. We do this by creating predictions for each model based on the testing set and then we will compare the results of the testing sets with the true ground truth values from the testing data. We want to keep the evaluation methods we use as close to the paper as possible, so we will do the root mean square error as well as the mean absolute error.

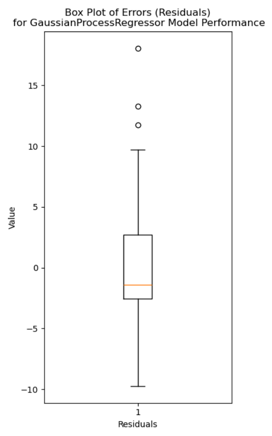


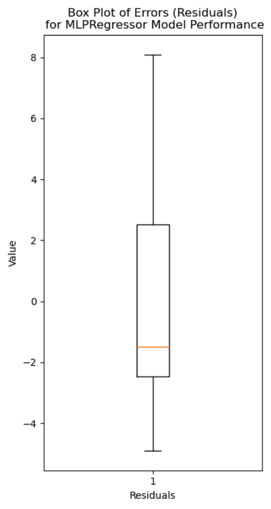
	LinearRegression	GammaRegressor	GaussianProcessRegressor	MLPRegressor
RMSE	2.936494	2.953998	3.102494	2.882950
MAE	2.651550	2.662781	2.793076	2.652499

We also calculate the residual error to create box plots of each of these models.

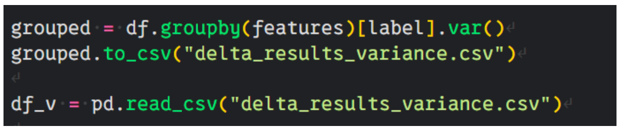






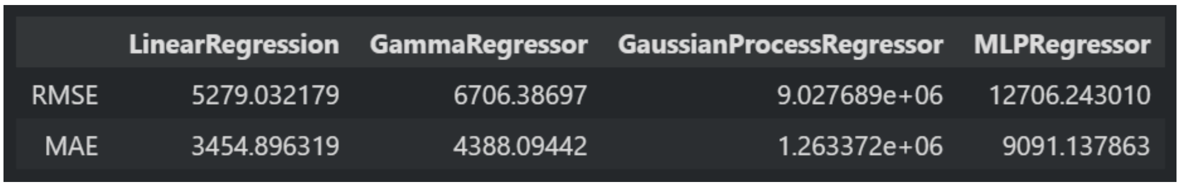


We will need to now create models for the variability of our data. To do this we will simply repeat the first step in preprocessing our data. However, in order to calculate the variability, what we do is group each of our features keeping the file size, record size, number of threads, and memory size the same. We then take the results where they differ and calculate the variability of each of those groups. (As you may notice in the code a new CSV file is generated containing the variance. This file will also be found in our repository.)

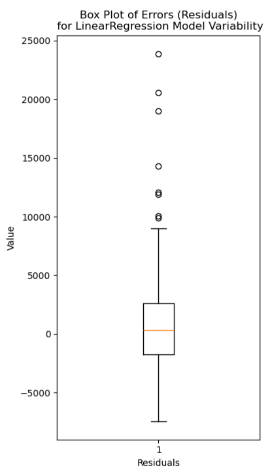


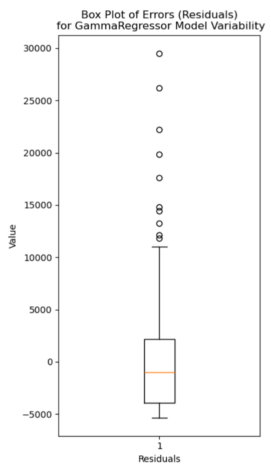
```
grouped = df.groupby(features)[label].var()
grouped.to_csv("delta_results_variance.csv")

df_v = pd.read_csv("delta_results_variance.csv")
```

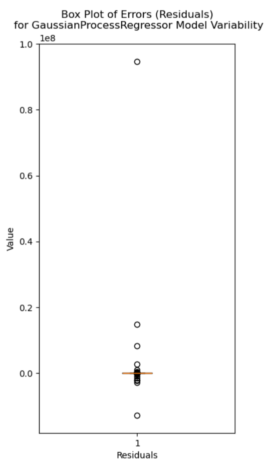
We can now repeat the same steps as above creating our testing and training sets, as well as training our models. We then can make our predictions for the root mean square error and the main absolute error of each of the models. We will also generate box plots for these variability models.

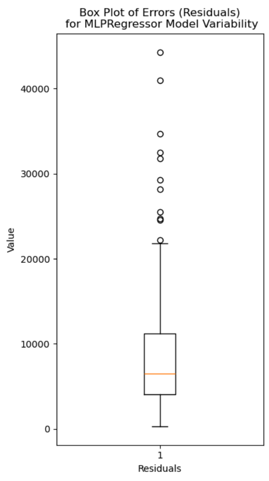


	LinearRegression	GammaRegressor	GaussianProcessRegressor	MLPRegressor
RMSE	5279.032179	6706.38697	9.027689e+06	12706.243010
MAE	3454.896319	4388.09442	1.263372e+06	9091.137863



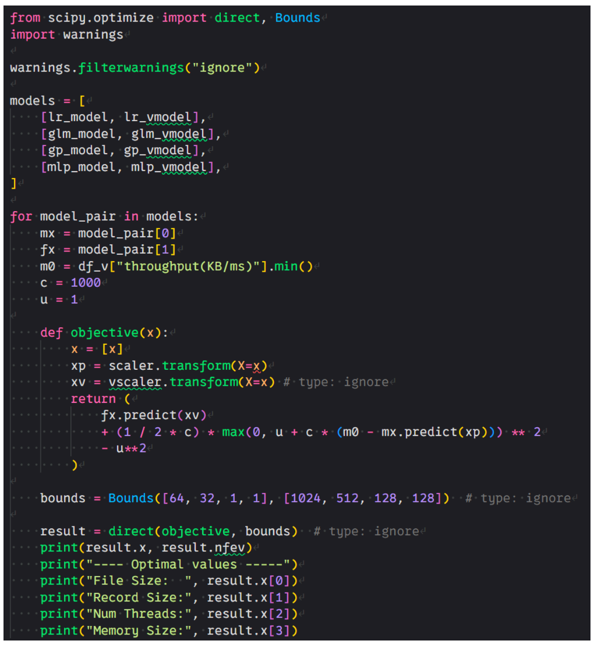






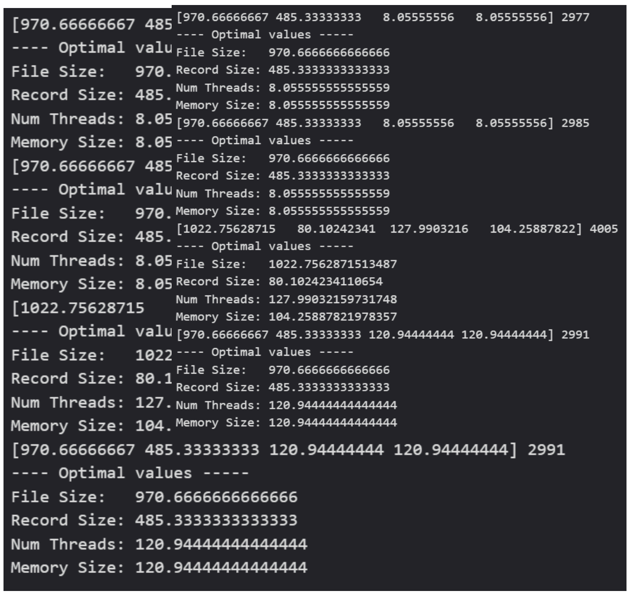
With both sets of models created we can now attempt to write and solve the optimization problem. We will not go into too much detail about the objective function itself since it is described earlier in this paper however, we will describe the code and why we made decisions that we did.

Firstly, we are able to use a scipy implementation of DIRECT found as scipy.optimize.direct. We are provided with another implementation called scipydirect but it seems as though DIRECT has already been implemented into scipy itself, so this separate package is no longer needed. In the paper the authors only analyze a single model. They choose what's called a linear shepherd model which unfortunately we were not able to find an implementation for. However, unlike the paper, we will optimize all of our models instead of a single one. We will then compare the results of each of them. Firstly, we store all of our model pairs. Next we will wrap our optimization function, bounds, and call to our optimization function in a loop that iterates


```python
from scipy.optimize import direct, Bounds
import warnings

warnings.filterwarnings("ignore")

models = [
    [lr_model, lr_vmodel],
    [glm_model, glm_vmodel],
    [gp_model, gp_vmodel],
    [mlp_model, mlp_vmodel],
]

for model_pair in models:
    mx = model_pair[0]
    fx = model_pair[1]
    m0 = df_v["throughput(KB/ms)"].min()
    c = 1000
    u = 1

    def objective(x):
        x = [x]
        xp = scaler.transform(X=x)
        xv = vscaler.transform(X=x) # type: ignore
        return (
            fx.predict(xv)
            + (1 / 2 * c) * max(0, u + c * (m0 - mx.predict(xp))) ** 2
            - u**2
        )

    bounds = Bounds([64, 32, 1, 1], [1024, 512, 128, 128])  # type: ignore

    result = direct(objective, bounds)  # type: ignore
    print(result.x, result.nfev)
    print("---- Optimal values -----")
    print("File Size:  ", result.x[0])
    print("Record Size:", result.x[1])
    print("Num Threads:", result.x[2])
    print("Memory Size:", result.x[3])
```

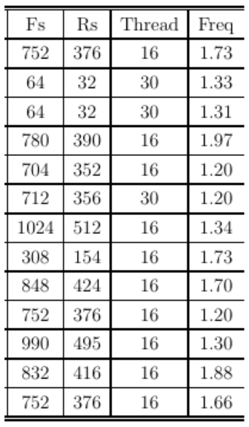
through these model pairs. Next set the parameters of our model. We set the performance surface to our performance model and the variability surface to our variability model. Two of the parameters are set as they were in the paper. Next we take our inputs and scale them using the same scalers we developed earlier. We do this because our models are trained on scaled values not the raw values. Finally we straightforwardly implement the optimization function as python code. The majority of the time is spent designing the models, data collection, and the benchmark code. Our bounds are set accordingly, we simply use the minimum and maximum values of our data.

Visible are the results of each model's optimization. We can see that there is a bit of a disagreement between each of the models. We must keep in mind the values should be interpreted as integers since floating point values cannot exist for file, record size or number of threads. System memory can, in theory, be less than a gigabyte in size but it's a bit unusual. We won't consider that here. Keeping this in mind, when we look at the results we see that the file size is roughly all on or around 1000 or what would be 1024 in a system. Record size is also nearly 512 in 3 of 4, with one outlier being that the neural network chose around 64 bytes instead. The two models on top chose around 8 threads and 8 gigs of memory while the two on the bottom chose 128 threads and gigs of memory. So which of these should we consider to be most accurate? To decide this we look at the results of our error calculations. When it comes to performance our smallest



```
[970.66666667 485.33333333   8.05555556   8.05555556] 2977
---- Optimal values -----
File Size:   970.6666666666666
Record Size: 485.3333333333333
Num Threads: 8.055555555555559
Memory Size: 8.055555555555559
[970.66666667 485.33333333   8.05555556   8.05555556] 2985
---- Optimal values -----
File Size:   970.6666666666666
Record Size: 485.3333333333333
Num Threads: 8.055555555555559
Memory Size: 8.055555555555559
[1022.75628715   80.10242341  127.9903216   104.25887822] 4005
---- Optimal values -----
File Size:   1022.7562871513487
Record Size: 80.1024234110654
Num Threads: 127.99032159731748
Memory Size: 104.25887821978357
[970.66666667 485.33333333 120.94444444 120.94444444] 2991
---- Optimal values -----
File Size:   970.6666666666666
Record Size: 485.3333333333333
Num Threads: 120.94444444444444
Memory Size: 120.94444444444444
[970.66666667 485.33333333 120.94444444 120.94444444] 2991
---- Optimal values -----
File Size:   970.6666666666666
Record Size: 485.3333333333333
Num Threads: 120.94444444444444
Memory Size: 120.94444444444444
```

error model is the MLP regressor, however, its variability is much higher so we won't choose this one. The gaussian process model has a variability error that is quite awful so we also will not use this one. Given that the other two models are very close in their results we can somewhat safely decide that those two are the best optimization solution. Translating the values given, we would have optimal settings as: file size 1024KB, record size 512KB, number threads 8, memory size 8GB. Comparing our values to the values found by the authors we can see that there are some similarities. This is regarding the file and record size. Most of their files are over 512. Most of their record sizes are over 256. Interestingly, all their Thread counts are around 16, which is double that of ours. It makes intuitive sense that the record size and the file size would closely affect the speed and variability of IO. On the other hand it's not intuitive why threads would make such a difference. One expects that writing or reading to a file would be a single threaded operation. It makes sense that the frequency would have an impact, since the number of clock cycles it might take would be faster or slower depending on how fast the frequency was. Unfortunately, we were not able to test this.



Fs	Rs	Thread	Freq
752	376	16	1.73
64	32	30	1.33
64	32	30	1.31
780	390	16	1.97
704	352	16	1.20
712	356	30	1.20
1024	512	16	1.34
308	154	16	1.73
848	424	16	1.70
752	376	16	1.20
990	495	16	1.30
832	416	16	1.88
752	376	16	1.66

Bibliography

[1] Xu, L., Lux, T., Chang, T., Li, B., Hong, Y., Watson, L., ... Cameron, K. (2021). Prediction of high-performance computing input/output variability and its application to optimization for system configurations. *Quality Engineering*