# How Deep is Your I/O? Toward Practical Large-Scale I/O Optimization via Machine Learning Methods

Robert Sisneros[*], Jonghoon J Kim[*], Mohammad Raji[†] and Kalyana Chadalavada[‡]
[*]*National Center for Supercomputing Applications*
*University of Illinois Urbana-Champaign, Urbana, Illinois 61801*
*Email: {sisneros,jkm}@illinois.edu*
[†]*University of Tennessee at Knoxville, Knoxville, Tennessee 37996*
*Email: mahmadza@vols.utk.edu*
[‡]*Intel Corporation, Folsom, California 95630*
*Email: kalyana.chadalavada@intel.com*

***Abstract*—Performance-related diagnostic data routinely collected by administrators of HPC machines is an excellent target for the application of machine learning approaches. There is a clear notion of "good" and "bad" and there is an obvious application: performance prediction and optimization. In this paper we will detail utilizing machine learning to model I/O on the Blue Waters supercomputer. We will outline data collection alongside usage of two representative machine learning approaches. Our final goal is the creation of a practical utility to advise application developers on I/O optimization strategies and further provide a heuristic allowing developers to weigh efforts against expectations. We have additionally devised an incremental experimental framework in an attempt to pinpoint impacts and causes thereof; in this way we hope to partially open the machine learning black box and communicate additional insights/considerations for future efforts.**

***Keywords*-component; formatting; style; styling;**

## I. Introduction

Recent developments in machine learning coincide with an explosion of successful applications across multiple domains. The ability of such techniques to combine non-mathematical, complex, or even non-understandable combinations of factors into incredibly accurate models continues to compel adoption in new areas. In an HPC environment however deep learning is nascent; while frameworks are deployed and supported, use cases are still fairly few and far between. This is expected as utilizing a machine learning model requires development and training of the model, and this training requires a dataset that represents a ground truth (that must be sufficient to allow the model to determine an answer). By nature, this is at odds with much of the at scale science utilizing HPC resources which tends toward exploration or the identification of causation, neither representing an obvious target for applying machine learning.

However, performance-related diagnostic data routinely collected by administrators of HPC machines is an excellent target for the application of machine learning approaches. First, there is a clear notion of good and bad regarding performance and modeling performance has an obvious application: performance prediction (and therefore optimization). For the Blue Waters supercomputer, the sheer amount of data collected offers challenges in applying machine learning, namely ensuring proper parameters are selected, both for the model and the input data, and that a training dataset is not sufficient to overfit the model. Our driving application is I/O optimization which is a routine bottleneck of HPC workflows. There is therefore obvious benefit to modeling I/O toward improved optimization. While the enormous configuration space makes this a difficult task, configurations are expressed through a relatively small number of easy-to-understand parameters.

In this work we explore machine learning methods for modeling application I/O on Blue Waters. We will show that even early-stage results prove useful for I/O optimization. The machine learning models we trained show room for refinement as their usefulness to accurately predict a single configuration's throughput is limited. The overall prediction distribution is however excellent. That is, one configuration's performance relative to another is accurately modeled. We will also outline an application-level utility that we developed to leverage these models' strengths in providing insights for I/O optimization.

In the remainder of this paper we will first motivate the adoption of machine learning modeling with a discussion of the I/O configuration space in Section II. We will follow with a brief background for both machine learning and I/O optimization in Section III. In Section IV we will describe I/O benchmark datasets and the codes we used to generate them. Then in Section V we will detail the derivation and testing of models on a small I/O benchmark dataset to gain initial insight into the efficacy of machine learning methods for I/O optimization. We then continue to incrementally improve/test to track progress and highlight impacts as well as demonstrate use of the optimization utility in Section VI. Finally we will close with a brief discussion in Section VII.

## II. Motivation

In this section we motivate our choice of methods for this work. In particular, why we are using machine learning to create a general I/O model for Blue Waters. Blue Waters has three distinct file systems: home file system for user home areas, project file system for team level storage/sharing, and a scratch file system for the large I/O of applications. All are Cray Sonexion Lustre storage technology (Lustre 2.5 gridraid) and each has separate metadata and object storage servers. Our target is modeling I/O on scratch file system, the largest and fastest of the three which provides 360 object storage targets (disks) and approximately 980 GB/s peak throughput [1].

While modeling I/O throughput or efficiency as a function of I/O configuration parameters seems straightforward but is belied by the lack of such models in practice. We believe the primary reason for this must be an overwhelming configuration space, one that would require a prohibitive amount of benchmarking to adequately model. We will informally discuss this space for Blue Waters below to illustrate the above point and explain our reasoning for turning to machine learning methods, namely to determine the viability of such methods to result in accurate, general models when trained on a significantly reduced collection of benchmark data.

There are several parameters to explain I/O configurations such as: job size, I/O size, aggregation level, number of files, file system settings, I/O library used, etc. The many interdependencies among these parameters make it difficult to clearly iterate through this space, i.e. certain values of one parameter may be incompatible with certain values of another and each such instance must be accounted for. We will, therefore, look at upper bounds to provide the notion of the I/O configuration space on Blue Waters some context. A natural first step is characterizing an I/O operation by its size. The largest "practical" write operation on Blue Waters would correspond to a job running on all 22,636 XE nodes and writing 50GB/node from each. 50GB is roughly 80% of the available memory on the node, leaving space for other storage requirements (node OS, states, etc.). This maximum operation is 1.08PB. To characterize all sizes, we need a minimum and an increment; selecting the minimum and increment to be a single byte the total number of operation sizes is the size of 1.08PB in bytes: 1,215,260,996,403,200. Even using the minimum file size of 512 bytes as both the minimum and increment reduces this number only to 2,373,556,633,600. While not a realistic characterization of I/O operation size in practice, this is indicative of the inherent difficulty in deploying theoretical approaches in Modeling such a space.

Keeping size constant, we now discuss other I/O configuration parameters. Staying on the XE partition, as stated above a job may run on [1, 22636] nodes and may use [1, 32] processors per node (PPN). We will make the simplifying assumption that whatever processors are participating in the operation, say a write, are writing the same amount. The operation could write a number of files from a single file to 724,352 (nodes*PPN) files. Lustre allows configuration of how a file is striped across disks both in the count of disks and size of stripes. On Blue Waters, a file may live on a single disk or striped across up to 360. We will only consider "reasonable" stripe sizes of 64KB up to 1GB, power of two increments for 15 total. The above parameters describe a space of 11,687,892,956,467,200 configurations. It may seem like this encapsulates significant overlap, for example, a 128KB write is counted both as 64KB written from two cores on one node as well as 64KB written from two cores on two nodes. This is a distinction we want to account for as the number of nodes a job is running on impacts factors such as potential network injection bandwidth and throughput to object storage servers.

The above number of configurations does, however, include many ill-advised Lustre striping settings. Assuming the "right" choice is always made (which certainly is not true in practice) the resulting number of configurations is 2,164,424,621,568. Even restraining operations to the only power of 2 node and core counts where each file is written to must be the same size (but again not true in practice) the number of possible configurations is 38,132,160. This number still does not include some parameters which have a practical impact, such as stripe offset, but assumes all I/O libraries are equivalent. Even this reduced number is prohibitive for any approach relying on extensive benchmarking.

## III. Background

### A. Machine Learning

Machine learning has often been used as a viable approach for capturing the relationship between complicated variables and outcomes. This relationship is mainly utilized through needs of prediction, regression, and classification. With the advancement of deep learning in recent years, complex inputs such as images [2], speech [3], and video [4] have been classified using various machine learning frameworks such as TensorFlow [5], and Caffe [6].

With various deep learning frameworks having increased the accessibility to machine learning algorithms, new applications of older approaches have also surfaced in recent years. For example, Yigitbasi et al. used machine learning approaches to tune MapReduce parameters [7]. Fan et al. used support vector machines to evaluate parallel computing frameworks such as Spark [8].

In our work, we compare a classic support vector regression approach with a deep neural network architecture in a new application of I/O optimization in HPC systems.

### B. *HPC I/O Optimization*

I/O stack on contemporary HPC systems has multiple components: application I/O profile, I/O middleware, file system client characteristics, network topology, file system configuration, and the file system architecture. Analyzing the cause of poor I/O performance is a complex undertaking, especially in large-scale parallel I/O systems comprised of multilayered software stacks and hardware components [9].

I/O profiling and optimization is complex due to the sheer number of factors/variables that affect the application I/O performance. In addition to application I/O pattern, other factors such as the system workload, other applications performing I/O simultaneously on the system, file system configuration, its default parameters, and file system specific features such as Lustre striping parameters all play a role in the the observed performance. Developers choice of I/O middleware also plays a role in the observed I/O performance.

Usually a process of experimentation and elimination is employed to determine the optimal values for various parameters for each application on a given system. This is a time consuming process and is typically a less productive use of system time and resources. As the breadth of applications and volume of data processed on systems increase, it becomes important to be able to provide a more efficient mechanism to achieve reasonably good I/O performance across all applications.

Tools like Darshan [10] provide a way to characterize the I/O profile of specific applications but users and system administrators still have to invest time in developing an understanding of the application behavior, and relate it to the system configuration and its state at the time of execution. This variability makes analysis complex.

These factors lead to a splintered approach to I/O optimization, targeting specific aspects or utilizing specific properties. Examples include leveraging hardware specifics [11], [12], developing middleware [13], and even hand tuning I/O configurations [14].

## IV. I/O Benchmark Datasets

The data described below was collected on Blue Waters during normal running conditions of the system. Beyond ensuring the model predictions are not artificially skewed by ideal conditions, our primary goal for this work is for the practical application of the models. Additionally, we believe machine learning, especially multilayered approaches, to be capable of incorporating system noise or runtime variability so as to better model I/O in practice. We utilize two benchmarks to collect performance data for both typical and exotic I/O configurations which are described below.

### A. *IOR*

The Interleaved or Random (IOR) I/O Benchmark [15] from Lawrence Livermore National Laboratory is an HPC center standard [16], [17], [18] for measuring read/write performance of parallel file systems. There are several parameters including file size, I/O transaction size, sequential vs. random access, and single shared file vs. file-per-process. The purpose of our IOR benchmarking is to collect data across the spectrum of what is typical of HPC I/O. We are therefore only measuring file-per-process and single shared file I/O patterns. In our experience, the vast majority of applications utilize one of these methods for writing simulation data. Furthermore, reducing the set of I/O patterns allows for the gathering of a wide range of other configuration parameters.

*1) File-per-process (600 tests):* utilizing 16 PPN we run on the following 14 total processor counts $2^i, 1 \leq i \leq 14$. File-per-process does not benefit from increased stripe counts so the files are written to single stripes. The file sizes are varied to correspond to multiple (12) stripe sizes between 512KB and 1GB: $2^i$bytes$, 19 \leq i \leq 30$. For one stripe size (32MB), we also tested PPN values of 1,2,4,8 across total processor sizes above up to 4096. Finally, there were three iterations of each test run.

*2) Single shared file (453 tests):* using similar parameter values as the file-process-test with adjustments to accommodate longer test times. We use the same set of stripe sizes, but only benchmark total processors up to 2048 (11 values). As opposed to file-per-process, when writing to a single shared it is crucial that stripe sizes be set appropriately. In these tests, the stripe size was always set to $min(\text{procs}, 360)$ with 360 being the maximum number of OSTs. At a certain point continuing to increase I/O parallelism beyond available OSTs will result in noticeable diminishing returns due to additional OST and network contention. In these cases added throughput comes at the cost of reduced efficiency. When the tests with processor counts higher than 360 were run, we also ran these with a stripe count of 256. We believe evenly distributing additional contention may improve efficiency and added this test to improve the likelihood of accurately modeling this and other impacts the limit of 360 OSTs may have. Similarly to file-per-process, for stripe size of 32MB we tested PPN values of 1,2,4,8 up to 1024 processors and again set number of iterations to three.

### B. *Custom Aggregation Benchmark*

There are many theoretical and maybe even some useful I/O configurations that IOR is not able to benchmark. This was the motivation behind a custom benchmark we developed to better understand HPC I/O [**?**]. That benchmark measures I/O models between file-per-process and single shared file across a constant I/O size. This corresponds to a space covering many combinations of $f$ files per node shared across $m$ nodes. In other words, the many ways an I/O operation may be aggregated within a running job are iterated. Figure 1 shows the configurations (from run 1 detailed below) in terms of overall number of processors

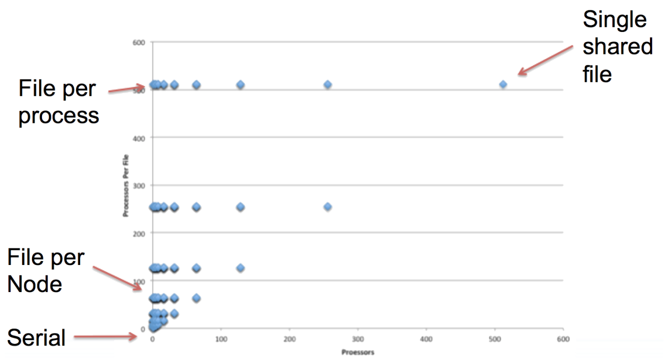


Figure 1: Direct display of 315 I/O models in the space of the number of processors performing I/O vs. the number of processors writing to each file.

vs. number of processors writing to each file and highlights common I/O patterns.

*1) Run 1 (315 tests):* writing 32MB from up to 32 nodes and for up to 16 PPN.

*2) Run 2 (315 tests):* writing 32GB from up to 32 nodes and for up to 16 PPN.

*3) Run 3 (826 tests):* writing 512MB from up to 512 nodes and for up to 16 PPN.

*4) Run 4 (675 tests):* writing 512GB from up to 512 nodes and for up to 16 PPN. The reason there are fewer possible tests for run 4 than run 3 is that serial I/O is no longer possible. An I/O operation of this size must be somewhat distributed to ensure proper available memory. This run had several very slow configurations some of which were disruptive to the file system. For this reason we collected results for chunks iterations from the beginning, middle, and end of the sets of configurations and completed 296 of the 675.

### C. ML Training/Testing Data

The above benchmarking code differ significantly with regard to configuration parameters. We however need to ensure the parameter space is consistent for using both datasets to train a machine learning model. While several parameters are not shared across codes with some even applicable to both, it is possible to derive the majority of parameters from one benchmark given the input values from the other. We wrote some simple utilities to go through the benchmark data and create such a unified set. Additionally though the selection of parameters to represent the configuration space is important in the modeling process. As in statistical or any other modeling intervariable dependencies, over/underrepresentation, etc. can be detrimental to resulting quality. The following is our decided set of parameters to describe the configuration space: nodes, PPN, number of nodes per file, number of files per node, and I/O size per processor. These are adequate to describe all of our tested aggregation models and are readily derived from both benchmark inputs. There are still some interdependencies among these variables but arise from inherent connections between specifying resources and resulting capabilities.

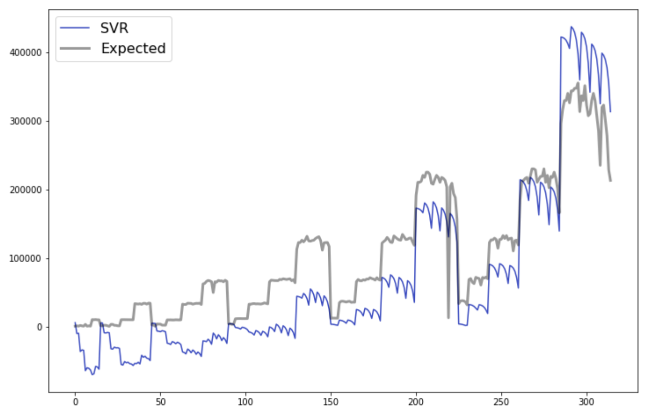
## V. Methods

### A. Machine Learning Methods

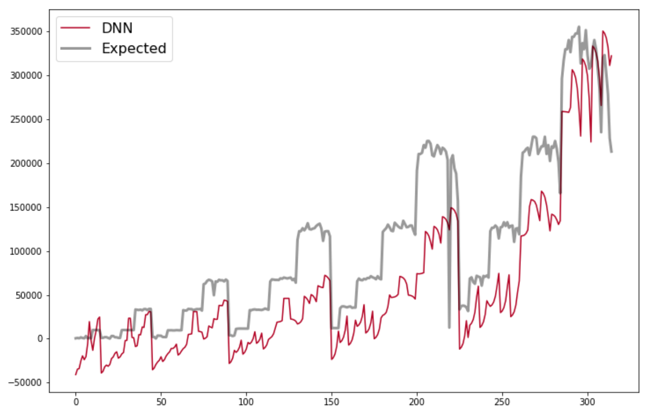
We selected two machine learning mechanisms to capture the relationship between various I/O parameters and throughput. Various degrees of efforts went into selecting the parameters with which to tune these algorithms. For some, there are several options with distinct theoretical capabilities and uses. Descriptions of possible parameters is beyond the scope of this work, we will however provide exact details for how we tuned the algorithms.

*1) Support Vector Machine:* refers to a class of supervised learning models that create a decision surface by fitting a set of hyperplanes to points in high dimensional space which is then used for classification or other purposes. One of these purposes is regression analysis and this subclass is referred to as Support Vector Regressors (SVRs). Our benchmark data as a function of configuration to throughput is a natural fit for this approach and we use the scikit-learn library [19]. Our selection of specific parameters was the result of a trial-and-error process, but we found the Radial Basis Function (RBF) kernel to consistent produce the best results for our data. Tuning the RBF kernel is accomplished through the setting of parameters `C`, the degree to which classification error is acceptable for smooth decision surfaces, and `gamma`, the impact of a single training example. Library documentation advises that high values for `C` favors correct classification and that `C` and `gamma` be spaced exponentially far apart. We used $\texttt{C} = 1000, \texttt{gamma} = 0.1 and \texttt{epsilon} = 0.001$. Data input to the model were normalized to be in the range $[0, 1]$.

*2) Deep Neural Network:* using Keras [20] which is built on TensorFlow. Deep Neural Networks (DNNs) model complex non-linear relationships and recent successes influenced our decision to test it. We used a "Sequential" model which is composed of a linear stack of layers. There are several configurable options, including the number of layers with each layer configurable as well. The other parameters correspond how to define error (metrics, objective function) and what approach to use to minimize it (optimizer). Our parameter selection was again the result of trial-and-error while working through documented example codes paired with test data resembling our I/O configurations. We found the Adaptive Moment Estimation (Adam) [21] optimizer to provide the best results, and used Mean Squared Error loss function with default metric, accuracy. We used five dense layers in total, with three hidden layers between the input and output layers. We specified Rectified Linear Unit (relu) activation for the first three layers and the identity activation function for the last. Input data for this model is normalized to be in the range of $[-1, 1]$ to accommodate relu activation.

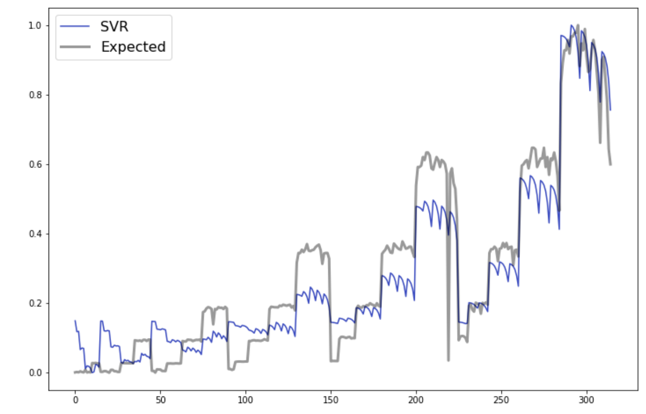


(a) SVR Predictions

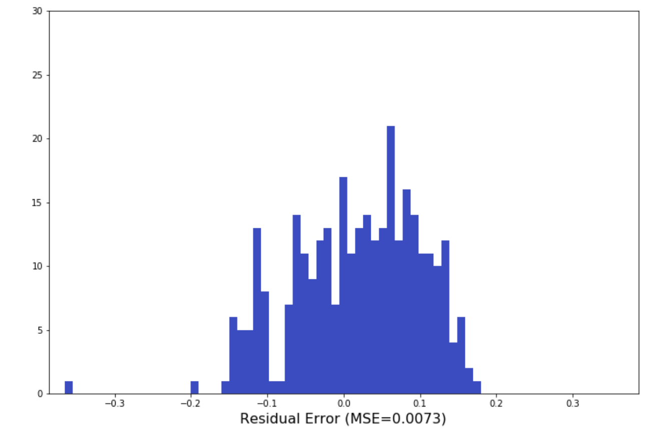


(b) DNN Predictions

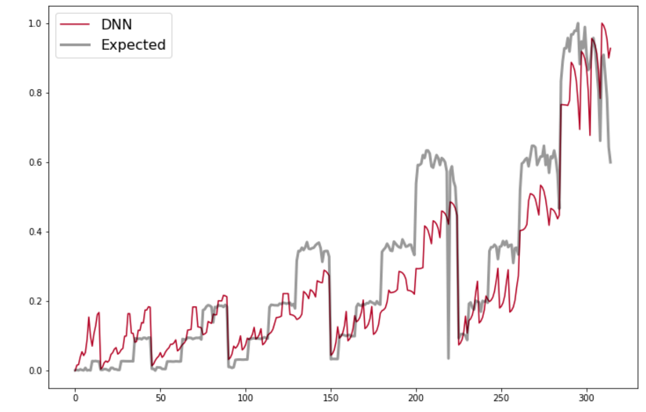
Figure 2: Expected throughput (gray) alongside (a) SVR predictions (blue) and (b) DNN predictions (red). Models were trained and tested on subsets of the aggregation benchmark data.



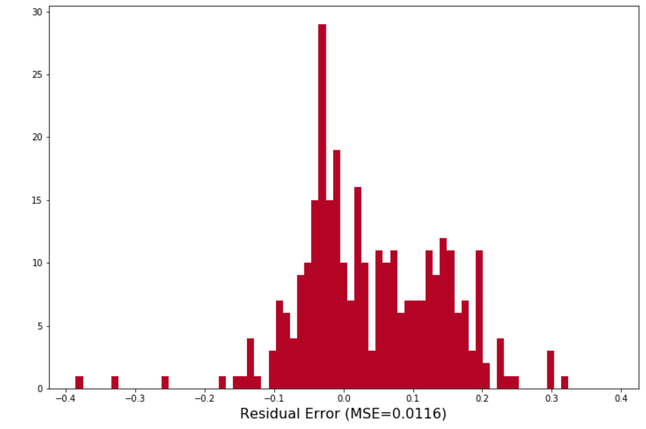
(a) Scaled SVR Predictions



(b) SVR Model Error



(c) Scaled DNN Error



(d) DNN Model Error

Figure 3: Expected throughput (gray) alongside (a) scaled SVR predictions (blue) and (c) scaled DNN predictions (red). The resulting residual errors are also shown for each (b), (c).

### B. *Utility Development*

The layered technical components of domain-specific concepts and jargon are made deployment of machine learning techniques far from trivial. The recent successes in the field are nevertheless a compelling testimony to the upside of overcoming these hurdles. The process of developing the full framework has a convenient and extremely useful property; difficulties in training accurate models are exclusive to that step with subsequent model use being entirely separable as a development effort. APIs include saving and loading mechanisms facilitating the independent development of codes using models. Furthermore, refined, improved, or even alternate models are simply drop-in replacements for older ones.

## VI. Results

In this section we show the incremental training and evaluation of I/O models. We follow with the details and results of a practical utility designed to provide insights to application developers endeavoring to improve I/O performance.

### A. *Benchmark Tests*

For our first model evaluation we trained each model on the aggregation benchmark's runs 2 and 3. To test we predicted the throughput measured in run 1. The purpose of this test is two fold. First, these are the fast running tests and therefore make sense as an initial test while continuing collection of larger/slower benchmark runs. Second, runs 2 and 3 will do not provide configurations for training that correspond to the smallest I/O sizes in run 1. Given our hope to model I/O from a reduced configuration set, such a test can prove insightful. Figure 2 shows the predictions of the SVR and DNN models. This figure is meant to give perspective as to the extent to which the scales of the predictions and expected values differ, especially apparent in the negative predictions. We believe numerical errors are likely to be tied to our tuning of the machine learning algorithms. Considering our relative inexperience in fine-tuning learning parameters for remaining results we will show expected values and predictions on independent scales to highlight how predictions generally variabilities across observed configurations.

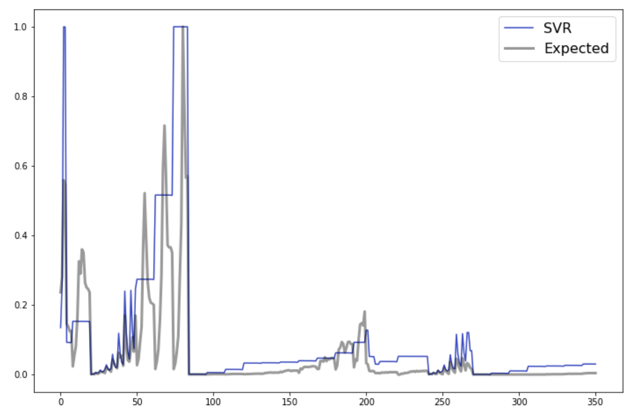
In Figure 3 we show the scaled predictions and the model errors. The small features in the expected values distribution corresponding to the aforementioned small-scale configurations show large relative errors for both models. The SVR model predicts local maxima where there are local minima whereas the DNN correctly predicts configurations corresponding to local peaks. The histograms of residual errors show a somewhat wide distribution highlighting overall model inaccuracy. Errors unevenly centered around zero can indicate a suboptimal set of parameters were used in training. Both exhibit somewhat balanced, also widespread errors overall as well as unbalanced features. While these plots show the models are not fundamentally wrong they also clear room for improvement.

We then trained each model on the full aggregation benchmark I/O data and tested the models on the IOR data. These results are shown in Figure 4. The IOR measurements show a distribution of expected values quite distinct from the previous test which illustrates the fundamental differences between benchmark codes. The residual error plots show perfectly centered and significantly reduced overall widespread errors. We believe the apparent capability of models trained on the series of uncommon configurations provided by the aggregation benchmark to accurately model standard benchmarking tests serves as both a proof of concept for this work as well as evidence for the viability of this application of machine learning. The DNN model again better matches the features of the distribution of expected values and as such remaining results will be based on the predictions of that model.

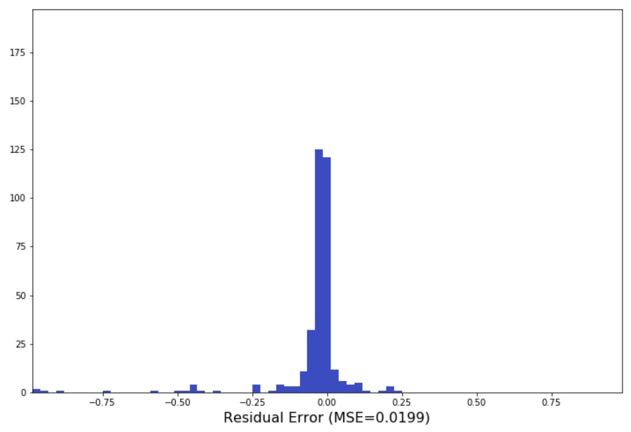
### B. *Optimization Utility*

Our model evaluations show the consistent disparity in scales between expected and predicted values. Across a set of configurations ,however, the distribution is quite similarly shaped to that of the expected values. That is, accurately predicting performance is not as reliable as predicting simply whether one is better than another (and even to what extent). We have therefore developed a way utilize this information in a utility for optimizing an application's I/O. The scientific value of an application is tied to a series of physics calculations that drive development, data structures, and scalability requirements. As a result, the data distribution may not be ideal for I/O creating or exacerbating a bottleneck. Worse yet, the considerable effort required to optimize I/O is readily apparent while the benefits of doing so are not.

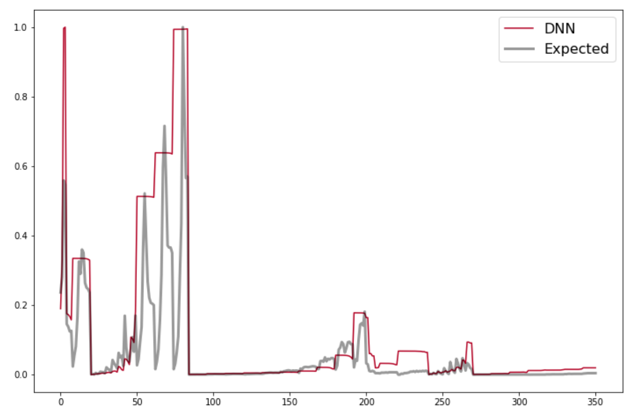
It is exactly this our utility addresses. Simulations run at targeted scales and an iteration's I/O specifics are also known. Our utility takes as input the number of nodes and PPN for the running simulation as well as the aggregate size of a step's I/O operations. From this information the utility we perform a brute-force iteration through all possible configurations, possible for a single application. For each configuration, performance is predicted and the full set of configuration predictions are plotted noting expected relative performances of typical approaches. Furthermore, an input of currently used I/O configuration may be specified (in terms of nodes per file, files per node, participating nodes and PPN) and this configuration will also be noted in the plots. For each configuration, we also show a plot of an efficiency metric of throughput per node (or write/node-hour) as this is how run time is measured and charged against an allocation on Blue Waters. Together these give a general context for measuring current performance as well as a potential for improvement. The utility also prints the predicted optimal configuration. Note, none of these outputs



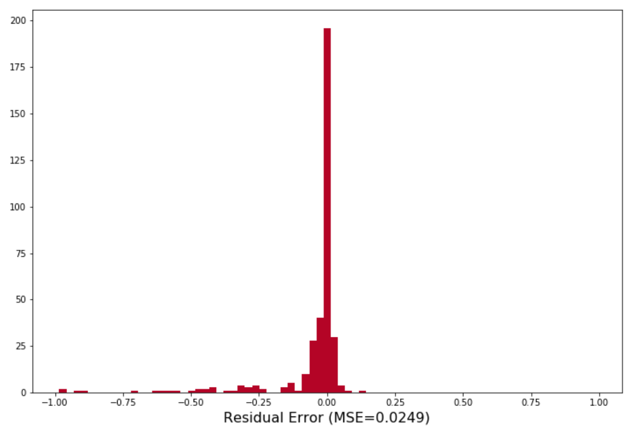
(a) 32MB I/O throughput.



(b) 32MB I/O throughput.



(c) 32MB I/O throughput.



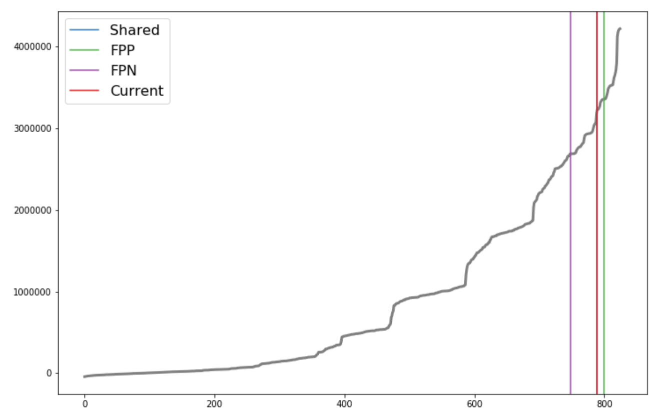
(d) 32MB I/O throughput.

Figure 4: Expected throughput (gray) alongside (a) scaled SVR predictions (blue) and (c) scaled DNN predictions (red). The resulting residual errors are also shown for each (b), (c). The models were trained on the full aggregation benchmark dataset and tested on the full IOR benchmark dataset.

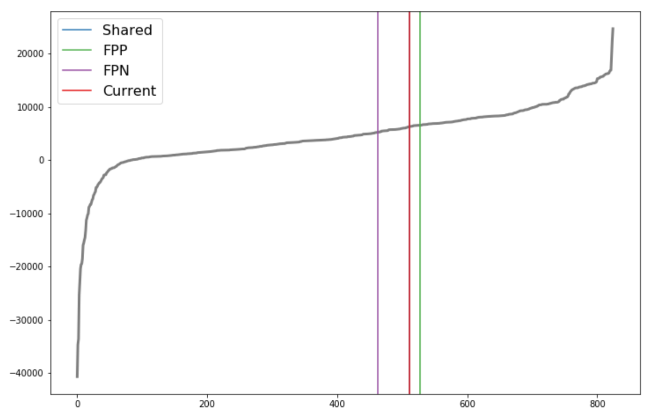
rely on communicating the actual prediction of any single configuration.

To show results of the utility we selected parameters corresponding an actual application running on Blue Waters for which we have profiled the I/O and have a general understanding of relative performance in practice. In this particular case single shared file, I/O was implemented to facilitate postprocessing and resulting I/O throughput was undesirable. The simulation was running 512 nodes, using a total of 8192 cores and writing 5GB per simulation iteration. While troubleshooting performance we found that the portion of the 5GB each processor was writing was further split across multiple simulation variables such that each write was only 32KB. The merit of this as a test case is that such small I/O is rare (and again wouldn't have been highly trained in our benchmarks) and without significant aggregation, we would expect poor performance. Figure 5 shows the throughput and efficiency predictions for this application's possible I/O configurations. The model unexpectedly suggests these configurations with little to no aggregation are among the best. The throughput-based predicted optimal involves slight aggregation, compressing 16 writes per node to 4, but no longer to a shared file but 2048 separate files. The range of negative values of efficiency show there are some configurations the model seems unsuited for.

Now we finally add the set of practical I/O data to our training set and train a final DNN model on our full set of benchmark data. The updated utility's predictions are displayed in Figure 6. As expected we are seeing higher throughput associated with aggregation (file-per-node vs. file-per-process) and that the file-per-process configuration is no longer among the those with the highest throughput. This time, the predicted optimal configuration is interesting: 256 nodes, 16PPN writing to 512 files. However, the configuration specifies each node writes to 8 files and that

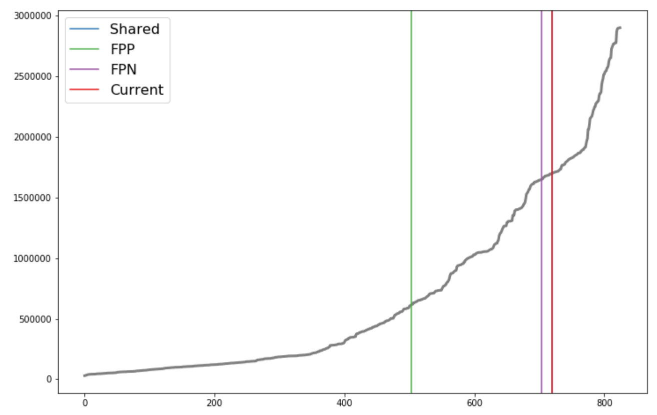


(a) Predicted throughput

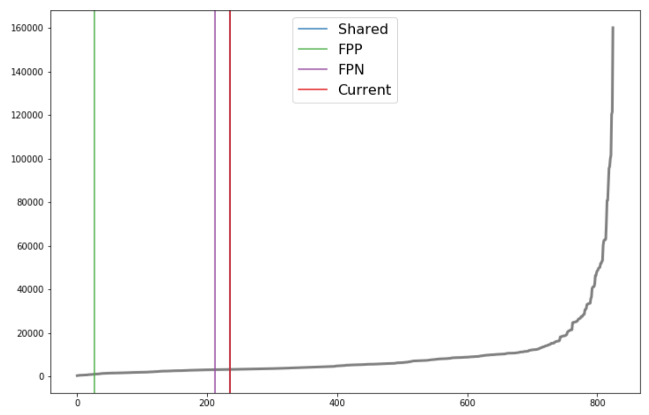


(b) Predicted efficiency

Figure 5: Throughput (a) and efficiency (b) charts of predictions of possible configurations of an application's I/O. The model used was the DNN evaluated in Figure 4. Shared file configuration is not shown in the charts as that is the applications current I/O model.



(a) Predicted throughput



(b) Predicted efficiency

Figure 6: Throughput (a) and efficiency (b) charts of predictions of possible configurations of an application's I/O. The model used was refined over previous tests by training on all available benchmark data. Shared file configuration is not shown in the charts as that is the applications current I/O model.

each file will be written to by 4 nodes. This "trick" of increasing the number of separate nodes writing to storage servers can improve utilization of the server's bandwidth as well as increase potential network injection. This likely represents a difficult implementation but not one totally dissimilar from other highly tuned applications. The efficiency chart no longer has a section of missed predictions and clearly communicates what we would expect, that the majority of configurations for such small I/O would not be efficient. Overall the model makes predictions, not at odds with common sense and shows a better representation of diminishing returns of highly parallel small-scale I/O.

## VII. Conclusion

In this work we have taken some early steps toward the creation of a practical and sustainable framework for utilizing machine learning models for optimizing parallel I/O. Even at this early stage, we have also provided evidence of the benefits of the approach supported through our alternate uses of model predictions. In our experience with the two models, SVR and DNN we found the relative difficulty in configuring the DNN learning to result in a model that better predicts relative performance across configurations, even when trained on fairly limited data. However, given the consistency of results and ease of tuning accompanying the use of SVR, we would certainly test both again given the chance, and would recommend doing so to others. There were some issues with the scales of our predicted values and in a future work, we would like to reapproach tuning the ML algorithms directed by scale adjustments.

## References

[1] "https://bluewaters.ncsa.illinois.edu/hardware-summary."

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[4] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[7] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema, "Towards machine learning-based auto-tuning of mapreduce," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*. IEEE, 2013, pp. 11–20.

[8] W. Fan, Z. Han, and R. Wang, "An evaluation model and benchmark for parallel computing frameworks," *Mobile Information Systems*, vol. 2018, 2018.

[9] C. Xu, S. Snyder, V. Venkatesan, P. Carns, O. Kulkarni, S. Byna, R. Sisneros, and K. Chadalavada, "Dxt: Darshan extended tracing," Argonne National Laboratory (ANL), Tech. Rep., 2017.

[10] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 2016, pp. 9–17.

[11] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 19.

[12] K. Chadalavada and R. Sisneros, "Analysis of the blue waters file system architecture for application i/o performance," in *Cray User Group Meeting (CUG 2013), Napa, CA*, 2013.

[13] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 155–163.

[14] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol, "Tuning parallel i/o on blue waters for writing 10 trillion particles," *Cray User Group (CUG)*, 2015.

[15] "IOR: interleaved or random hpc benchmark." [Online]. Available: https://github.com/chaos/ior

[16] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," in *Cray Users Group Meeting (CUG) 2007*, Seattle, Washington, May 2007.

[17] P. Wauteleta and P. Kestener, "Parallel io performance and scalability study on the prace curie supercomputer," Partnership For Advanced Computing in Europe (PRACE), Tech. Rep., September 2009.

[18] *Demonstrating lustre over a 100Gbps wide area network of 3,500km*, Salt Lake City, Utah, 11/2012 2012.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[20] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.