Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance

Megha Agarwal[1*†], Divyansh Singhvi[1*‡], Preeti Malakar[*§], Suren Byna[¶]
[*]IIT Kanpur, [¶]Lawrence Berkeley National Laboratory
[†]meghaagr@iitk.ac.in, [‡]dsinghvi@iitk.ac.in, [§]pmalakar@iitk.ac.in, [¶]sbyna@lbl.gov,

***Abstract*—Parallel I/O is an indispensable part of scientific applications. The current stack of parallel I/O contains many tunable parameters. While changing these parameters can increase I/O performance many-fold, the application developers usually resort to default values because tuning is a cumbersome process and requires expertise. We propose two auto-tuning models, based on active learning that recommend a good set of parameter values (currently tested with Lustre parameters and MPI-IO hints) for an application on a given system. These models use Bayesian optimization to find the values of parameters by minimizing an objective function. The first model runs the application to determine these values, whereas, the second model uses an I/O prediction model for the same. Thus the training time is significantly reduced in comparison to the first model (e.g., from 800 seconds to 18 seconds). Also both the models provide flexibility to focus on improvement of either read or write performance. To keep the tuning process generic, we have focused on both read and write performance. We have validated our models using an I/O benchmark (IOR) and 3 scientific application I/O kernels (S3D-IO, BT-IO and GenericIO) on two supercomputers (HPC2010 and Cori). Using the two models, we achieve an increase in I/O bandwidth of up to 11× over the default parameters. We got up to 3× improvements for 37 TB writes, corresponding to 1 billion particles in GenericIO. We also achieved up to 3.2× higher bandwidth for 4.8 TB of non-contiguous I/O in BT-IO benchmark.**

***Index Terms*—Parallel I/O, auto-tuning, active learning, performance prediction, machine learning**

I. Introduction

Scientific applications are routinely executed on high performance computing systems. These applications read and write gigabytes of data [1]. Some applications spend more than 50% of total execution times in I/O [2]. This not only causes a huge slowdown for the applications, but may also lead to wastage of useful compute resources when significant overlap of compute and I/O is not possible. One of the primary reasons for I/O being a bottleneck in the execution of scientific applications is the exponential growth in compute rates as compared to the I/O bandwidths. However, the MPI-IO layer of the MPI libraries and parallel file systems help in improving performance of parallel I/O to some extent. They provide a multitude of configuration parameters, which may help in increasing I/O bandwidth significantly, if set properly. Examples of such parameters are collective I/O buffer size, number of data aggregators for collective I/O, whether or not to perform data sieving, Lustre file system stripe size and count, etc.

The values of these various parameters affect the overall I/O time, and hence the I/O bandwidth. For example, larger the collective buffer size, lesser may be the number of iterations required to actually perform the POSIX reads/writes. It is also known that larger stripe size and stripe count in the Lustre file system is beneficial. The challenge however, is that the optimal value of the stripe size and stripe count depends on the data size, number of nodes and many other factors [3]. It is well-known that determining the optimal values of all these parameters to obtain the best I/O performance is difficult for a user due to the complex (and often unknown) interaction of I/O stack, network, application I/O patterns and other middleware related to file systems [4], [5]. An application developer spends maximum effort on code optimizations, rather than I/O parameter optimization. Therefore application developers resort to using default parameters.

Tuning various kinds of parameters also require an in-depth understanding of the I/O hierarchy, will be more complex in the future exascale systems. Automatically tuning parameters of interest has been used in various contexts, including parallel I/O [5], [6]. This relieves the application developers from performing parameter sweep experiments to tune the I/O performance. Auto-tuning hides the complexity of I/O interactions at various layers from the developer. Various approaches exist to automatically tune parameters, such as genetic algorithms, heuristic search etc. However, some of these approaches can take up to a few hours [4]. We propose a framework that automatically tunes the I/O parameters and find a good set of parameters. Our approach requires minimal human intervention and is independent of the application or the system. Our framework consists of two models for auto-tuning and an I/O prediction model.

Our first model, an execution-based auto-tuning model (ExAct), uses active learning to search better parameter values. The advantage of using active learning is that the model itself chooses the parameter values for next run and thus, eliminating the need of large data sets for training which is difficult to obtain due to large run-time of application codes. Active learning is also well-suited in our case due to complex interaction of system parameters and workload which is difficult to account in statistical models. We have used Bayesian optimization for parameter selection as it can handle a large number of parameters. Our model builds a surrogate function (probability model) based on past evaluations of the objective function, which in our case is the I/O bandwidth of the previous runs.

[1]Equal contribution.

978-1-7281-6005-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PDSW49588.2019.00007

Our approach achieves good performance in a few iterations ($\sim$20). It requires no manual effort beyond the initial set-up of selecting the range for each tunable parameter such as collective I/O buffer size, Lustre stripe size, etc. Using this model, we have automated tuning of Lustre and MPI-IO parameters related to data sieving and collective buffering. This model runs in a few seconds, however, the training time includes running the application with the model-selected parameters. We propose a second model for auto-tuning that also uses active learning, called prediction-based auto-tuning (PrAct). This model uses predicted values to determine the best set of parameters instead of executing the application with the model-selected parameters. For this, we have also built a performance prediction model using extreme gradient boosting machine learning algorithm. This reduces the model runtime by at least 40×.

We evaluated our models using three scientific application I/O kernels (S3D-IO, BT-IO and GenericIO) and an I/O benchmark (IOR). We showed performance improvements on up to 1024 cores on two supercomputers. We have experimented with different input data sizes, resulting in several terabytes of output and a maximum improvement of 11× over the bandwidth with default parameters. Overall, we have achieved a speedup of 3× across different benchmarks using the ExAct model. The I/O bandwidths using the parameters from the ExAct model resulted in 3× improvement for 37 TB writes in case of GenericIO and 4 TB I/O in case of BT-IO. We ran only a few time steps for our experiments, however, this gain is expected to multiply for production runs which are long running with multiple data write phases.

Our main contributions are as follows. We developed

1) a novel auto-tuning approach based on active learning that improves both read and write performance.
2) an execution-based auto-tuning of MPI-IO tunable parameters, that is able to achieve up to 11× speedup.
3) a fast prediction-based auto-tuning that can tune I/O parameters in 0.5 minutes.

The rest of this paper is organized as follows. We survey related work in §II. Our approaches are explained in §III and IV. The experimental setup and evaluations are presented in §V, followed by concluding remarks in §VI.

II. Related Work

Auto-tuning I/O Parameters: There exists several approaches to select optimal parameters from a large search space by using auto-tuning [6], such as genetic algorithms, Bayesian optimization, gradient descent etc. Many of these have been applied in various domains of high performance computing including parameter selection for parallel I/O [3]–[5], [7]. Behzad et al. [3], [5] used a heuristic-based search with a genetic algorithm to tune I/O performance. The heuristic-based search has a long run time and could not be applied on a different configuration than the trained configuration. Howison et al. studied manually tuning HDF5 applications on Lustre file systems [8]. It is a cumbersome process to manually tune; the application developer typically uses the default parameters. McLay et al. [9] study tuning parallel I/O on a specific system and suggest that maximizing stripe count has a significant impact on performance. In contrast to the above approaches, we propose a method to automatically find a good set of parameters for I/O in relatively less time using active learning.

Performance Prediction of I/O: There has been a lot of work on analytically modeling I/O performance. Lee and Katz [10] developed analytical models of disk arrays to approximate their utilization, response time, and throughput. Song et al. [11] proposed an analytical model to predict the cost of read operations for accessing data. Data organization in different layouts on the file system is used to predict cost. Barker et al. [12] used analytical performance models for two applications to test their performance for new storage system deployment. With increasing complexity of system software, hardware and complex layout of files ystems, developing analytical models is often time-consuming and inaccurate to obtain the expected prediction accuracy. Therefore, several researchers propose machine learning (ML) models for modeling I/O performance.

Behzad et al. [5] propose non-linear regression models for performance prediction, the model being specific to an application. Herbein et al. [13] use a statistical model, called surrogate-based modeling to predict the performance of the I/O operations. Behzad et al. [4] developed a semi-empirical approach to model the performance of MPI-IO operations. Isaila et al. [14] combined analytical and ML approaches for modeling the performance of ROMIO collectives. Xie et al. [15] developed micro-benchmarks to characterize storage system write performance, identified important input parameters, and developed ML-based models. In contrast, we use fast ML algorithms to predict with reasonably good average prediction accuracy. We use these predicted values to reduce the runtime of our active-learning model to tune the I/O parameters.

III. Auto-tuning parallel I/O

The best parameterization for parallel I/O is obtained by fine-tuning several MPI-IO parameters and file system parameters. It is difficult for users and application developers to fine-tune the I/O parameters based on their application, data size, file access patterns and system. This is due to the huge search space. E.g., in case of Lustre parameters, typical stripe size can be set between 1 MB – 8 MB and stripe count can be between 1 – maximum number of object storage targets (OST) (which can be more than 200 [16]). Thus, the search space with these two parameters is about 1600 if size is varied by unit of 1MB. Additionally, there are several MPI-IO parameters, thereby increasing the search space. Therefore, a feasible and acceptable solution is to automatically tune the system using machine learning (ML). Further, we have used active learning to reduce the sample space associated with ML models. Next, we describe the various auto-tuning and prediction models we have used. In this section, we describe the ExAct model that uses active learning based on the actual execution times for auto tuning of I/O. We also describe some of the implementation details for auto tuning. Section IV describes two models – an I/O bandwidth prediction model (Predict), and an enhanced auto-tuning model (PrAct) that uses Predict with active learning.

Execution-based Auto-tuning (ExAct): Active learning is a special case of semi-supervised machine learning in which a learning algorithm is able to interactively query the user (or some other information source) to obtain the desired outputs at new data points. The main hypothesis in active learning is that if a learning algorithm can choose the data it wants to learn from, it can perform better than traditional methods with substantially less data for training. Therefore, we are able to obtain good results by just providing an input configuration. The input configuration refers to data size (e.g., grid size or number of particles). Our aim here is to use Bayesian Optimization to find values of MPI-IO and Lustre [17] parameters that gives the best performance in terms of I/O bandwidth for a given data size of the particular application and the underlying file system. We can state our problem as

$$x* = \arg\max_{x \in \mathcal{X}} f(x)$$

where $f(x)$ represents the objective function to minimize, i.e, inverse of I/O bandwidth. x can take any value in space of parameters denoted by $\mathcal{X}$ and $x*$ denotes the set of values for tunable parameters which minimizes the function.

The two key problems that auto-tuning models face are [4]:

- finding best set of parameters in optimal time
- feeding those parameters without modifications in code

We have resolved the second problem by inserting the parameter values obtained from the model through environment variables at run-time. This avoids application-code modifications and hence recompilation of whole application. A naïve strategy to obtain the best set of parameters is to run the application on a specific system across the entire parameter space with all possible values and obtain the best performing set [18] [19]. However, this is not only tedious, but will incur large computation overhead especially when the search space is continuous. We can reduce the search space analytically or improve efficiency by doing random search. However, random search is unreliable and may not result in the best set or close to best set every time. Also, in some cases, parameters selected from random search results in performance degradation. Due to complex interaction between parameters and run-time, the above approaches are less desirable. Therefore, we use Bayesian Optimization(BO) in our work, as described next. The advantage of BO over random search is that when the number of parameters to be tuned increases, the latter performs miserably.

1) Bayesian Optimization: Bayesian Optimization (BO) [20], [21] selects the set of parameters as an informed decision based on performance in previous runs whereas random search selects values randomly, independent of results in the previous trial runs. BO avoids expensive evaluations of the objective function by selecting the next set of parameters based on those that gave good results in the past [22] [23]. Bayesian Optimization keeps track of previous evaluations by computing P which is defined below:

$$P(score|parameters)$$

In our case 'score' is I/O bandwidth. In literature, this model is named as 'surrogate' for objective function and is denoted by $P(y/x)$. The Bayesian method optimizes this surrogate rather than the objective function. Figure 1 denotes the steps taken by Bayesian Optimization in our work. At each step, a decision is

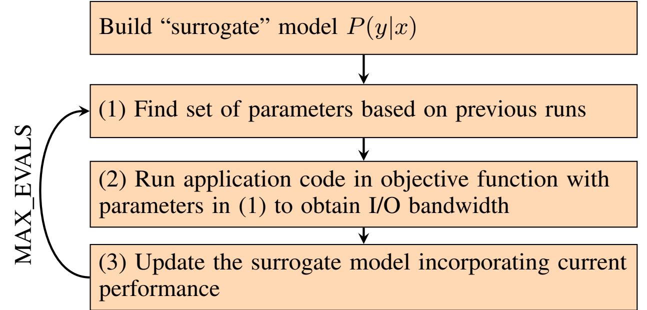


Fig. 1: Bayesian Optimization steps

taken by the model in an informed manner. The time spent in taking this decision for selecting values of parameters for next run is insignificant as compared to the time taken to run the application at each step (Step 2 in the figure). The model runs for MAX_EVALS steps (we ran for 20 iterations). Next, we discuss some algorithmic details of our model.

2) BO Model Details: Sequential model Bayesian Optimization algorithms are formalization of Bayesian Optimization. [24] As the name suggests, sequential denotes running trials one after the other, and each time finding parameters by applying Bayesian reasoning and updating surrogate (as shown in Fig 1). There are five aspects to optimization as discussed in [22]; these are stated below.

- Domain: The domain of values of parameters to be tuned are the Lustre parameters and MPI-IO hints (see V-A). Search space of each parameter is stated as range in case of continuous space and set of values in case of discrete space. For e.g., search space of stripe_count is stated as [1, Total #OSTs]. One can associate the probability function with search space associating higher probability to region where optimal parameters are most likely to be available. We have assigned uniform function as it is equally likely to find optimal parameters in the entire range.
- Objective function: It takes the parameter values as input and outputs the loss or $f(x)$ that we want to minimize. In our case, the loss is reciprocal of I/O bandwidth of the application. The objective function runs the application code to obtain loss and hence is the most expensive part. The goal of the entire model is to converge in fewer iterations and to minimize the number of calls to this function (MAX_EVALS).
- Surrogate model of objective function: It is a probabilistic model built using previous evaluations. There are various functions for modeling surrogate like Gaussian Processes and Random Forest. We have used Tree-structured Parzen Estimator (TPE) for our purpose. TPE constructs the model by using Bayes rule. Instead of directly using

$p(y|x)$, it uses

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

$p(x|y)$ is the probability of the set of parameter values given score.

- Selection function: It denotes the criteria by which next set of parameters values are chosen from the surrogate function. We used Expected Improvement (EI) criteria.

$$EI_{y*}(x) = \int_{-\inf}^{y*} (y^* - y)p(y|x)dx$$

$y*$ is a threshold value of the objective function, y is the actual value of the objective function using parameter set x. The model objective is to maximize EI with respect to x.

- History: Each time the algorithm finds a new set of candidate values of parameters, it evaluates them with the actual objective function and records the result in a pair (score, values of parameters). These records form the history and are used in subsequent runs.

The run-time of ExAct is directly proportional to product of the number of times the application is run (20 in our case) and the time taken to run the application. Our approach of using Bayesian optimization reduces the time to find a feasible solution. The advantage of using ExAct over existing auto tuning framework is faster convergence. However, it can still take up to an hour to run a few iterations of an I/O benchmark on a few nodes, especially for large data. Therefore, to reduce the total time taken by ExAct, we first propose a performance prediction model to predict the I/O bandwidth, and then use that to further reduce the auto-tuning model runtime. We describe both in the next section.

IV. I/O Performance Prediction and Tuning

In this section, we first describe our performance prediction model for I/O. Next, we describe our enhanced auto-tuning model which uses these predictions to output best parameters within a few seconds.

A. *Performance Prediction Model (Predict)*

Performance modeling is an important problem in HPC. A performance model should be able to predict the performance (e.g. runtime) of an application on a given system. Performance models are useful for system administrators and developers alike, in order to correctly estimate the total job runtime. This is useful to reduce queue wait times, and improve system throughput. Further, accurate I/O predictions can help application developers tune the I/O frequency appropriately.

Typically, developers routinely run their application codes on a system for a long time. They also experiment with different parameters for various input configurations. Science users typically archive their runs. We used the data collected from the ExAct runs, however logs of prior runs can be also used. We used machine learning algorithms to predict the I/O performance. Particularly, we have used extreme gradient boosting (XGB) [25]. They have been shown to give good predictions for non-linear relationships between the input parameters [7], as is our case of parallel I/O. The input data consists of various input configurations (problem sizes, number of nodes etc.) and input parameters (MPI-IO parameters and file system parameters). The output is I/O bandwidth. Details of the hyperparameters of XGB are presented in Section V-A. The train-test accuracy of the model is discussed in Section V-D2.

B. *Prediction-based Auto-tuning (PrAct)*

In this section, we describe our modified auto-tuning model. As mentioned earlier, computing objective function is a bottleneck of ExAct model. We propose a model to overcome this shortcoming. This model utilizes the Predict model (see §IV-A) as the objective function. Instead of running the application, we incorporate the Predict model values in Step (2) of Figure 1. This significantly reduces the execution time of our active learning model-based auto-tuning to determine the best set of I/O parameters. The reduction in model time is at least 40×, as compared to ExAct, thereby significantly improving the training time.

V. Experiments and Results

In this section, we first describe our implementation details, followed by details of the four I/O benchmarks and application I/O kernels that we have used in this work. We next describe the two supercomputers on which we ran our experiments, followed by demonstration of our model results.

A. *Implementation*

The various parallel I/O parameters that we tune are MPI-IO parameters such as romio_ds_read, romio_ds_write, romio_cb_read, romio_cb_write, cb_buffer_size, cb_nodes and Lustre file system parameters such as the stripe_count and the stripe_size. It is well-studied [14], [26] that the above hints can greatly affect the parallel I/O performance, and hence have been chosen in our work. All the models described in §III are implemented in Python using Hyperopt [27] (a Python library for hyperparameter optimization). For pre-processing in prediction based model, we applied MinMaxScaler transformation to scale each input and output between 0 and 1 before training and applied the inverse transformation after prediction so that evaluation metrics are computed in the original scale. Tree-structured Parzen estimator (TPE) is used as the optimization algorithm in Hyperopt. The TPE approach models $P(x|y)$ and $P(y)$ where x represents parameters and y the associated score (I/O bandwidth). Thus using TPE as search algorithm and bias defined on sample space, Hyperopt finds the best set of parameters. In the beginning of first iteration, each value is initialized by TPE on the basis of bias defined on search space. In our case, for each hint we have given uniform distribution between min and max values as bias. In ExAct, for each configuration, 20 iterations are used to obtain the best parameter values. Increase in number of iterations will yield better set of parameters, closer to the optimum value.

We use environment variables (specific to system and MPI implementation) to set the I/O hints.

In *Predict* (§IV-A), XGB is used with max_depth=10, n_estimators=1000 and linear regressor. 30/70 train-test split is done and results are averaged for 10 such random splits to increase confidence in model. For each benchmark, two models are trained – one for predicting read bandwidth and the other for write bandwidth. We had 1136 data points for S3D-IO, 414 data points for BT-IO, 658 for IOR and 1004 data points for GenericIO. These data points were generated while running ExAct, and serve as the training and test data for XGB. The input data includes runs of various configurations over various MPI and Lustre parameters while obtaining ExAct parameters. Each data point contains the I/O bandwidth, the tuning parameters and the configuration on which it was run. In PrAct, the Predict model is used as objective function and results are obtained on unseen or new configurations. The number of iterations in PrAct for model to converge is 100 for each input. Time taken for training Predict is ~ 11 seconds (data-dependant). Training time for ExAct is $20 \times$ runtime because we run ExAct 20 times, and ExAct runs the application every iteration. However, we only run for a small number of time steps for each benchmark (e.g. 3 time steps for S3D-IO). Training time for PrAct is ~ 18 seconds, independent of configuration. Thus, while on an average it takes ~ 1000 seconds to obtain best parameters from ExAct for GenericIO, it takes only ~ 18 seconds for PrAct ($44\times$) reduction.

B. Benchmarks

We describe below the I/O benchmarks and application I/O kernels that we have used in our work.

Interleaved or Random I/O benchmark (IOR) [28], [29] is an I/O benchmark developed at Lawrence Livermore National Laboratory (LLNL). It is a highly configurable benchmark and is one of the most widely used I/O benchmarks. The input to IOR is transfer size and block size. We used MPIIO API. We used "-e" (fsync) option to flush the cached pages to the Lustre file system upon writes. The Lustre directives were set with "-O" option and MPI-IO hints were passed using "-U" option. We also experimented with and without file-per-process option of IOR. We experimented with various transfer (data per transfer) and block sizes (data per task).

S3D-IO [30] is an I/O kernel for the S3D combustion application [31], developed at the Sandia National Laboratories. It uses the PnetCDF library to perform parallel I/O. The input configuration consists of 3D grid sizes and number of processes in each dimension. We used non-blocking collective I/O.

GenericIO [32] is the I/O kernel for the HACC cosmology simulation code, developed at the Argonne National Laboratory. It consists of 2 inputs – number of particles and seed. Each particle is defined by nine variables, such as the coordinates and physical properties. It is a data-intensive application, our experiments output data of the order of several TBs.

BT-IO [33] benchmarks the performance of PnetCDF and MPI-IO for the I/O pattern used by the BT code of NAS Parallel Benchmarks. It uses a block-tridiagonal partitioning pattern on a 3D array across a square number of MPI processes. The input consists of 3D grid sizes. We used PnetCDF non-blocking I/O.

C. System setup

We used two systems – HPC2010 and Cori. HPC2010 [34] is a 368-node supercomputer at the Indian Institute of Technology Kanpur. Each node is an Intel Xeon (Nehalem) 8-core processor with 48 GB RAM. It has a peak performance of 34.5 TFlops. The nodes are connected by Infiniband switches in a fat-tree topology. It has a Lustre file system (100 TB storage) with 24 OSTs. We used a maximum of 128 processes. Cori [16] is a CrayXC40 supercomputer at the Lawrence Berkeley National Laboratory with 2388 Intel Xeon Haswell processor nodes and 9688 Intel Knights Landing processor nodes. We ran all our experiments on upto 16 nodes of the Haswell partition. Each Haswell node has 32 cores. Cori has a Lustre file system (30 PB storage) with 248 OSTs. We used a maximum of 512 processes. We ran our experiments using the Intel MPI on HPC2010 and Cray MPI on Cori. On both systems, the default stripe count is 1 and the default stripe size is 1 MB.

D. Results

In this section, we show the results from ExAct, Predict and PrAct from all our benchmarks and on the two systems. The experiments have been repeated 5 times, and average values have been plotted.

1) Execution-based Auto-tuning (ExAct): We obtained the read and write bandwidths by running ExAct (20 iterations) on all four benchmarks for various configurations. The results are obtained by running the benchmarks with the best parameters obtained from the model. We compare this with the I/O bandwidths obtained from running the benchmarks with default parameters on the system.

Figure 2 shows the results for S3D-IO on HPC2010. The y-axis shows the default and ExAct model read-write bandwidths (y-axis ranges are different for each plot). The figure shows strong scaling results for 10 different data sizes on 16 – 128 processes. We achieved a maximum of $6\times$ and $4\times$ improvement in read and write bandwidths over the default across all the 40 configurations shown in the figure. Note that we have experimented with different skewed data sizes and achieved a good performance improvement. For e.g., we obtained a read bandwidth of 710 MBps as compared to the default 115 MBps for the configuration $100 \times 200 \times 400$ (16 processes). Also, note that for higher process count, ExAct is able to achieve higher performance improvements for the same data size. This is because the effect of using more OSTs (as auto-tuned by ExAct) is more pronounced at higher node counts, since more nodes/tasks are able to drive the concurrency achieved with more OSTs. For the input configuration of $200 \times 400 \times 400$ on 128 processes, the default parameters resulted in I/O time of 19.13 seconds, whereas with ExAct parameters, the I/O time was 12.65 seconds. This was due to disabling aggregation, and using stripe_count=7 by ExAct as compared to default of 1.

Figure 3 shows strong scaling and data scaling for IOR on HPC2010. This was without using the file per process option

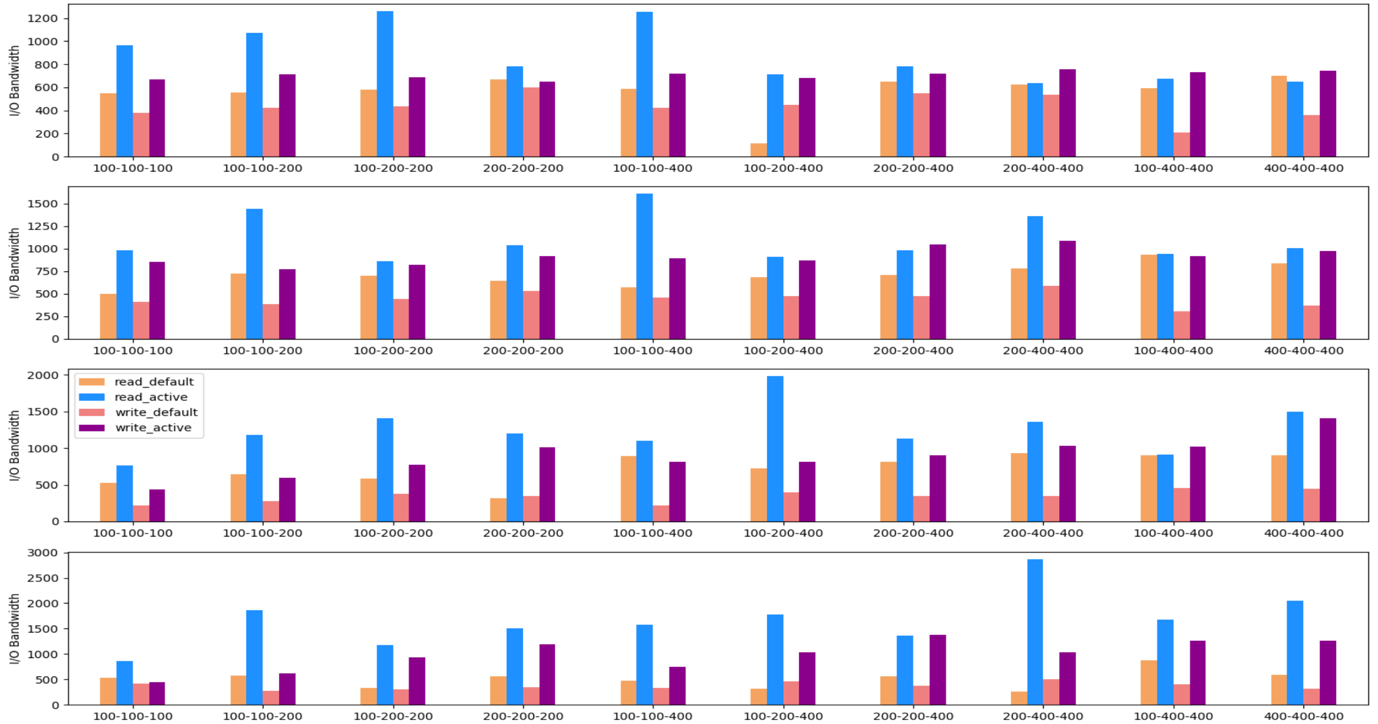
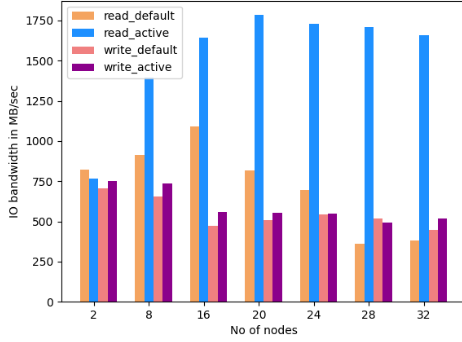


Fig. 2: Default vs. best bandwidth from ExAct for various configurations of S3D-IO on 16, 32, 64 and 128 processes of HPC2010. X-ticks of each plot (of the form x-y-z) represents the 3D grid size of the input data.

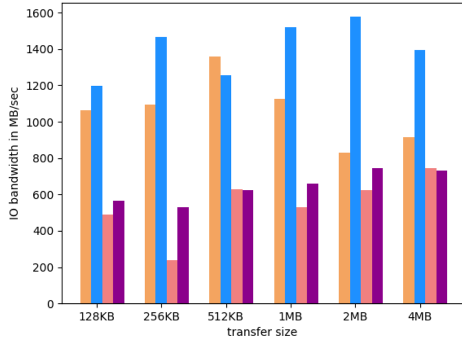
and with the fsync option. We achieved 87% and 20% average improvement in read and write bandwidths across all runs. The reason for this huge improvement is that in each iteration, our model tries to minimize loss, which in our case is the execution time of the application. It does so by selecting better parameters based on the history (see §III). For instance, the model read bandwidth for 28 nodes (Figure 3(a)) is 1708.16 MBps as compared to 360.59 MBps in the default case. In this case, the block size (i.e. amount of data written per task) is 200 MB, resulting in about 43 GB data. The improvement is because our model selects better parameters based on active learning. For example, it selected a stripe size of 20 MB, as compared to the default of 1 MB, and a stripe count of 16 as compared to default of 1 in this case. We show speedups in Figure 3(b) for varying transfer sizes (100 MB block size). We also observe higher improvement in read bandwidth than write bandwidth as compared to default I/O bandwidths. The application reads and writes during each run, and the data sizes for reads/writes are the same. Increase in read bandwidth minimizes loss, giving positive feedback to the model to change parameters. However, since our loss function is the overall runtime of the application, the model tries to optimize for that, instead of minimizing read and write times separately.

Figure 4 shows strong scaling results for GenericIO. The x-axis shows various data sizes per task (MB), and the y-axis shows I/O bandwidths (MBps). The four plots show results on 16, 32, 64, 112 processes on HPC2010. The number of particles (shown in x-axis) per task varied from 26 million – 1 billion. The maximum improvement in read and write bandwidths are 5.24$\times$ and 5.99$\times$ respectively. In contrast to IOR and S3D-IO, improvement of write bandwidth over default is greater than read bandwidth. Also note that ExAct is able to auto-tune the MPI-IO parameters and Lustre parameters such that write performance of large data sizes (37.9 TB corresponding to 1 billion particles) improved by 3$\times$. Further, for higher number of cores and data size, performance of GenericIO with ExAct I/O parameters is better. For example, ExAct parameters give 1.6$\times$ and 2.4$\times$ better results on average than the default for 28 nodes (112 processes). This shows that the default parameters are not at all suitable for large-scale I/O. A model like ours can help the application developers in automatically tuning the I/O parameters through active learning in a very few iterations with few sample points, as compared to traditional machine learning or other approaches.

Figure 5 shows strong scaling results for BT-IO on 16 and 64 cores respectively (BT-IO requires a perfect square number of processes). For large number of cores and large data size, ratio of ExAct to default bandwidth increases, suggesting more improvements for large number of cores. We achieved an average improvement of 1.3$\times$ and 2.1$\times$ for read and write bandwidths. Note that, in this case we achieved a higher performance improvement for writes. This is because BT-IO uses a block tri-diagonal write pattern [35]. In this case, the writes are not from contiguous blocks, and therefore tuning the I/O parameters is very important for BT-IO. The input configuration $500 \times 500 \times 500$ corresponds to 4.8 TB of I/O data. For such high data sizes, we have achieved 3.2$\times$ speedup because of a robust model. Figure 6 shows read and write bandwidths of S3D-IO in Cori, averaged over 5 runs. We used a maximum of 56 OSTs on 16 nodes of Cori. Therefore, we



(a) IOR I/O bandwidths for varying node counts.



(b) IOR I/O bandwidths for varying transfer sizes.

Fig. 3: Default vs. ExAct I/O bandwidths using IOR on HPC2010. (a) Strong scaling on 16 – 128 processes (b) Data scaling on 64 cores with 100 MB block size.

set the range of stripe_count to 56 on Cori for ExAct. Similar to HPC2010, the read bandwidth improvements are higher. On Cori, which uses Cray MPI, the number of cb_nodes is also set to the stripe count. The default stripe size is 1 MB and stripe count is 1. The default collective buffer size is 16 MB. Whereas, our model runs S3D-IO on a given number of nodes, selects the I/O parameters based on the previous runs that gave best results, and runs again until it finds the best configuration. For example, the average default read and write bandwidths are 13.8 and 3.7 GBps, whereas, with the ExAct parameters (average stripe_size=21 MB, average stripe_count=52 from 5 runs), the average read and write bandwidths were 18.5 and 4.3 GBps. We also ran IOR on Cori with ExAct parameters and the default for varying transfer sizes (128 KB – 8 MB) for a block size of 200 MB. This was using file per process and collective I/O. On an average, we observed about 18%, 11%, 22%, 14%, 45%, 40%, 45% performance improvements in reads for transfer sizes of 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB and 8 MB respectively. We note higher improvements for larger transfer sizes, because tuning the default parameters matters more for large data transfers.

Figure 7(a) shows how the density of the sample space varies before and after running the model. We can see that our hypothesis in all cases of stripe_size was uniform distribution between a range. The Bayesian optimization model tuned the density to predict better stripe_size. For each input, the optimum values lie in different ranges so uniform distribution as bias on sample space is the best choice. We also note that in trial run, stripe_count in Figure 7(b) which is given a prior of log-uniform probability distribution flattens somewhere and peaks elsewhere. It shows that even if our initial hypothesis of the sample space is wrong, the model will eventually find the optimal parameters. ExAct model improvements for each benchmark on HPC2010 are summarized in Table I. Columns 2 and 4 show the average and maximum read bandwidth improvements, columns 3 and 5 show the average and maximum write bandwidth improvements with ExAct.

Benchmark	Read (Avg)	Write (Avg)	Read (Max)	Write (Max)
S3D-IO	1.97X	2.21X	11.14X	4.03X
IOR	2.1X	1.0X	4.73X	2.23X
BT-IO	1.07X	1.76X	2.93X	4.86X
GenericIO	1.44X	1.51X	3.04X	3.06X

TABLE I: Overall % improvement of ExAct Model on HPC2010

2) Performance Prediction (Predict): In this section, we show results for our performance prediction model, Predict. We used the data collected while running ExAct for all the benchmarks as the training and test data. Data can also be collected from the production run logs of the applications. We trained two models – one for predicting read bandwidth and other for predicting write bandwidth, given the value of input configuration and hints. This is because ExAct model outputs different read and write parameters (e.g. romio_ds_read and romio_ds_write). We next experimented with extreme gradient boosting (XGB) for regression. We ran XGBoost [36] regression model for all the benchmarks for 30/70 train/test split to predict the read and write bandwidths. The model took around 20 seconds to train for each application. Figure 8 shows performance of XGB predictions (Predict) for all four benchmarks on HPC2010. The results are average of 10-fold cross validations (10 different sets of train/test splits). For BT-IO and S3D-IO, the plots are well-centered around the black dotted line (ideal case with 100% accuracy). For S3D-IO write bandwidth, we observed high R^2 scores for even 10/90 train/test split. The R^2 scores for 50/50, 30/70, 20/80, 5/95 train/test splits are 0.87, 0.85, 0.85 and 0.62 respectively. Thus, even for small sample data set, the XGB model gives good result. This is because XGB has a novel tree learning algorithm to handle training point weights efficiently. Note that typical ML models train/test split is 80/20. Our model is able to achieve good predictions because of good sample data (from ExAct) and appropriate hyperparameter tuning of XGB [7].

Table II shows the median absolute percentage error (MdAPE) and R^2 score for XGB predictions, across all configurations for four benchmarks on HPC2010 and one benchmark on Cori (last row) with 10-fold cross validation to obtain robust results. Coefficient of determination (R^2) is used to assess the goodness of model fit. It has a maximum value of 1 which indicates model perfectly fits the data. Higher the R^2, better the model fits the data. We observe less than 20% error for most cases, except read predictions for IOR. The R^2 score for IOR on HPC2010 is really poor. A possible reason for low

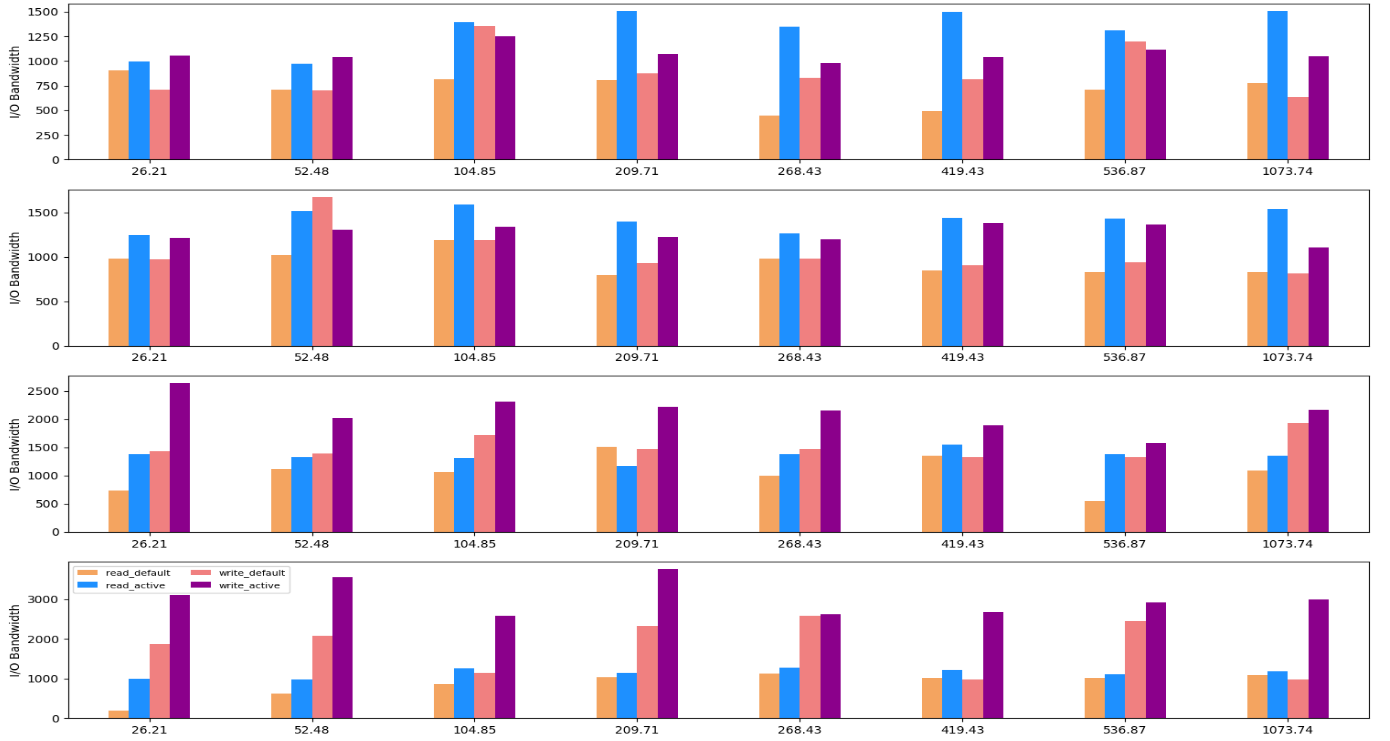


Fig. 4: Default vs. best bandwidth using ExAct on GenericIO for various particle sizes on 2, 4, 16, 28 nodes of HPC2010 (8 processes per node). Y-axis represents I/O bandwidth in MBps and x-axis represents number of particles (in millions).

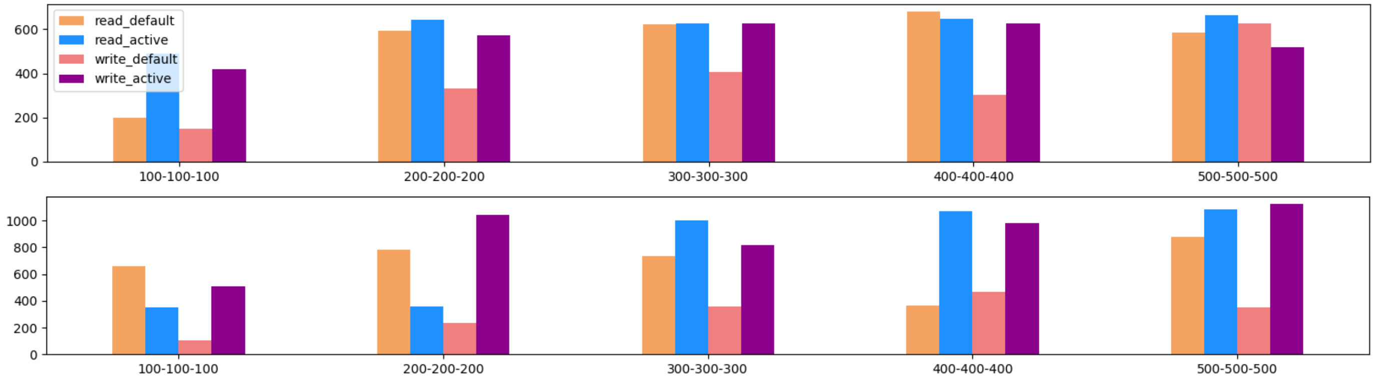


Fig. 5: Default vs. best bandwidth from ExAct for BT-IO on HPC2010 for various configurations. Each group of 4 bar charts represent read and write default and ExAct bandwidths. X-ticks for each plot represent the 3D grid dimensions. The top and bottom plots are for 2 and 8 nodes respectively (8 cores per node).

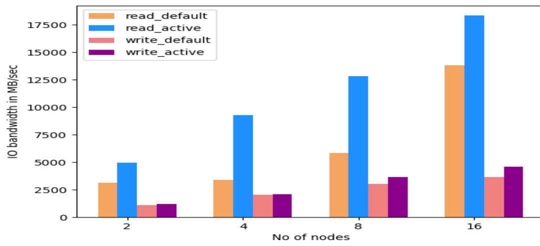


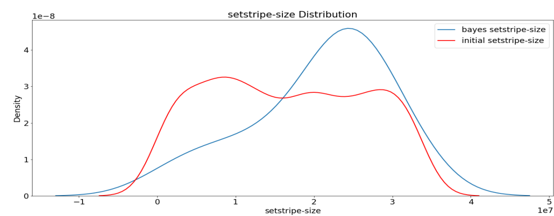
Fig. 6: Weak scaling results for S3D-IO on 2 – 16 nodes of Cori (32 processes per node)

Benchmark	MdAPE (%)		R^2	
	Read	Write	Read	Write
S3D-IO	23.72	10.13	0.50	0.88
BT-IO	21.24	19.23	0.45	0.79
IOR	30.58	10.25	-0.20	0.47
GenericIO	12.22	13.42	0.42	0.24
S3D-IO	8.01	7.25	0.90	0.91

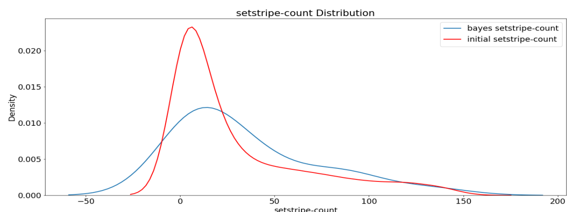
TABLE II: Predict performance on HPC2010 (rows 1 – 4) and Cori (last row)

R^2 score of read and write for IOR can be non-diverse data collected while running ExAct (which we used as training set for Predict). A better training set can improve the result. We plan to investigate this in future.

3) Prediction-based Auto-tuning (PrAct): The read and write bandwidth prediction model (using XGB) is used to predict read and write bandwidths in PrAct (§IV). For each input configuration, PrAct runs 100 iterations to converge. The output are the values of the I/O hints that may yield high I/O bandwidths for the system on which Predict was trained. With these hints, we run the application code to get PrAct results. The results are shown in Fig9(a) and Fig9(b) for S3D-IO and

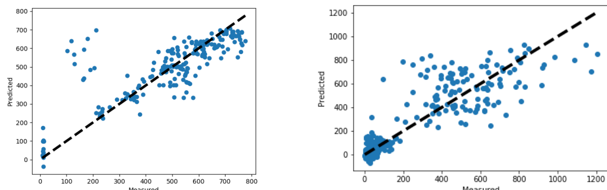


(a) Probability density of stripe_size.

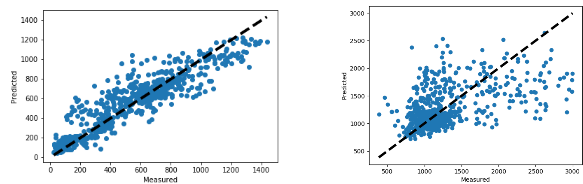


(b) Probability density of stripe_count.

Fig. 7: Training distributions for ExAct. Red and blue curves show density distribution before and after training of model respectively. Note: The curves are plotted by sampling.



(a) IOR (b) BT-IO

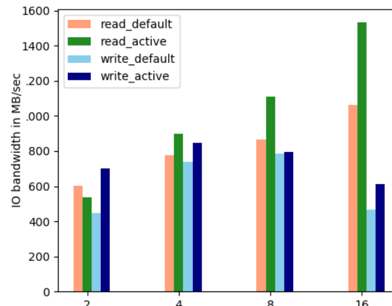


(c) S3D-IO (d) GenericIO

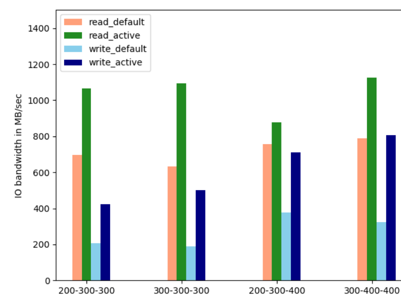
Fig. 8: Scatter plots of XGB-predicted values vs. measured values of write bandwidths for IOR, BT-IO, S3D-IO and GenericIO on HPC2010 for 30/70 split of train/test data. The ideal graph should be dashed black line; the actual values are denoted by blue dots.

BT-IO. The PrAct output is compared with the runs using default parameters. The read and write bandwidths (dark green and dark blue in the figures) are obtained by executing with hints obtained by running PrAct with predicted bandwidths (from Predict) as objective function. Separate set of hints were used for read and write bandwidths as recommended by PrAct.

We observe that BT-IO I/O bandwidths are higher than the default with this model over various configurations (x-axis). PrAct achieved a maximum of 1.6× and 1.2× performance improvement in reads and writes in S3D-IO, and a maximum of 1.7× and 2.5× performance improvement in reads and writes in BT-IO. We ran both the benchmarks over various configurations which were not present in our training or test data set. We observe that PrAct is able to perform well even for these new configurations with the predictions. Note that we ran the experiments for a few time steps. However, production runs are long-running and our models can help in reducing the I/O cost by a few hours, when ran in production mode.



(a) S3D-IO weak scaling on unseen configurations.



(b) BT-IO with unseen configurations.

Fig. 9: PrAct results for S3D-IO and BT-IO.

We observed degradation in read bandwidths in case of IOR, especially at high node counts. This is expected as the R^2 scores were low (refer Table II) for IOR, which shows prediction model does not perform well. This is perhaps due to high variability for IOR runs as well as insufficient training data for learning. Average training time of PrAct is 18 seconds whereas average training time for ExAct is 800 seconds which is lower than the existing models. Note that PrAct reduces the training time significantly because the active learning model in PrAct uses the predicted values rather than the actual runtimes of the application. We get good predictions for many of the configurations, and therefore PrAct is able to find the best parameters for new configurations as well.

VI. Conclusions and Future Work

We presented two active learning-based auto-tuning approaches, ExAct and PrAct, to tune the MPI-IO and Lustre parameters. The only system-specific input to the model is the range for stripe count, since the maximum number of OSTs vary per system. We also presented a performance model to predict the I/O bandwidths. All our models are application-agnostic and can be trained for any benchmark with negligible effort. We have demonstrated upto 11× improvements in read and write bandwidths with both ExAct and PrAct for a few important I/O benchmarks and application kernels on two supercomputers (up to 128 cores on HPC2010 and 1024 cores on Cori). Execution-based active learning model, ExAct, obtains a speedup of 200%

on an average for various benchmarks in just 20 iterations. ExAct runs the application and learns, whereas PrAct uses predicted values from Predict to learn. Thereby, the training time of PrAct to find best parameters is drastically reduced from few hours (application-dependent) for ExAct to only 20 seconds on average. This is a tremendous improvement in training time over past models for auto-tuning. ExAct is able to improve write performance of large data sizes (1 billion particles in GenericIO) by $3\times$. Moreover, ExAct achieves an average bandwidth improvement of $2.1\times$ for non-contiguous writes in BT-IO. Thus, we demonstrate that active learning can indeed be useful for auto-tuning in HPC.

Predict model uses XGBoost, and obtains less than 20% median prediction errors for most cases, even with 30/70 train/test split where results are averaged by running over 10 different set of train/test splits. Since the data set for Predict is obtained from the ExAct runs, the input training data is at times insufficient for the model to learn in case of few applications. This can be further improved by feeding more input data to Predict, which we plan to experiment in future. The loss function is the overall runtime of the application, the model tries to minimize loss for overall execution, thereby tuning parameters for both read and write simultaneously. In some cases the read bandwidth improvements were better, and in some cases the write bandwidth improvements were better. In future, we plan to tune reads and writes separately.

VII. Acknowledgments

We thank the High Performance Computing Facility of the Indian Institute of Technology Kanpur funded by the Department of Science and Technology and IIT Kanpur. We also thank the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 33–44.

[2] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of petascale I/O workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009.

[3] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir *et al.*, "Taming parallel I/O complexity with auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 68.

[4] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel I/O autotuning with performance modeling," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, 2014, pp. 253–256.

[5] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing I/O performance of HPC applications with autotuning," *TOPC*, vol. 5, no. 4, pp. 15:1–15:27, 2019.

[6] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in High-Performance Computing Applications," *Proceedings of the IEEE*, vol. 106, no. 11, 2018.

[7] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 33–44.

[8] M. Howison, , Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf, "Tuning HDF5 for Lustre file systems," 2010.

[9] R. McLay, D. James, S. Liu, J. Cazes, and W. Barth, "A user-friendly approach for tuning parallel file operations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 229–236.

[10] E. K. Lee and R. H. Katz, "An analytic performance model of disk arrays," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 21, no. 1. ACM, 1993, pp. 98–109.

[11] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "Cost-intelligent application-specific data layout optimization for parallel file systems," *Cluster computing*, vol. 16, no. 2, pp. 285–298, 2013.

[12] K. J. Barker, K. Davis, and D. J. Kerbyson, "Performance modeling in action: Performance prediction of a Cray XT4 system during upgrade," in *International Symposium on Parallel Distributed Processing*, 2009.

[13] M. Matheny, S. Herbein, N. Podhorszki, S. Klasky, and M. Taufer, "Using surrogate-based modeling to predict optimal I/O parameters of applications at the extreme scale," in *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 568–575.

[14] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and P. Hovland, "Collective I/O tuning using analytical and machine learning models," in *IEEE International Conference on Cluster Computing*, 2015.

[15] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting output performance of a petascale supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 181–192.

[16] "Cori," http://www.nersc.gov/users/computational-systems/cori.

[17] "Lustre," http://www.lustre.org/.

[18] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[19] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for automatic performance tuning," in *Proceedings of the International Conference on Computational Science*, ser. LNCS, vol. 2073. San Francisco, CA: Springer, May 2001, pp. 117–126.

[20] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian Optimization Algorithm," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, 1999, pp. 525–532.

[21] P. W. Koch, B. Wujek, O. Golovidov, and S. Gardner, "Automated Hyperparameter Tuning for Effective Machine Learning," 2017.

[22] Automated Hyperparameters tuning. [Online]. Available: https://www.kaggle.com/willkoehrsen/automated-model-tuning

[23] P. I. Frazier, "A Tutorial on Bayesian Optimization," *arXiv e-prints*, p. arXiv:1807.02811, Jul 2018.

[24] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[25] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," *CoRR*, vol. abs/1603.02754, 2016. [Online]. Available: http://arxiv.org/abs/1603.02754

[26] G. Congiu, S. Narasimhamurthy, T. Sub, and A. Brinkmann, "Improving Collective I/O Performance Using Non-volatile Memory Devices," in *IEEE International Conference on Cluster Computing*, 2016.

[27] Hyperopt. [Online]. Available: https://github.com/hyperopt/hyperopt/wiki/FMin

[28] IOR Code. [Online]. Available: https://github.com/hpc/ior

[29] "IOR Documentation," https://buildmedia.readthedocs.org/media/pdf/ior/latest/ior.pdf.

[30] "S3D-IO code," https://github.com/wkliao/S3D-IO.

[31] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki *et al.*, "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science & Discovery*, vol. 2, no. 1, 2009.

[32] "Genericio," https://trac.alcf.anl.gov/projects/genericio.

[33] W.-k. Liao, "Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 260–272, 2011.

[34] HPC2010. [Online]. Available: https://www.hpc.iitk.ac.in

[35] W.-k. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[36] Xgboost. [Online]. Available: https://xgboost.readthedocs.io/en/latest/