HPC I/O Throughput Bottleneck Analysis with Explainable Local Models

Mihailo Isakov*, Eliakin del Rosario*, Sandeep Madireddy†,
Prasanna Balaprakash†, Philip Carns†, Robert B. Ross†, Michel A. Kinsy*
* *Adaptive and Secure Computing Systems (ASCS) Laboratory*
Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843
{mihailo,eliakin.drosario,mkinsy}@tamu.edu
† *Argonne National Laboratory*, Lemont, IL 60439
smadireddy@anl.gov, {pbalapra,carns,rross}@mcs.anl.gov

***Abstract*—With the growing complexity of high-performance computing (HPC) systems, achieving high performance can be difficult because of I/O bottlenecks. We analyze multiple years' worth of Darshan logs from the Argonne Leadership Computing Facility's Theta supercomputer in order to understand causes of poor I/O throughput. We present Gauge: a data-driven diagnostic tool for exploring the latent space of supercomputing job features, understanding behaviors of clusters of jobs, and interpreting I/O bottlenecks. We find groups of jobs that at first sight are highly heterogeneous but share certain behaviors, and analyze these groups instead of individual jobs, allowing us to reduce the workload of domain experts and automate I/O performance analysis. We conduct a case study where a system owner using Gauge was able to arrive at several clusters that do not conform to conventional I/O behaviors, as well as find several potential improvements, both on the application level and the system level.**

***Index Terms*—HPC, I/O, diagnostics, machine learning, clustering**

I. Introduction

Because of the scale and evolving complexity of high-performance computing (HPC) systems, critical gaps still remain in our understanding of HPC applications' runtime behaviors, specifically, compute, communication, and storage behaviors. This situation is further complicated by the fact that HPC applications come from a diverse set of scientific domains, can have vastly different characteristics, and are executed simultaneously, thereby contending for shared resources.

One such gap is the understanding of I/O utilization in these systems. Currently, application programmers and systems administrators still heavily rely on limited observations, anecdotes, and scattered experiences to develop design patterns for applications and manage their runtime performance either at the node level or at the system level. This approach is tractable only to the extent permitted by limited application developer and facility support staff resources and their expertise. Therefore, automated data-driven methods are needed to streamline this process and reduce the turnaround time from capturing information to understanding and enacting improvements in I/O utilization and efficiency. One intuitive way to approach this situation is not by simply considering application performance in isolation but by identifying commonalities that reduce the volume of characterization data, simplify performance modeling efforts, and exploit opportunities for performance improvement across application domains.

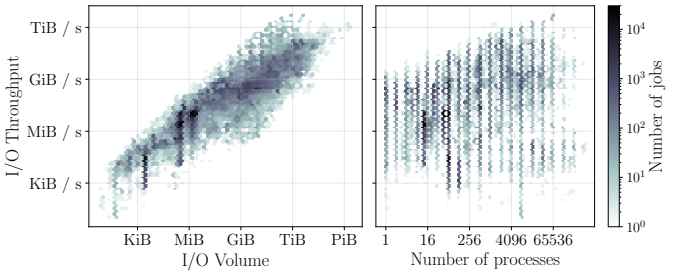

Fig. 1: Frequency of jobs with respect to I/O throughput and the total number of bytes transferred. Data are collected from the Argonne Leadership Computing Facility (ALCF) Theta supercomputer. Note that the color bar is logarithmic.

Machine learning (ML) is a promising approach for the data-driven analysis of I/O performance data. This is evidenced by the growing interest in the design and development of ML-based methods for various I/O performance analysis and modeling tasks [1]–[6]. However, analyzing I/O performance is not trivial. Figure 1 shows that I/O throughput spans almost 14 orders of magnitude and can vary as much as five orders of magnitude for jobs with the same amount of I/O volume. Given the complexity of the I/O performance data, the relationship between the I/O performance and the factors that affect it are often nonlinear. Consequently, there is a trade-off between explainability and predictive accuracy when out-of-the-box ML methods are adopted for I/O performance analysis. In particular, the models that are explainable and intuitive to I/O experts are often simple and based on linear models. The models that have high predictive accuracy are often black box and cannot be used directly for explaining the I/O performance.

We develop an explainable ML platform for I/O performance analysis to answer a number of I/O performance questions: how can we cluster applications together? Given an application or task execution, what existing I/O behavior cluster does the job fall into? What are the key characteristics of the cluster itself? Does it match the expected execution or performance profile, for example, the requested resources

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

and the optimality of those resources' utilization? How does this job's performance rank with the rest of the cluster? What parameters influence the job placement within the cluster? How does the cluster rank with other clusters?
The goal of this work is to answer these questions, and to this end our contributions are as follows:

- We introduce a log-based feature engineering pipeline for HPC applications. Our analysis uses 89,844 Darshan logs of I/O volume greater than 100 MiB collected on the Argonne Leadership Computing Facility (ALCF) Theta supercomputer from 2017 to 2020.
- We show that agglomerative clustering can reveal a large amount of structure in the dataset and that training models on fine-grained (local) clusters instead of on the whole dataset yields more robust and useful predictions.
- Using different ML methods, we demonstrate that despite I/O throughput varying across many orders of magnitude, we can on average predict individual job I/O throughput within $\sim$20% of the real value. Furthermore, we show how the interpretation of ML prediction models can yield useful advice for increasing application performance.
- To validate the practicality of the proposed feature engineering pipeline and the clustering techniques, we introduce Gauge, an exploratory I/O throughput analysis tool with adjustable data granularity and interpretable I/O throughput models. We illustrate how it can be used by system owners and I/O experts to optimize the HPC clusters for the workload present or by application developers to optimize their jobs.
- We release a web-based version of Gauge, information about the tool is available at http://ascslab.org/research/gauge.

II. Exploration of the Application Space

A. Darshan Log Dataset Generation

Darshan [6] is an HPC I/O characterization tool that transparently captures I/O access pattern information about applications running on a system. It collects information such as numbers of POSIX operations, I/O access patterns within files, and timestamps of events such as opening or closing files. We have collected 661,553 Darshan logs from the ALCF's Theta supercomputer, ranging from January 2017 to March 2020. Since using Darshan is optional and as many legacy applications do not support it, our dataset covers only 28% of all jobs run on Theta.

For each job, Darshan can be used with a number of instrumentation modules, such as POSIX, MPI-IO, stdio, HDF5, and Lustre. In this work, since we focus on I/O characterization, two modules are of interest: POSIX and MPI-IO. We choose to use only POSIX for two reasons: (1) since the MPI-IO layer passes through the POSIX layer, POSIX operation counters are strictly equal to or larger than that of MPI-IO and (2) MPI-IO requires POSIX to be enabled, but only 28.2% of the jobs instrumented with POSIX are also instrumented with MPI-IO.

The POSIX module measures 86 features, such as number of bytes read and written; number of accesses per each of the 10 bins; number of consecutive and sequential read and write operations; file and memory aligned operations; sizes and strides of the top four most common POSIX access sizes; number of common POSIX calls such as `seek()`, `stat()`, and `fsync()`; cumulative time spent reading, writing, and in other operations such as `seek()`, `stat()`, and `fsync()`; and timestamps of the first and last POSIX open/read/write/close operations. More information can be found in the Darshan-util documentation [7]. On top of this set, we appended 10 Darshan metaparameters including the job's I/O throughput; number of unique or shared files opened; and number of read-only, read/write, and write-only files. Logs containing all of these features serve as the raw dataset, which we process with a data sanitization and normalization pipeline described below.

B. Sanitization and Normalization Pipeline

Our data preprocessing pipeline consists of several steps:

1) **Data sanitization**: In this step, we remove jobs that are not instrumented with POSIX or have invalid values (e.g., negative values). Negative values are typically present when a job was not closed properly or when a hardware fault occurred. Similarly, we remove features that have a large number of missing values. These typically arise since the version of Darshan running on Theta has changed over the years and new features were introduced. Hence, we choose to ignore these new features.
2) **Feature pruning**: We remove common access size features and all time-sensitive features. Common access size features store the most common access sizes in bytes. Because of the difficulty in converting these values to a more ML-digestible format, we choose to remove them. Time-sensitive features measure timestamps such as when files are first opened or closed last. The rationale for removing them is that Darshan uses some of these features in calculating job throughput. By leaving them in, we risk that an ML model might pick up Darshan's implementation details, instead of the wanted insight. In Section III-B, we provide evidence that models trained on datasets that have only five features (four time-based features and I/O volume) significantly outperform models trained on datasets with all non-time-based features (Table I). Note that other than time-sensitive features, all other features are a function of the application and input parameters; that is, these values are largely independent of the actual system the application is running on.
3) **Data normalization**: We apply feature engineering to force the values to a more manageable range. Quick investigation shows that the majority of features in our dataset have values in a wide range; for example, the total number of bytes a job has transferred can vary from tens of bytes to multiple petabytes—almost 15 orders of magnitude. This distribution is consistent across features; and without treating it in a special way, it is hard to use ML methods on such wide ranges of values. To tackle this problem, we convert the majority of the features from absolute values

to values relative to some other features. We can do so because many of the features represent quantities that are portions of other quantities. For example, Darshan records both the number of read operations and the number of consecutive and sequential reads. Therefore, since the last two features are at most equal to the number of reads, we can convert them to a percentage of total reads. Arguably, this approach cannot be universally applied; for example, the number of POSIX seek operations cannot be expressed as a ratio of a different feature. To tackle this situation, we replace the values of feature f with $log_{10}(f)$. We use log_{10} instead of ln since it is simpler to interpret and translate back to the original value. Since all of the features in the dataset are positive or zero, we increment the feature by a small constant (e.g., 10^{-5}) so that the logarithm is always defined. Doing so forces the values into a more controllable range: a majority of the values lie in (-5, 12).

We are left with 45 percentage features and 12 logarithmic features, not counting I/O throughput (also logarithmic). In Table I we give a brief overview of several sets of features, and we refer the reader to our open-source repository with the experiments for this work [8]. We also prune the set of jobs. From the 661,553 collected jobs, we first discarded 163 jobs that contained corrupted instrumentation data, 284,464 jobs (43.0%) for which Darshan did not instrument POSIX calls, and 287,082 jobs (43.3%) that have less than 100 MiB of total I/O volume, leaving us with 89,844 (13.6%) jobs. Since these small jobs occupy a fraction of the total traffic [9] (small jobs transferred 974 GiB in total, while large jobs transferred 58.9 PiB), in this work we focus on analyzing large jobs. In future work, we plan to investigate both large and smaller jobs, as well as the impact of many smaller jobs on the system.

Note that because we do not have full visibility into the system, we are unable to reconstruct the system's I/O utilization at a given timestamp. Therefore, our analysis is geared more toward explaining internal reasons for a job's I/O throughput (e.g., by detecting good or bad I/O patterns), and less on external reasons (e.g., I/O contention).

C. Exploring Dataset Structure

Once we have preprocessed the dataset, it is ready for analysis. Since every job in the dataset can be represented as a point in a 57-dimensional space and since we have only ∼90,000 jobs, the majority of this space is unoccupied. Furthermore, since these jobs are derived from a small number of applications (the top 6 applications account for 50% of all jobs), we expect that the majority of the dataset exists in clusters. Since we expect the jobs to sparsely occupy this high-dimensional space, we are interested in exploring what underlying structure the dataset has and whether we can exploit any statistical properties to reduce the dimensionality of the data.

To better understand structure in the dataset, we need a metric to compare individual jobs. Selecting one is nontrivial because the features are heterogeneous (percentage ones are relative and logarithmic ones are absolute). In Figure 2 we show the Manhattan (L1) and Euclidean (L2) distances when comparing 96 jobs from the top six applications. Both metrics result in similar distance matrices, albeit at different scales. Here we used single linkage clustering—a type of hierarchical clustering—to group similar jobs together. This can be seen by the application colors on the left and top of the distance matrix. While we have started with jobs sorted by application, the clustering has rearranged rows and columns to form the clusters, namely, the series of dark boxes on the main diagonal. Note that the quality of clusters varies—not all are equally dark, and some clusters consist of jobs only vaguely similar. We note that often jobs from the same application can be very different; that is, multiple clusters exist within applications. Furthermore, jobs from different applications can often be more similar than jobs within the same application, as can be seen with the cosmology and fluid dynamics applications. Note that without the feature normalization described previously (though not pictured in Figure 2), the differences between jobs are far more pronounced, even within the same application.

TABLE I: Condensed Feature Set

Feature set	Count
Logarithmic features	
log_{10} of the total number of {files, accesses, bytes}	3
log_{10} of the number of POSIX {`open`, `seek`, `stat`, `mmap`, `fsync`, `mode`} calls	6
log_{10} of the number of processes	1
log_{10} of {memory, file} alignment in bytes	2
Ratio features	
% of all accesses that are {reads, writes}	2
% of all {reads, writes} that are {consecutive, sequential}	4
% of all accesses that switch between reading and writing	1
% of {read, write} accesses of size in ranges [0B, 100B], [100B, 1KiB], [1KiB, 10KiB], [10KiB, 100KiB], [100KiB, 4MiB], [4MiB, 10MiB], [10MiB, 100MiB], [100MiB, 1GiB], 1GiB+	20
% of accesses in the {1, 2, 3, 4}-th most common access size	4
% of non-aligned {file, memory} accesses	2
% of all bytes that are {read, written}	2
% of {shared, unique, read-only, read-write, write-only} files	5
% of bytes read/written from {shared, unique, read-only, read-write, write-only} files	5

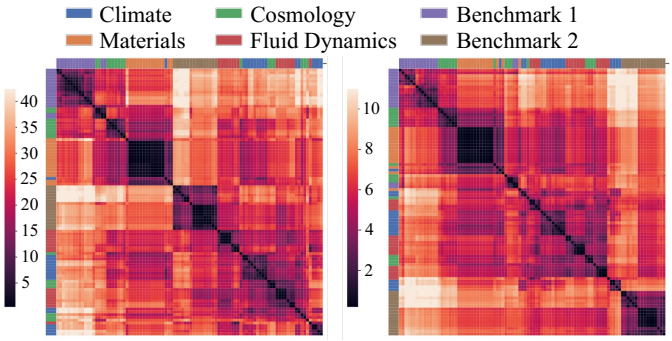

Fig. 2: Manhattan (left) and Euclidean (right) distance matrices of 96 jobs chosen at random from 6 I/O-intensive applications, with 16 jobs per application. Each row and column represent a job belonging to an application denoted by the left or top border color.

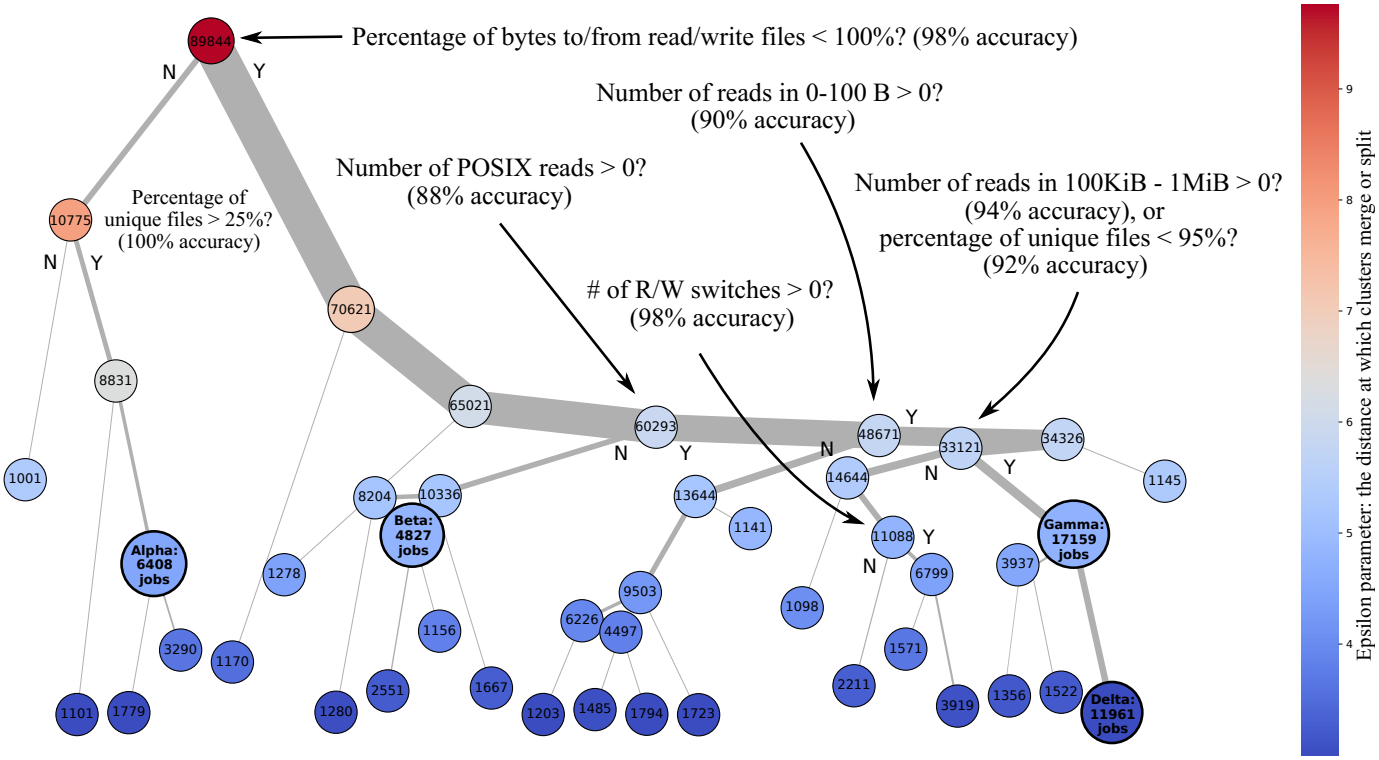

Fig. 3: HDBSCAN single linkage tree, pruned of clusters smaller than 1,000 jobs and of clusters clustered at $\epsilon < 3$. Note the four clusters marked Alpha to Delta. These clusters are hand-selected using this tree and are used later in the analysis.

The existence of these clusters and of clusters of varying densities motivated us to further explore the structure of the dataset. For that purpose, we used HDBSCAN [10], a hierarchical version of the DBSCAN agglomerative clustering algorithm. Briefly, DBSCAN works by clustering together points within a certain distance ϵ and connecting graphs of these neighboring points into larger clusters. DBSCAN is highly sensitive to the choice of this distance parameter ϵ, with small values leading to many small "islands" of jobs and large values leading to the whole dataset being placed in a single cluster. HDBSCAN allows us to visualize what the original DBSCAN algorithm would arrive at clustering at any ϵ value. HDBSCAN is sensitive to a second parameter k, which controls the minimal size of clusters. As we decrease ϵ, clusters will periodically split; and when these smaller "child" clusters contain less than k points, their contents will be treated as outliers. In our experiments, we use the default value of $k = 5$. A small k value allows us to explore clusters with fine granularity. In Figure 3, we show HDBSCAN's single linkage tree, with both color and position of nodes on the y axis specifying the ϵ value at which the cluster merges and splits. We can see that the whole dataset gets clustered in a single cluster for large ϵ values and that, as we decrease the value, the cluster gets progressively split into smaller and smaller clusters. The lines connecting the clusters specify which clusters get split into or merge into which, with the line thickness being proportional to the number of jobs going to that cluster. The values in clusters specify the number of jobs in that cluster. The number of jobs between parent and children clusters may not always add up; some jobs are lost as they belong to smaller clusters that we do not plot, in order to avoid visual clutter.

Note that we have four clusters with larger circles marked Alpha, Beta, Gamma, and Delta. We have selected these clusters for further analysis in Section V. These specific clusters were selected because they are far from each other in the tree and hence may have very different behaviors. Additionally, we wanted to explore the impact of granularity in our analysis. That is why we have selected the cluster Delta (bottom right) to be a sub-cluster of cluster Gamma (right). Gamma contains all of the jobs in Delta, plus 5,000 other jobs.

D. Interpreting Dataset Structure

While Figure 3 reveals the existence of a rich structure in the dataset, it does not increase our understanding of it. To get better intuition about the data distribution, we perform a simple experiment: since every node of the HDBSCAN single linkage tree consists of a number of smaller merged clusters, we train a decision tree that predicts where each job in the cluster will end up once the cluster splits. To help interpretability, we train decision trees only of depth 1, namely, trees with only 2 leaves and a single decision splitting the dataset. The annotations and arrows pointing to nodes in Figure 3 explain what the decision trees at those nodes have learned.

Right away, the clustering splits the dataset into jobs that

use only read/write files and jobs that also use read-only and write-only files (top cluster's annotation). Although this split is imbalanced (70K jobs vs. 10K jobs), the accuracy of the decision tree (98%) is high enough to still be informative. Looking at its smaller child (node with 10,775 jobs), we see that it splits the dataset with perfect accuracy into jobs that have more or less than 25% of unique files (unique files are files accessed only by a single process). Note that other nodes' decisions might not be so accurate. This is due to our choice of using a decision tree with a depth of 1. If we allow deeper decision trees, the accuracy of decisions increases, but interpreting these models becomes more tedious. In practice, we use the HDBSCAN tree in a more interactive process, allowing us to test different decision trees using different features, depths, and acceptable accuracies.

Through analyzing this tree in more depth, we concluded that the clusters in the dataset occupy very distant spaces, showing different behaviors across several and often tens of different features. As we decrease ϵ, the clusters get smaller, and the jobs within them more and more similar, until we arrive at just dozens of almost identical runs. We claim that to analyze throughput and extract insight out of a cluster, first we must select the right granularity at which to make the analysis. Too fine, and we may be learning the behavior of a single job, not any general trends shared by different applications running on the HPC system. Too coarse, and we may arrive at very general interpretations that are not specific to the jobs we are interested in.

III. I/O Behavior Characterization

Before we can explore why I/O throughput behaves a certain way, we first seek to predict it. Once we have a machine learning (ML) model that does a good job of predicting I/O, we can use the trained model to extract insight.

A. Evaluating I/O Throughput Predictions

To predict I/O performance, we first need to define a metric for evaluating predictive accuracy. Common metrics such as L1 or L2 loss might not be a good fit for our data because the throughput ranges from KiB/s to TiB/s and hence will penalize jobs with higher throughput more than the low- and medium-range ones, forcing our models to ignore the latter. Therefore, we adopt root mean squared logarithmic error (RMSLE) and mean absolute logarithmic error (MALE) functions:

$$\begin{aligned} RMSLE(y,\hat{y}) &= \sqrt{\frac{1}{n}\sum_{i=0}^{n}(log_{10}(y_i)-log_{10}(\hat{y}_i))^2} \\ &= \sqrt{\frac{1}{n}\sum_{i=0}^{n} log_{10}\left(\frac{y_i}{\hat{y}_i}\right)} \end{aligned} \tag{1}$$

$$\begin{aligned} MALE(y,\hat{y}) &= \frac{1}{n}\sum_{i=0}^{n}|log_{10}(y_i)-log_{10}(\hat{y}_i)| \\ &= \frac{1}{n}\sum_{i=0}^{n}\left|log_{10}\left(\frac{y_i}{\hat{y}_i}\right)\right|. \end{aligned} \tag{2}$$

Both of these equations penalize the *ratio* of the prediction vs. the real value. Hence the ML model is scale independent and should equally penalize relative prediction errors on both small and large throughput jobs. Since our ML models already receive percentage and logarithmic features and are tasked with predicting the logarithmic I/O throughput value, they can natively use MSLE/MALE; that is, we never have to convert the predictions back to the raw values. In subsequent text, we choose to use MALE over RMSLE, since MALE is less sensitive to outliers and is more directly interpretable. Using MALE allows us to directly translate median errors to English, for example, by saying that the model on average predicts I/O throughput with $1.15\times$ error. For example, if a model predicts a throughput of $13GiB/s$ for a job that in reality achieves only $10GiB/s$, the MALE error is $MALE(10\times10^9, 13\times10^9) = |-0.114|$. To interpret this loss, we calculate the relative error $10^{|-0.114|} = 1.3\times$. The same value is calculated if we swap the prediction and target value; that is, this model may underestimate or overestimate throughput. In either case, we have a good estimate of the range of the target value.

B. Modeling I/O Performance

We now attempt to create accurate I/O throughput prediction models. The goal of modeling I/O performance, instead of simply observing it, is to potentially develop accurate models and analyze them for underlying causes of over-/underperforming I/O throughput. If the model has good predictive power and generalizes well (can accurately predict I/O throughput of new jobs), we can apply ML model interpretation and explanation methods to answer questions such as what parameters influence this job's I/O throughput the most and what steps we should take to improve performance. We have evaluated a number of different types of ML models, such as linear regression, decision trees, random forests, gradient boosting machines, and neural networks, and have chosen to use XGBoost [11] for our predictions. XGBoost is a gradient boosting machine library that shows excellent performance on tabular data and is simpler to tune compared with other powerful models such as neural networks.

In Figure 4, we present training and test errors of an XGBoost model trained on the whole training set (box plots on the right marked "Global"), as well as a number of XGBoost models trained on clusters of various granularities (discussed later). The global model achieves a median error of less than $1.2\times$; that is, half of the predictions are less than $1.2\times$ off the true value. Through discussion with HPC domain experts and system owners, we learned that this sort of misprediction is acceptable, since I/O throughput can vary by orders of magnitude. Furthermore, pushing this error lower may be difficult because of external, unobservable factors such as I/O weather [12]. Since our analysis here does not take into account the system's I/O contention present during the job's execution, we fundamentally cannot achieve better predictions than I/O weather would allow. For brevity, we leave the analysis and discussion of I/O weather's impacts on our models to future work.

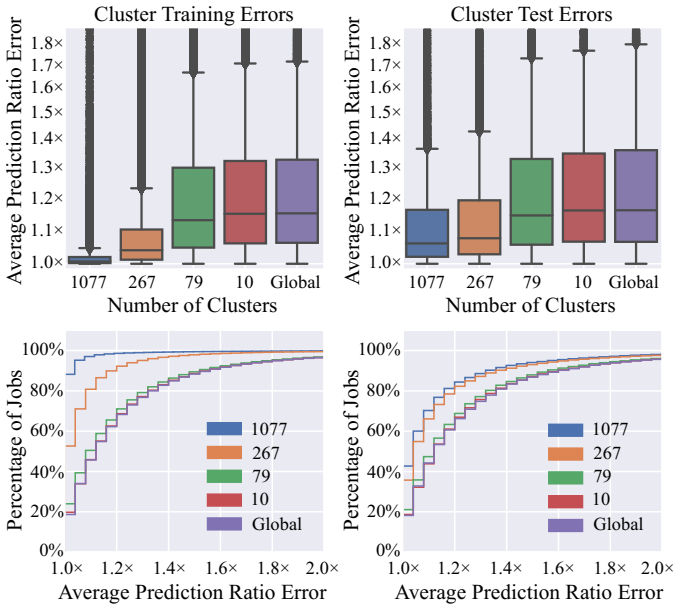

Fig. 4: I/O throughput prediction results. Top row shows the training and test error distribution of XGBoost models trained on clusters of various granularities. The rightmost model is a single model trained on the whole dataset. The bottom row shows the cumulative histogram of the above error.

Given that the global model is performing well, we seek to gain insight into how the model is arriving at its predictions. To do so, we use the permutation feature importance (PFI), a statistical method [13] that evaluates the impact of every feature on the trained model output. The method works by taking the trained model, selecting a feature whose importance we want to test, and permuting its values from different rows. This approach guarantees that the input distribution of that feature remains the same but carries no information. It permutes values instead of, for example, zeroing out that feature, since this may additionally hurt the model's predictive performance. In Figure 5, we show the top features selected by PFI using XGBoost predictors, on two different datasets (blue and orange). The first PFI experiment (orange line and labels) shows feature importance of models trained on a dataset consisting of all jobs larger than 100 MiB and with features from Table I. The second PFI experiment (blue line and labels) uses the same jobs but now has ten additional time-based features: runtime; cumulative read, meta, and write time relative to runtime; maximum read / write operation durations relative to runtime; and time periods between first and last open, close, read, and write operations relative to runtime. The figure can be interpreted as follows: For each row, the classifier is trained on that row's selected feature, in addition to the features from the rows above. For example, the third row's orange XGBoost predictor is trained on the total number of bytes, read / write accesses, and percentage of reads in the [1, 10] KiB range.

First, let us look at PFI values on the original (orange) dataset, which does not contain time-based features. We briefly analyze why these features may have been selected. We note that important features do not necessarily imply that increasing those feature values is beneficial for performance; in other words, strong negative correlations will also be treated as important. The total number of bytes transferred is selected as the most important feature. The reason is that I/O-heavy applications typically have higher throughput than smaller applications have (Figure 1), so just from volume the model can narrow down throughput to two orders of magnitude. Throughput can scale with volume for a variety of reasons, for example, because larger applications being able to amortize some of the slower operations or because more effort may be spent on optimizing more intensive applications. The number of read and write calls likely helps the model estimate the average transfer size, another rough predictor of throughput. Similarly, small accesses in the 1–10 KiB range are likely correlated with low throughput. The number of files can be important, especially for applications that open one file per process (in Section V we analyze a cluster of such jobs). Consecutive and sequential reads and writes are preferable to random access operations. Other features are less intuitive, such as the number of `stat()` calls or why PFI selected the percentage of the third most common access size vs. all accesses. One of the reasons some of the later features may be less intuitive is that XGBoost's performance is relatively constant once the top 6 or 7 features are provided. If adding less important features only slightly improves model predictions, PFI has a high chance of incorrectly ordering features. Therefore, we should not invest too much time analyzing features that are not in the top of Figure 5.

Looking at PFI results on the dataset that has additional time-based features, we see that just with the top four or five features the model can outperform any model trained on the dataset without time-based features. The reason is likely due to the model learning the details of Darshan's I/O throughput calculation implementation. The model therefore relies heavily on time-based features and does not utilize the rest of the dataset. This reliance prevents us from using model explanation techniques on it, since in our experiments the techniques only reinforce that time-based features are important. Hence, in order to force the model to learn relationships between the I/O patterns and throughput, we remove the time-based features.

While the feature importance obtained here matches the insights from domain experts, it does not significantly change how we would analyze misbehaving jobs or systems. In other words, if we were to extract from the model advice such as "higher I/O volume is typically correlated with higher throughput," "consecutive reads and writes are faster than non-consecutive," or "unique files are preferable to shared files,"[1] this advice does not help with diagnosis. We therefore focus on developing more personalized, "local" advice that is applicable only to a specific family or cluster of jobs. By trying to detect

[1]In Darshan, unique files are files that are accessed only by a single process. If two or more processes access the same file, that file is considered shared.

motifs in parallel execution, we hope to arrive at a greater understanding of a smaller subset of jobs, instead of general understanding of all the jobs.

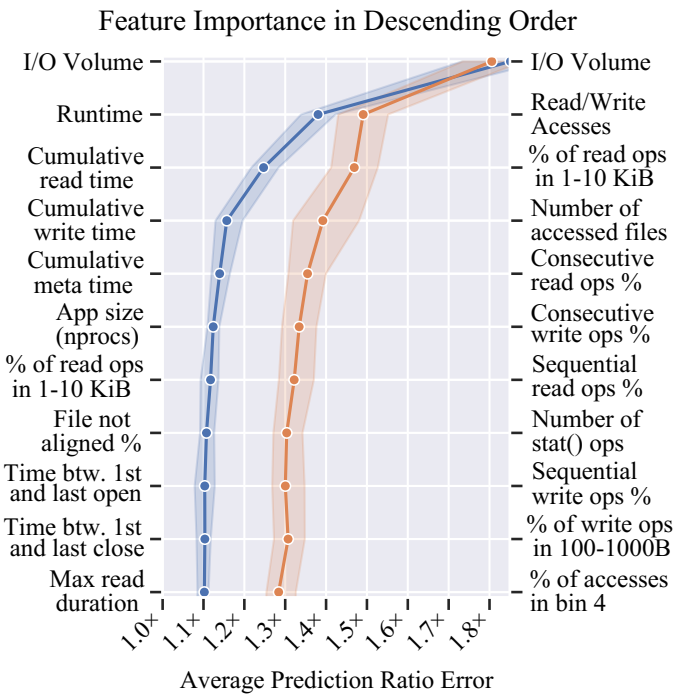

Fig. 5: Permutation feature importance (PFI) [13] of XGBoost models on two datasets. The blue line and labels represent PFI on a dataset with time-based features, and the orange line and labels represent PFI on a dataset without time-based features.

IV. I/O Model Interpretation

In this section, we focus on developing local models of I/O throughput and interpreting them. By local, we mean cluster-specific models, instead of the whole dataset. As seen in Figure 4, by using different ϵ values in DBSCAN clustering, we have selected several clustering granularities that result in splitting the whole dataset into different numbers of clusters. At each granularity, we split each cluster into a training and a test set, with a $70--30$ ratio. We train one XGBoost model per cluster and predict I/O throughput on the cluster's test set. In Figure 4, we present a box plot of the concatenated errors of all clusters. As we can see, with higher granularity we achieve better I/O predictions. Note that since we are evaluating the models on a different set of data from what the model was trained on, we are confident that the model is not simply memorizing job-throughput pairs but has generalized well enough. However, notice that in Figure 4 the global model and models trained on coarse-grained clusters (red and green bars) achieve similar performance on both the training and test sets. On the other hand, when the dataset is split into hundreds of clusters (orange and blue bars), where each of the per-cluster models is trained on a small portion of the total data, the models that have excellent performance on the training set have a considerably worse performance on the test set. This is evidence of overfitting, pointing to the conclusion that for smaller clusters we should use simpler models or stronger regularization. Even with possible overfitting, however, on the test set these small-cluster models achieve considerably better accuracies compared with the global model. Therefore we seek to interpret these local models. For the interpretations, we use SHapley Additive exPlanations (SHAP) [14], [15], a game theoretic approach to interpreting black-box ML models. SHAP allows us to gain insight into the impact of each feature on a per-job level, providing us with information not only about which features are important but also about how they affect the prediction and how they react to other features.

To apply SHAP on these local models, we design an interactive HPC job analysis tool we call Gauge. It allows system administrators and I/O experts to select clusters from the HDBSCAN tree from Figure 3 and plot each cluster's information on a dashboard. In Figure 6, we present a screenshot of Gauge's dashboard, showing information about four clusters, with one cluster per column. Gauge succinctly shows the general information about each cluster in the first two rows. Note that the same clusters that were highlighted in Figure 3 are plotted here. We have selected these clusters to better illustrate behavior of dissimilar jobs and clusters at different granularities (note the large difference in ϵ between the clusters, as shown in Figure 3).

A. Gauge Dashboard

The first row shows parallel plots of logarithmic features deemed most important by I/O experts. These plots allow the user to quickly gain insight into the cluster's I/O throughput volume, numbers of files, processes, and their relationships. Note the different units: we display throughput in MiB / s, volume in GiB, and the number of accesses in thousands, while numbers of processes and files are not modified. We selected MiB / s, GiB, and accesses in thousands so that the values can be plotted on the same logarithmic scale.

The second row shows another parallel plot, this time for ratio features. Note that jobs within the same cluster often have similar percentage features but differ in logarithmic ones. The reason is that jobs from, say, the same application may exhibit identical behavior but, because they were run with different inputs, show different IO volumes, throughputs, runtimes, and so forth.

The third row shows the error distribution of three ML models for predicting I/O throughput. The first is simply a median predictor—its prediction is a constant, selected as the median I/O throughput for the whole cluster. While trivial, this predictor is useful because it often outperforms other predictors once the cluster is very fine-grained, and it can serve as a baseline. The second classifier is a linear regression, and the third one is XGBoost. The linear regression performs well on medium-granularity clusters (with hundreds or thousands of jobs), where typically only one application's jobs exist in the cluster and XGBoost may overfit. For anything larger, XGBoost outperforms the constant and linear predictors, as can be seen in the third row of Figure 6.

The fourth row shows SHAP's summary plot for the ML

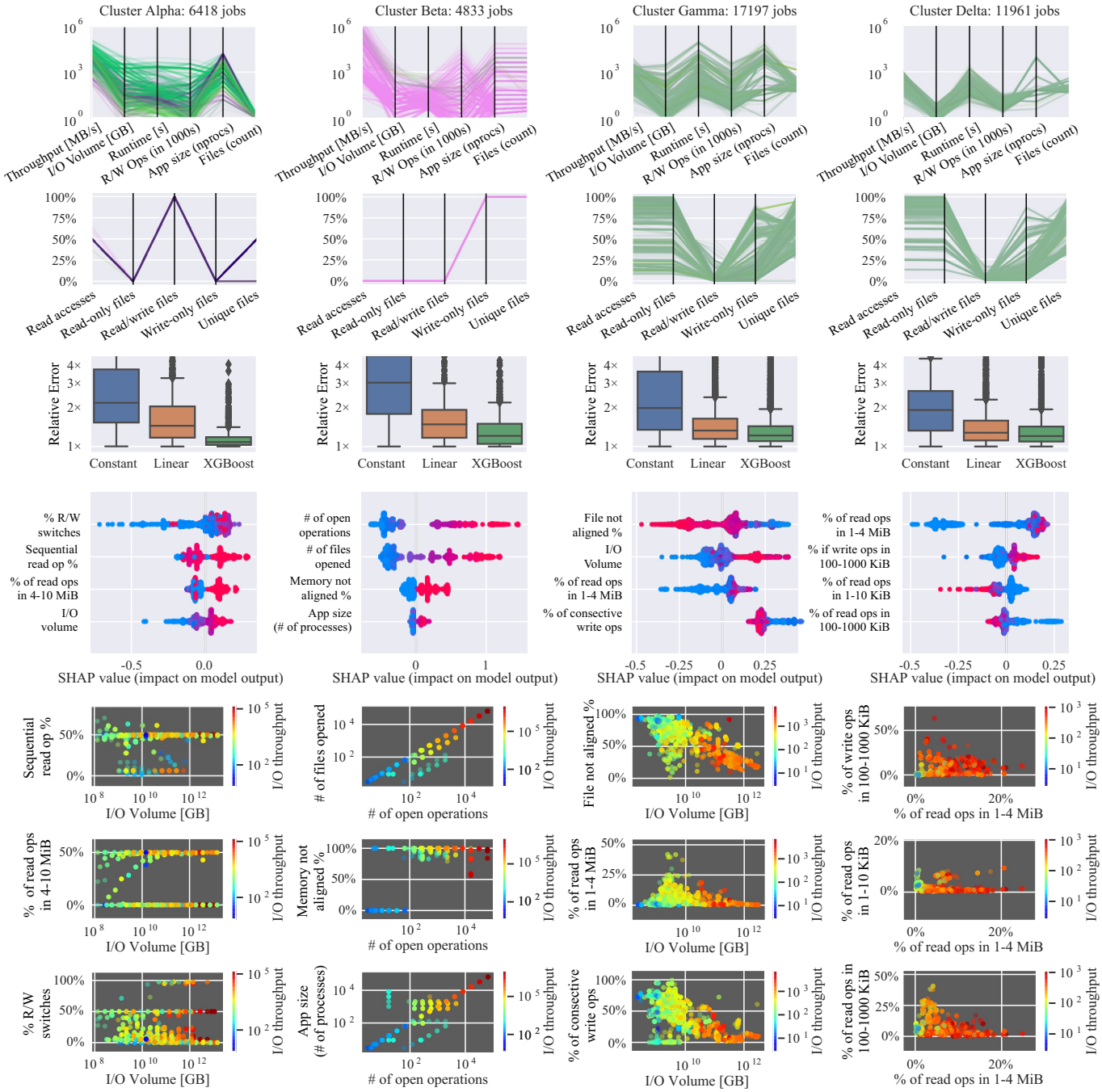

Fig. 6: The Gauge Dashboard. The first and second rows show parallel coordinate plots of the logarithmic and percentage features for four different clusters. The third row shows the performance of three different ML models trained on each cluster. The fourth row shows the SHAP summary plot for each of the clusters. The last three rows show scatter plots of features selected by SHAP, with the color indicating the I/O throughput.

models. These graphs should be interpreted as follows, There exist four rows per plot (though this is parameterizable), and each row represents a feature. Red markers correspond to jobs with higher values and blue markers to lower values of that feature. The position of these markers indicates SHAP's predicted impact of the feature on the model's prediction. Markers on the right indicate that that job's feature has a positive impact on predicted I/O throughput, and markers on the left indicate that the impact is negative. As an example, the first column's "Data volume row" has red markers (high values) on the right, indicating that higher data volume results in higher throughput for that cluster. Alternatively, the "% R Ops 1–10 KiB" row in the fourth column has red markers on the left. Thee mean that larger numbers of reads in the [1–10] KiB range result in decreased I/O throughput. Note that the scale allows us to compare the impact of different features on

I/O throughput. This is further analyzed in Section V.

Rows five, six, and seven show scatter plots of different features, colored with I/O throughput. The top feature according to SHAP is used for the x axis on all three scatter plots, while for the three y axes we use the second, third, and fourth most important features, respectively. These scatter plots are useful for getting a better understanding of any correlations or relationships between features. For example, looking at the bottom three plots in the second column (cluster Beta) we can immediately spot a linear relationship between the number of `open()` operations and the number of files accessed, as well as the number of processes. Gauge supports increasing the numbers of SHAP features and scatter plots, as well as using other types of plots such as correlation matrices showing feature correlations, but these are not shown because of space constraints.

Note that Gauge is an interactive tool. The user is expected to explore different clustering granularities (perhaps around a certain job or application) and different cluster sizes, analyze ML models at those granularities, compare local and global patterns, and explore the (often nonlinear) relationship between the features using the scatter plots. Next, we provide a case study using Gauge to analyze jobs from ALCF's Theta supercomputer.

V. Case Studies

The conventional approach to I/O performance analysis in the context of user/facility interactions is to work collaboratively with individual users to address specific concerns or improve the productivity of high-profile applications. This hands-on focus has proven effective in numerous examples [16]–[18] but fails to capitalize on the potential of guidance derived from broader contextual analysis:

- Does a given application conform to a contemporary I/O motif at *this facility* that is amenable to known optimizations?
- Would novel improvements to this application likely be applicable to other production applications?
- Is this I/O motif widespread enough to warrant strategic adjustments to provisioning or procurement?
- Beyond ad hoc user feedback, how can administrators allocate limited support resources for maximum impact?

In this section we focus on four candidate clusters identified using Gauge to illustrate how its capabilities could impact production workloads observed on the Theta system. The clusters are named Alpha to Delta and correspond to the large nodes with the same names in Figure 3 and columns in Figure 6.

A. Cluster Alpha

Cluster Alpha is notable because it goes against common access pattern expectations in that it is dominated by jobs that perform read/write access to almost all files. There are few pure read-only or write-only files. Read/write access to a single file could be caused by out-of-core computations, but in this case we see a different cause. The two most common codes in this cluster are an I/O benchmark application and a data science/ML application.

I/O benchmarks generally measure performance as the elapsed time needed to write a file (or files) and then the subsequent elapsed time needed to read it back. Its prominent presence in an unguided analysis of production I/O activity is not directly relevant to scientific productivity, but it may help a facility operator better understand the impact of performance measurement on the system.

The second most common code in this cluster is a data science/ML application, which is a relatively new type of workload for HPC systems. Its presence and the fact that it has a pointedly different access pattern from other clusters could potentially help inform strategic provisioning or procurement decisions that are more responsive to the mix of production applications running at a facility, for example, by providing more storage resources that are optimized for that particular workload.

The SHAP analysis shows a correlation between performance and the number of times that the job alternates between read and write access. This is not an intuitive correlation at first glance, but it may indirectly indicate applications that benefit most from caching of recently accessed data, which is also an indicator of potential procurement optimizations by providing burst buffer resources to such applications. Other correlations indicate positive associations with data volume (which can amortize startup and metadata costs), read access size, and sequential property of accesses.

The third scatter plot (first column, seventh row) indicates that increased data volumes improve performance regardless of how frequently applications switch between read and write access patterns.

B. Cluster Beta

Cluster Beta is a notable case study because it does not conform to conventional I/O tuning expectations in the SHAP analysis. The most prominent correlations identified by Gauge are *positive* correlations with the number of times that the `open()` system call was invoked and the number of files accessed: behaviors that are not intuitively associated with improved throughput. The first scatter plot (second column, fifth row) exhibits a diagonal pattern corresponding to jobs for which each file was opened exactly once, but there are also points below the diagonal that indicate jobs for which individual files were opened multiple times. The third correlation appears only in this cluster: a positive correlation with unaligned memory accesses. The second scatter plot gives an indication of why this may be misleading, however. It appears that the jobs are either exclusively well aligned in memory or heavily unaligned in memory, and all of the larger jobs (and thus higher-performing jobs) fall into the latter category, producing a misleading correlation with memory alignment. Memory alignment is indeed a performance factor, but it usually is not prominent on an HPC system because of the proportionally larger impact of file alignment due to secondary storage latency. However, this cluster is a good example of

why this exploratory analysis should be interactive. At a finer granularity, these unaligned and well-aligned jobs may be split into two distinct clusters, with more interpretable models trained on each of them.

This cluster includes data science/ML, chemistry, and thermodynamics applications, as well as I/O benchmarks. Further analysis is required to ascertain why this cluster does not conform to expectations for I/O performance correlations. One possibility is that it accessed a different storage system (for example, possibly relying to a large degree on small local solid-state storage devices present on each node of Theta), a factor that is not captured by Gauge analysis. A substantially different storage system, particularly one that does not provide a shared namespace that can cause contention, is likely to have significantly different performance properties. The unusual clustering in this case may be an indicator of emerging performance phenomena warranting attention that may not be apparent from anecdotal user interactions.

C. Cluster Delta

Before we analyze cluster Gamma, we first look at its sub-cluster, cluster Delta. Delta includes 11,961 jobs. It is a notable cluster because all samples within it were produced by just three applications. Those three applications belong to the same scientific domain: climate modeling. It is not surprising that distinct applications in a given science domain may share similar datasets, data formats, and data access methods, but such a grouping might not be readily evident to a facility operator. Even if the grouping were evident at an administrative level, there is no guarantee that applications in a given domain will be so closely related. This cluster is an example of how Gauge can be used to identify related applications that may benefit from a common optimization.

The applications in this cluster are characterized by the majority of files being either pure read-only or pure write-only (mixed read/write workloads to the same file are rare). The overall read/write ratio varies considerably from job to job, but more jobs are read-heavy than write-heavy overall.

The SHAP analysis of correlations between metrics and performance indicates factors that mostly conform to conventional I/O tuning wisdom. Jobs that read data in relatively large chunks (1 MiB to 4 MiB, i.e., multiples of the parallel file system block size) have a positive correlation with performance, while jobs that access data in small chunks (100 bytes to 1 KiB) are negatively correlated with performance. This relationship is confirmed for read accesses in all three scatter plots, which report higher performance intensity as the number of read operations in the 1–4 MiB range increases but are less sensitive to changes in the number of writes in the 100–1000 KiB range, reads in the 1–10 KiB range, and reads in the 100–1000 KiB range.

D. Cluster Gamma

Cluster Gamma is notable as a contrast to cluster Delta. By composition, it includes the same three climate simulation codes but adds a fourth additional climate simulation code as well as a plasma simulation (i.e., an application from an unrelated scientific field). This is notable for two reasons. First, it illustrates the potential for common optimization strategies to cross scientific domain boundaries if applications (either by coincidence or by design) share similar properties. Second, despite the intersection of application composition from cluster Delta, the SHAP analysis illustrates a distinct performance correlation that was not present in the latter cluster. This suggests that potential optimizations are not necessarily applicable to all instances of an application but rather to particular *configurations* of an application that may place it in different workload clusters.

The SHAP analysis for cluster Gamma shows that the dominant correlation in this case is a negative correlation between the percentage of unaligned (in file) accesses and application performance. This factor is known to hinder performance in some use cases [19], particularly when accessing shared files, and is most often addressed by refactoring I/O access loops in the application code or by applying high-level library optimizations to restructure access patterns before they reach the file system.

The distinction in which factor is most critical to performance in clusters Delta and Gamma is due to an underlying factor that can be observed in the slope of the lines connecting the last two axes in the top row's parallel coordinates plot (column 3, row 1). A downward slope (cluster Gamma) indicates the presence of shared files (more processes than there are files), which would be more sensitive to alignment because of the potential for false sharing between adjacent processes accessing a common file. An upward slope (cluster Delta) indicates a file-per-process workload (more files than there are processes), in which there is unlikely to be false sharing between two processes in a file and instead the key optimization at this scale is the ratio of productive work per latency intensive operation (i.e., access size).

VI. Related Work

The I/O performance bottleneck in large-scale HPC systems has been the subject of several studies. In [20] and [21], the authors presented some of the earlier attempts to provide a comprehensive description of I/O pattern characteristics based on their evaluation of the performance of a file system while running real application workloads. Other studies took a monitoring software approach, including TAU [22], Paraver [23], SCALASCA [24], Paradyn [25], and Darshan [26]. These software approaches generally consist of capturing I/O access events from HPC application runs, which, when analyzed and coupled with domain expert insight, could result in better system characterization, more targeted resource allocation, and ultimately improved application or system performance.

Because of the complexity of the parallel file systems and the applications run on them, ML methods have been increasingly adopted to extract insights and model I/O performance. Supervised ML approaches involve learning a mapping that can predict quantities of interest such as runtime or I/O performance. Several works have adopted supervised ML for predicting I/O performance. These include the work of Kim

et al. [27], who modeled key characteristics of HPC systems, such as bandwidth distribution, the ratio of request size to performance, and idle time. Unfortunately, there is no evidence showing these characteristics correlate to application behavior. Dorier et al. [1] proposed the Omnisc'IO approach, which builds a grammar-based model of the I/O behavior and is then used to predict when future I/O events will happen. Madireddy et al. [3] proposed a sensitivity-based modeling approach that leverages application and file system parameters to find partitions in I/O performance data from benchmark application jobs and builds Gaussian process regression models for each partition to predict the I/O performance. In their follow-up work [28], the authors adopted a neural-network-based approach to build global models to predict I/O performance while considering the application and file-system parameters as categorical variables. McKenna et al. [29] manually extracted features from job logs and used several ML methods to predict HPC job runtime and I/O behavior. Rodrigues et al. [30] used features extracted from log files and batch scheduler logs and presented an ML-based tool that integrates several methods to predict resource allocation for HPC environments. Matsunaga et al. [31] presented a Predicting Query Runtime Regression (PQR2) algorithm that was used on selected features from a dataset generated by running bioinformatics applications to predict execution time, memory, and disk requirements. In a more recent study, Li et al. [4] presented PIPULS, a long short-term memory neural network implementation coupled with a hardware prototype used for online prediction of future I/O patterns, for applications such as flash memory solid-state drives scheduling and garbage collection. These approaches are designed primarily for predicting the I/O performances for specific scenarios but provide limited insights into the similarities in I/O characteristics across applications and I/O bottlenecks.

Unsupervised ML approaches have been used as well to uncover features or patterns from a dataset, instead of predicting I/O throughput. This approach was explored by [5] to discover I/O behaviors, where k-means clustering was used to group similar jobs, and identified nine access patterns representing 72% of the I/O behavior. Liu et al. [32] proposed AID (Automatic I/O Diverter), a tool for automatic I/O characterization and I/O scheduling. Deployed on the Titan supercomputer, AID identified I/O-heavy applications and reduced their I/O contention through better scheduling. Our approach uses a combination of unsupervised and supervised techniques to provide greater model stability and robustness.

VII. Future Work

Our future work will explore three directions:

Investigating external impacts on I/O performance: in this work, we focus primarily on internal, job-specific impacts on I/O throughput and have ignored external impacts (e.g., I/O contention). To explore external impacts, we plan to integrate logs from systems that measure I/O contention, as well as include jobs smaller than 100 MiB in our analysis, which may allow us to better estimate I/O contention.

Improving model generalization and explanations: we plan to explore how we can further preprocess Darshan logs, as well as to investigate better models of I/O, both in order to increase I/O throughput prediction accuracy and to arrive at models that better encapsulate I/O behavior. With better explainability, we hope to reduce the reliance on manual analysis, allowing broader usage of Gauge.

Exploring sources of errors: there are four possible reasons why our models make errors: either the system or distribution of applications may have changed, I/O contention may affect jobs, or our models require further tuning. We hope to arrive at a better understanding of how much each of these components contributes to prediction error.

VIII. Conclusion

Writing and tuning high-performing HPC applications requires significant domain expertise, as well as a good understanding of the HPC system the application will be running on. Often, developers and system owners find that despite their best effort, their HPC jobs utilize only a fraction of the promised I/O throughput. In this work, we tackled the problem of modeling application I/O performance in order to extract insight into why applications are behaving as they do, and generalize this insight to larger groups of jobs. While domain experts can analyze individual logs and give feedback on the causes of poor I/O, such an approach is not scalable. In this work we set out to build tools for automated, unsupervised grouping of jobs with similar I/O behaviors, as well as methods for analyzing these groups. We present Gauge: an exploratory, interactive tool for clustering jobs into a hierarchy and analyzing these groups of jobs at different granularities. Gauge consists of two parts: a clustering hierarchy of HPC jobs and a dashboard allowing a system owner or developer to gain insight into these clusters. We build this hierarchy using 89,884 Darshan logs from the Argonne Leadership Computing Facility (ALCF) Theta supercomputer, collected in the period between 2017 and 2020. Using this tool and data, we ran a case study from the perspective of a Theta's system administrator, showing how Gauge can detect families of applications and spot strange I/O behavior that may require further fine-tuning and optimization of these applications, hardware provisioning, or further investigation. Gauge provides novel information that leads to new insights, but it still requires guidance from a domain expert. Future work lies in reducing this reliance so that the techniques from Gauge can lead to more agile improvements in the field.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 623—634.

[2] 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 *26th International Symposium on High-Performance Parallel and Distributed Computing*. New York: ACM, 2017, pp. 181–192.

[3] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. M. Wild, "Machine learning based parallel I/O predictive modeling: A case study on Lustre file systems," in *High Performance Computing*. Springer, June 2018, pp. 184–204.

[4] D. Li, Y. Wang, B. Xu, W. Li, W. Li, L. Yu, and Q. Yang, "PIPULS: Predicting I/O patterns using LSTM in storage systems," in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE, 2019, pp. 14–21.

[5] P. J. Pavan, J. L. Bez, M. S. Serpa, F. Z. Boito, and P. O. A. Navaux, "An unsupervised learning approach for I/O behavior characterization," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2019, pp. 33–40.

[6] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O Characterization with Darshan," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, 2016, pp. 9–17.

[7] *Darshan-util installation and usage*, 2020 (accessed April 22, 2020), https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html#_darshan_parser.

[8] *Gauge supporting experiments*, 2020 (accessed June 5, 2020), https://anonymous.4open.science/r/1c9a3777-0133-4db8-bff8-deffed0cad95/.

[9] 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 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 33—44.

[10] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering," *The Journal of Open Source Software*, vol. 2, no. 11, mar 2017.

[11] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[12] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns, "UMAMI: A recipe for generating meaningful metrics through holistic i/o performance analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, ser. PDSW-DISCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 55–60.

[13] A. Altmann, L. Toloşi, O. Sander, and T. Lengauer, "Permutation importance: a corrected feature importance measure," *Bioinformatics*, vol. 26, no. 10, pp. 1340–1347, 04 2010.

[14] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774.

[15] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee, "From local explanations to global understanding with explainable AI for trees," *Nature Machine Intelligence*, vol. 2, no. 1, pp. 2522–5839, 2020.

[16] R. Latham, C. Daley, W. keng Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary, "A case study for scientific I/O: improving the FLASH astrophysics code," *Computational Science & Discovery*, vol. 5, no. 1, p. 015001, mar 2012.

[17] J. Kodavasal, K. Harms, P. Srivastava, S. Som, S. Quan, K. Richards, and M. García, "Development of a Stiffness-Based Chemistry Load Balancing Scheme, and Optimization of Input/Output and Communication, to Enable Massively Parallel High-Fidelity Internal Combustion Engine Simulations," *Journal of Energy Resources Technology*, vol. 138, no. 5, 02 2016, 052203.

[18] A. Srinivasan, C. D. Sudheer, and S. Namilae, "Optimizing massively parallel simulations of infection spread through air-travel for policy analysis," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 136–145.

[19] X. Zhang, K. Liu, K. Davis, and S. Jiang, "ibridge: Improving unaligned parallel file access with solid-state drives," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 381–392.

[20] A. N. Reddy and P. Banerjee, "A study of I/O behavior of perfect benchmarks on a multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 312–321, 1990.

[21] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/output characteristics of scalable parallel applications," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995, pp. 59–es.

[22] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[23] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1. Citeseer, 1995, pp. 17–31.

[24] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[25] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[26] 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*. IEEE, 2009, pp. 1–10.

[27] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer, "Workload characterization of a leadership class storage cluster," in *2010 5th Petascale Data Storage Workshop (PDSW'10)*. IEEE, 2010, pp. 1–5.

[28] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild, "Modeling I/O performance variability using conditional variational autoencoders," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 109–113.

[29] R. McKenna, S. Herbein, A. Moody, T. Gamblin, and M. Taufer, "Machine learning predictions of runtime and IO traffic on high-end clusters," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 255–258.

[30] E. R. Rodrigues, R. L. Cunha, M. A. Netto, and M. Spriggs, "Helping HPC users specify job memory requirements via machine learning," in *2016 Third International Workshop on HPC User Support Tools (HUST)*. IEEE, 2016, pp. 6–13.

[31] A. Matsunaga and J. A. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 495–504.

[32] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 819–829.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran 6 experiments, and each of the experiments corresponds to a figure in our paper: 1. some simple statistical plots that showed the distribution of our data, 2. we ran permutation feature importance experiments to analyze XGBoost models and how they attribute importance, 3. we ran hierarchical clustering on our dataset and have trained predictors on individual clusters as well as the whole dataset, 4. we have ran hierarchical clustering visualizations, 5. we have plotted distance matrices of points in our dataset, and 6. we have a script that shows our tool (Gauge). Gauge runs a lot of small plots, as well as training XGBoost models, and running SHAP analysis on them (https://github.com/slundberg/shap).

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://anonymous.4open.science/r/1c9
↪ a3777-0133-4db8-bff8-deffed0cad95/
Artifact name: Code supporting the submission
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Applications and versions: Python3.6

Libraries and versions: cycler==0.10.0 Cython==0.29.16 decorator==4.4.2 graphviz==0.13.2 hdbscan==0.8.26 joblib==0.14.1 kiwisolver==1.2.0 matplotlib==3.1.2 networkx==2.4 numpy==1.18.1 pandas==1.0.0 pydot==1.4.1 pyparsing==2.4.7 python-dateutil==2.8.1 pytz==2019.3 scikit-learn==0.22.1 scipy==1.4.1 seaborn==0.10.0 shap==0.35.0 six==1.14.0 sklearn==0.0 tqdm==4.45.0 xgboost==1.0.2

Key algorithms: HDBSCAN, SHAP, gradient boosting machines

ARTIFACT EVALUATION

Verification and validation studies: For the several machine learning models we have trained, we have always used 70-30 ratio cross-validation.

Accuracy and precision of timings: Our work does not perform any timings.

Used manufactured solutions or spectral properties: /

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: Our experiments are (for the most part) deterministic. We might have not hardcoded seeds in all of the experiments, but the only place where results could change is in the ML models, and all of the ML models we use (XGBoost for the most part), are robust to initial conditions.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. Multiple of our experiments use box plots to provide an estimate of classifier error.