Machine Learning Assisted HPC Workload Trace Generation for Leadership Scale Storage Systems

Arnab K. Paul*†
arnabp@goa.bits-pilani.ac.in
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
BITS Pilani, K K Birla Goa Campus
Zuarinagar, Goa, India

Jong Youl Choi*
choij@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Ahmad Maroof Karimi
karimiahmad@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Feiyi Wang
fwang2@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

ABSTRACT

Monitoring and analyzing a wide range of I/O activities in an HPC cluster is important in maintaining mission-critical performance in a large-scale, multi-user, parallel storage system. Center-wide I/O traces can provide high-level information and fine-grained activities per application or per user running in the system. Studying such large-scale traces can provide helpful insights into the system. It can be used to develop predictive methods for making predictive decisions, adjusting scheduling policies, or providing decisions for the design of next-generation systems. However, sharing real-world I/O traces to expedite such research efforts leaves a few concerns; i) the cost of sharing the large traces is expensive due to this large size, and ii) privacy concern is an issue.

We address such issues by building an end-to-end machine learning (ML) workflow that can generate I/O traces for large-scale HPC applications. We leverage ML based feature selection and generative models for I/O trace generation. The generative models are trained on I/O traces collected by the darshan I/O characterization tool over a period of one year. We present a two-step generation process consisting of two deep-learning models, called the feature generator and the trace generator. The combination of two-step generative models provides robustness by reducing the bias of the model and accounting for the stochastic nature of the I/O traces across different runs of an application. We evaluate the performance of the generative models and show that the two-step model can generate time-series I/O traces with less than 20% root mean square error.

*Both the authors contributed equally to the paper.
†The majority of the work was done by the author during his postdoctoral research at Oak Ridge National Laboratory, USA. He is presently working as an assistant professor at BITS Pilani, Goa, India.

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06…$15.00
https://doi.org/10.1145/3502181.3531457

CCS CONCEPTS

• **Information systems → Distributed storage**; • **Computing methodologies → Machine learning approaches**; • **Software and its engineering → File systems management**.

KEYWORDS

Darshan, Generative Modeling, Feature Selection, Clustering, Parallel File System

ACM Reference Format:
Arnab K. Paul, Jong Youl Choi, Ahmad Maroof Karimi, and Feiyi Wang. 2022. Machine Learning Assisted HPC Workload Trace Generation for Leadership Scale Storage Systems . In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3502181.3531457

1 INTRODUCTION

High Performance Computing (HPC) is evolving from traditional simulation bulk-synchronous workloads to big data based artificial intelligence (AI), data analytics, deep learning, and complex multi-step workflows. The current trend for HPC systems is that processor performance improves at a rate of 20% per year, while disk access time improves by only 10% every year [21]. Therefore, much effort is being dedicated to implement and optimize high performance parallel file systems to support the increasing I/O needs of HPC applications.

1.1 Motivation

I/O optimization techniques frequently originate from many academic institutions and research laboratories that do not have access to extremely large-scale HPC systems. While, it is possible for such institutions to test their technique on a smaller (< 100 nodes) cluster; to ensure that the evaluation is relevant, scalable and accurate, these optimizations to storage systems must also be studied and evaluated in the context of leadership scale HPC systems. Leadership scale systems have thousands of nodes and cover a huge range of supercomputers, for example, Fugaku in Japan [3], Summit at Oak Ridge National Laboratory (ORNL), USA [6], and Perlmutter[5] at National Energy Research Scientific Computing Center (NERSC)

in Lawrence Berkeley National Laboratory, USA. These machines are typically procured with tax-payer's money, and machine time is allocated through open calls, granted through a peer-review process. These are not as open as cloud platforms, but are classified as "open science" machines, with about hundreds of big science projects running every year. To this end, researchers use synthetic I/O traces which are fed to trace replay tools [29, 53] for simulation of large scale workloads [30, 32]. However, real-world large-scale I/O traces are not readily available due to several constraints, such as security, cost of sharing, and limitation in accessing data. Therefore, I/O workload trace generation from real-world large-scale HPC systems is a critical but frequently overlooked aspect of storage system simulation and evaluation.

The first step to I/O workload trace generation is to accurately capture I/O workloads from leadership scale HPC systems. Various tracing tools [26, 49], and server side instrumentation methods, such as Lustre Monitoring Tool (LMT) [4] are deployed to capture the workload and storage server statistics. On most leadership scale HPC systems around the world, to capture an accurate picture of application I/O behavior, including properties such as patterns of access within files, with minimum overhead, darshan [1] - one of the most popular HPC I/O characterization tools, is deployed. The publicly available darshan traces [2] are mostly the overall or high-level summary statistics of completed jobs which are collected using the darshan-total API. While, there has been a plethora of research analyzing these logs to study I/O behavior of large-scale supercomputers [14, 36, 37], such logs are not useful for generating I/O traces. To generate I/O traces from real-world darshan logs, finer-grained logs need to be used which contain time-stamps of the I/O performed during an application runtime. However, these fine-grained logs are not publicly available as they are very large in size as well as extremely difficult to anonymize, because of being proprietary and export-controlled. Therefore there is a need of an I/O workload generation technique that generates anonymized I/O traces which is space-efficient and accurately mimics the large-scale production HPC workloads.

1.2 Limitations of State-of-Art Approaches

Trace generation for the application analysis is an area of research in which the tools or algorithms are developed to simulate real-world applications' behavior. Mueller and Pakin [31] provide one of the first techniques to automatically generate performance accurate benchmarks from parallel applications. Behzad et al. [10] show how to create a compressed I/O kernel automatically, by executing the target application with an instrumented I/O library, next compressing the resulting I/O traces into a compact C program that generates those traces. However, both these works are either application-specific or require data collected from I/O workloads during runtime. Snyder et al. [46] use Darshan logs for I/O workload generation for the first time. Though the approach is useful to generate traces for one run of an application, it fails to capture the variability in the I/O behavior of an application when it runs for multiple times. There is also an additional effort required to anonymize the I/O traces generated from such an approach so that the trace can be publicly shared. The machine learning (ML)-based approach shown in this paper aims to alleviate these limitations.

Recently, Costa et al. [14] show the I/O performance variability across different jobs of the same application running on the same scale. This is majorly due to I/O contention at the cluster level, which in turn leads to a change in the I/O traces. However, there has been no work that use the I/O performance variability as a factor in I/O trace generation to generate different traces for the same application. We aim to use this factor and present an end-to-end ML workflow to generate I/O traces from leadership scale HPC storage systems. To this end, we apply generative ML model on fine-grained darshan logs collected on Summit– one of the fastest supercomputers in the world according to the latest Top500 list [7], which is housed in ORNL, USA.

Various ML methods have been used previously on darshan logs. Madireddy et al. [27] use conditional variational auto encoders (CVAE) for I/O performance modeling, and to quantify performance variability. Bang et al. [8] use ML techniques on darshan logs for HPC workload characterization. Isakov et al. [22] use local methods, like clustering and feature selection on darshan logs to analyze I/O throughput bottleneck. However, all of these works focus on the darshan-total logs rather than the fine-grained logs. Also, the ML methods, such as feature selection used in these works do not consider the importance of selecting different features for different scales of application runs. For example, I/O performance of an application might be heavily impacted by small reads on a small scale application run but big reads when running the application on a huge scale. Our work considers fine-grained darshan logs to generate I/O traces by considering the varying impact of the scale of application run as well as the variability in an application run at the same scale.

1.3 Methodology and Contributions

The aim of this work is to be able to generate different I/O patterns for a HPC application given the scale of the application run. To this end, we consider fine-grained darshan logs for one year (2020) on Summit. The fine-grained darshan logs maintains details of every file that is opened by the application, for example, the first and last timestamps of POSIX read/write. Since this work focuses on generating traces for leadership scale systems, we only consider applications which run on more than 92 nodes. We further filter and choose only those applications which contribute to more than 90% of the I/O load (cumulative read and write bytes) in a particular scale of Summit run. After filtering the applications, the fine-grained darshan logs for the applications are converted to time-series based I/O workload using the I/O workload modeling approach described in [46].

Darshan logs typically have more than 200 features which warrant feature engineering and selection. We use mutual information scores [18] to rank features. To select the best features in each scale of application run, we use clustering techniques that provide the optimal clusters for applications in a given scale. These selected features are then used to train a generative model that can reproduce ORNL I/O patterns. Building flexible yet robust generative models is challenging. Using untrained input features can easily make the generative model output noisy or nonsensical. We, therefore, propose a data-driven, ML-based method to overcome such issues by separating processes in preparing features and generating

traces. Our proposed workflow uses two generative models, called the Feature Generator (FG) and the Trace Generator (TG). First, the FG will generate a set of features (called as *feature seed*) to be used as an input to the TG. This step ensures anonymization of the I/O traces as the only input to the model is the application name and the scale. The same application will generate different feature seed values for different scales of the application run. The feature seed values also differ for the same scale within a threshold to account for the I/O performance variability of different jobs of the same application running at the same scale. Taking the feature seed as input, the TG will then generate I/O trace data.

Specifically, our work makes the following contributions.

- Design and evaluate an *end-to-end ML-based workflow to generate anonymized I/O traces* from leadership-scale HPC systems. The workflow considers the different scale runs of the same application as well as I/O performance variability of an application running on the same scale at varying periods of time.
- Engineer and identify *important features for I/O workload trace generation* for different scales of application runs on leadership scale HPC systems. This helps in compressing the fine-grained darshan logs that makes the model more space-efficient.
- Build a two-step generative model to *generate time-series write bytes, read bytes and number of files for different applications at different scales*. Our experiments indicate that the accuracy of our approach is more than 80% when compared with the baseline darshan based trace generation approach inspired from Snyder et al. [46].

1.4 Limitations of the Proposed Approach

The primary focus of this work is to design and validate an end-to-end ML workflow to generate I/O traces from fine-grained darshan logs of leadership-scale HPC systems. Therefore, we only consider POSIX I/O traces. I/O trace generation for different interfaces, like, MPI-IO and STDIO, specifically how to model the I/O traces of collective and independent MPI-IO calls will be done as part of future work. Also, while the methodology is generic and can be used for I/O trace generation of any HPC cluster, our generative model needs to be trained with darshan logs from different HPC facilities to make it truly universal.

2 BACKGROUND

This section briefly describes the Summit supercomputer, darshan tool and common deep generative modeling techniques.

2.1 Summit Supercomputer

The Summit supercomputer is based upon IBM AC922 system and deployed at ORNL. It consists of 4,608 compute nodes. Each node is equipped with 2 IBM POWER9 (P9) processors and 6 NVIDIA Tesla V100 (Volta) GPUs. Also, each node has 512 GB of DDR4 CPU memory and each GPU has 16 GB of HBM2 memory. An NVLink 2.0 bus connects each P9 CPU to 3 V100 GPUs. An InfiniBand EDR network with a fat-tree topology connects the nodes. A 1.6 TB NVMe device is present on each compute node to be used as node-local storage. Summit is connected to Alpine, a 250 PB IBM Spectrum Scale (GPFS) file system. Summit can access Alpine at 2.5 TB/s in aggregate under a large, sequential write I/O access pattern. Alpine is a center-wide file system and directly accessed by all other ORNL resources.

2.2 Darshan - HPC I/O Characterization Tool

Figure 1 provides an overview of the darshan architecture. As an application executes, the darshan instrumentation module for MPI, POSIX and STDIO generates data records characterizing the application's I/O workload within different components of the I/O stack. The instrumentation modules for the various components are registered with the darshan core library. During application shutdown, each module organizes its records, compresses it and writes those collectively to the log file.

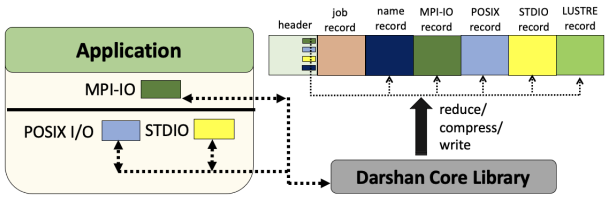


Figure 1: High-level overview of darshan's architecture.

POSIX module records each $read()$ and $write()$ call. MPI-IO is recorded for every $MPI\_File\_read()$ and $MPI\_File\_write()$ calls. Many applications rely on text-based I/O in leadership class computing facilities. Therefore, the STDIO module characterizes the *stdio.h* family of functions, such as $fopen()$, $fprintf()$, and $fscanf()$.

Darshan provides $darshan-util$, which has a collection of tools for parsing and summarizing log files produced by darshan instrumentation. For our feature selection process, we use the `-total` option in $darshan-parser$ to get all statistics as an aggregate total where each field is either summed across files and processes or zeroed out (for nonsensical values). We use $darshan-parser$ to parse the darshan logs without any option to get I/O statistics for every file in each module accessed by the application. These fine-grained logs are used in the model for generating I/O traces.

2.3 Deep Generative Modeling

Building a generative model is one of the most popular ML methods. It is a technique to discover the generative processes of a given data and to enable the regeneration of new data sets that are closely similar to the original data. In statistics, finding probabilistic models of the observed data and fitting parameters for the model have been researched based on the Bayesian rules, and conditional probabilities.

Recently, deep learning methods for building generative models, called deep generative modeling, have gathered great attention. Many research works and papers demonstrate the possibility and success in many scientific applications, such as image generation [33], physics simulation learning [42], and surrogate modeling [54]. Most deep generative models currently available can be categorized into three types of algorithms; variational autoencoders (VAEs) [24], generative adversarial networks (GANs) [19], and normalizing flows [39]. VAEs construct a generative process using latent variables through an encoding-decoding process. GANs build a

generating function by incorporating feedback from a discriminator. Normalizing flows find a probabilistic generative process based on a series of invertible transformations.

3 DATA PREPROCESSING

As discussed in Section 2.2, *darshan − parser* is used to parse the raw darshan logs of one year (January to December 2020) on Summit and get the file-wise I/O statistics in each module (MPI-IO, POSIX, or STDIO) accessed by the application. The metadata of each parsed darshan log consists of *jobid*, *userid*, *start_time*, *end_time*, *executable*, *no_of_processes*, and *runtime*. Summit's scheduler logs are merged with the darshan logs to get the science domain (Physics, Chemistry, Computer Science, or others) and the number of nodes on which the HPC jobs ran. A total of 845, 036 jobs ran on Summit in 2020. Out of this, darshan logs were generated for 279, 642 jobs.

3.1 Classification Based on Scale of Jobs

Job Scale	Min Nodes	Max Nodes	Max Walltime
Scale-1	2,765	4,608	24 hrs
Scale-2	922	2,764	24 hrs
Scale-3	92	921	12 hrs
Scale-4	46	91	6 hrs
Scale-5	1	45	2 hrs

Table 1: Scheduling policy in Summit. In this work, scales 1–3 are considered.

Table 1 shows the scheduling policy in Summit. All darshan records are therefore classified into the five bins (Scales 1 − 5). It should be noted that Summit encourages large-scale production runs. Therefore, applications which run on more than 921 nodes have the maximum walltime of 24 hours. This provides a unique opportunity to work on I/O trace generation for large-scale production workloads which cannot be easily accessed by academicians. Therefore, in this paper, we focus on jobs running on more than 92 nodes by only considering scales 1, 2, and 3. The number of darshan records for one year in each of these three scales are listed below.

- *Scale-1* − 251
- *Scale-2* − 97, 075
- *Scale-3* − 698, 837

3.2 Selecting Applications from Each Scale

For each scale described above, we select x applications, where x is the number of applications that cumulatively consume at least 90% of the I/O load (total read and write bytes) in a particular scale for the entire year. The number of applications consuming 90% of I/O load in every scale is shown in Figure 2. For Scale-1, only 3 applications and in Scale-2, only 6 applications consume more than 90% of the total I/O load. Therefore, to get a considerable number of darshan logs to train the machine learning model, we consider the top 10 applications for scales 1 and 2, and 12 applications for scale-3.

After the filtering of appropriate darshan logs and applications, each darshan record is converted from a file-wise I/O characterization to a time-series based I/O trace.

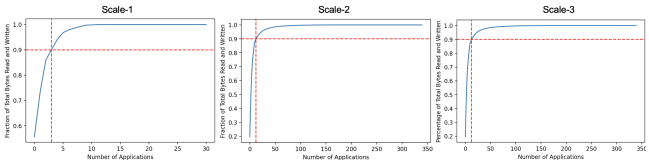


Figure 2: The cumulative density function (CDF) plots for the number of applications and I/O load in every job scale. The number of applications that consume at least 90% of I/O load in scales 1, 2, and 3 are 3, 6, and 12 respectively.

3.3 Darshan Trace Formation

The Darshan I/O characterization tool maintains details of every file that is opened by an application. The I/O interfaces recorded are POSIX, MPI-IO, STDIO, HDF5, and PnetCDF used to access the file. The number of files using POSIX, MPI-IO and STDIO interfaces by all jobs running on Summit for the year 2020 is shown in Table 2. It is seen that POSIX is the chosen I/O interface by more than 50% of the files used by applications running on Summit. Therefore, in this paper, we focus on only POSIX I/O.

POSIX (Millions)	**MPI-IO** (Millions)	**STDIO** (Millions)
795	157	631

Table 2: Number of files (in millions) using the STDIO, MPIIO, and POSIX I/O interfaces on Summit.

The major counters collected by Darshan for every file at the POSIX interface are:

- Timestamps the first file open/read/write/close operation began - `POSIX_F_*_START_TIMESTAMP`
- Timestamps the last file open/read/write/close operation ended - `POSIX_F_*_END_TIMESTAMP`
- Cumulative time spent on reading from the file - `POSIX_F_READ_TIME`
- Cumulative time spent on writing to the file - `POSIX_F_WRITE_TIME`
- Total number of bytes that were read from the file - `POSIX_BYTES_READ`
- Total number of bytes written to the file - `POSIX_BYTES_WRITTEN`
- Rank which accessed the file - `rank`

Accurately building time-series I/O traces using darshan logs poses a significant challenge. The compact format used by darshan to characterize application I/O behavior omits several details that would be helpful in reconstructing the original workload. For example, darshan records the time span in which I/O occurs and the number of I/O operations, but it does not indicate precisely how those I/O operations are distributed within that time span. To overcome such limitation, we apply basic heuristics as described in [45] to derive representative workload streams.

Figure 3 shows the general process of transforming darshan I/O logs into comprehensive I/O traces which is in sync with the technique proposed by Snyder et. al [46]. The ranks which access the file denote whether the application was run as a file-per-process or

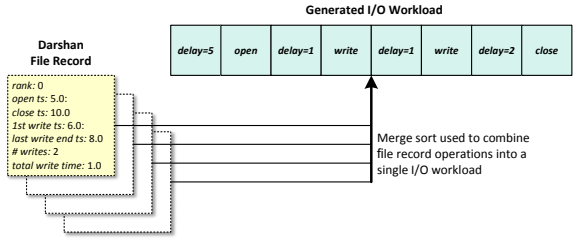


Figure 3: Transforming Darshan file records into an I/O workload [46].

a single-shared file mode. Each file is arranged based on increasing order of first open timestamp. The I/O idle time for every file is calculated by subtracting the cumulative time spend on reading and writing from the duration between the file open start and file close end timestamps. This idle time (or delay) along with the cumulative bytes read or written are uniformly distributed for every file within the file open start timestamp and file close end timestamp. Once the distribution of every file's I/O activity is done, insertion sort is used to sort and combine the I/O activity of the application in a time-series manner. As described in Section 1.2, the darshan traces generated using the approach given by Snyder et al. [46] only gives us traces for one run of an application. Therefore, an ML-based approach is needed which can depict the stochastic I/O behavior of applications. Thus, these darshan traces grouped by all runs of an application in a particular scale are first used as the training dataset for our model. Finally, the anonymized and stochastic traces generated using our model are validated against these baseline darshan traces.

3.4 Dataset Description

The final I/O trace dataset from Summit for scales 1 – 3 used for I/O trace generation after filtering applications and using darshan trace formation technique is described in Table 3. The number of unique users running the selected number of applications on Summit are 11, 14 and 39 for scales 1, 2 and 3 respectively. More than 550, 000 darshan logs are processed and applications belong to diverse set of science domains – Biology, Chemistry, Computer Science, Earth Science, Engineering, Machine Learning, Materials, Nuclear Energy, and Physics. The total read and write bytes on Summit for the year 2020 that is modeled from the selected applications are 96 PB and 57 PB respectively. The 32 applications which are finally considered for our approach are representative of the I/O load on Summit as they consume more than 90% I/O activity for all the three scales. Also, among the chosen applications, there are applications which run on only one scale as well as applications that scale up from scales 1 to 3. Therefore, we are able to validate our approach for both classes of applications.

4 FEATURE SELECTION

As discussed in previous sections, every darshan log has a header module which contains features regarding the executable, userid, jobid, start time, end time, number of processes, and job runtime. Each interface that a file uses has a huge number of features describing the I/O behavior of the file specific to the interface. For feature selection, we focus on only POSIX, MPI-IO and STDIO interfaces. Therefore, each darshan record contains a total of ∼300 features. In this section, we first explore which features contribute the most to the I/O behavior of an application and if the scale of an application run plays an important role in determining the important features.

4.1 Feature Engineering

Darshan features specific to a particular run of an application are removed as those do not contribute to the general I/O behavior of the entire application. The features which are removed are:

- start time, end time
- userid, jobid
- POSIX_START/END_TIMESTAMP
- MPIIO_START/END_TIMESTAMP
- STDIO_START/END_TIMESTAMP

In addition to the remaining features which darshan provides, we also engineer the following features which are already established as important features for I/O characterization for HPC applications.

- *POSIX/MPIIO/STDIO Read Write Ratio* – Bytes read divided by bytes written – An indication of the read or write intensive nature of an application.
- *Node hours* – An important metric in all HPC applications.
- *Fraction of time job spends on POSIX/STDIO/MPIIO read/write/metadata* – Read or Write or Metadata Time divided by application runtime – An indication of the fraction of time spent in different I/O activities.
- *POSIX/STDIO IO Rate* – Gives the I/O performance of the application when using a particular I/O interface.

Table 4 describes the numeric features at the POSIX level. At the MPI-IO level, along with similar features at the POSIX level, there are also features for non-collective and collective MPI-IO operations. The records for the STDIO interface in darshan logs have a subset of features shown for the POSIX layer in Table 4.

Once the feature set is prepared, we apply a feature scaling method to make the scale of the values of all features similar to help steady the convergence of ML models. The two main techniques of feature scaling are normalization and standardization methods. Since standardization scaling is generally used for the dataset having normal distribution, we apply min-max normalization. The process of min-max normalization of data is shown in Equation 1. By doing so, all features will be transformed into the range $[0, 1]$ such that the minimum and maximum value of a feature is going to be 0 and 1, respectively.

$$x_{normalized} = \frac{x - min(x)}{max(x) - min(x)} \tag{1}$$

4.2 Feature Ranking

After features are removed and engineered, the next step is to rank all the features from our dataset. The input variables are all the features in the darshan logs. The out variable is the the science domain of the application. As both input and output are categorical variables, the only option for feature ranking is either mutual information or chi-squared method.

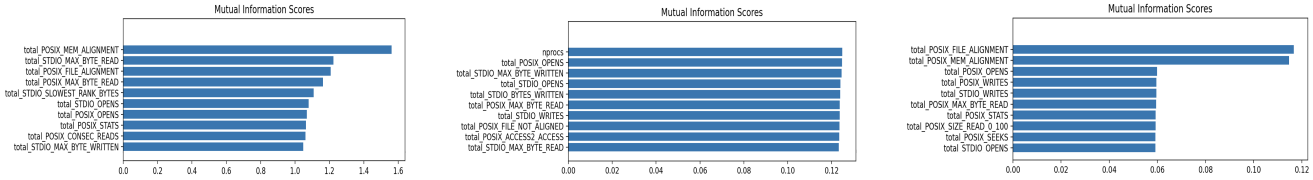
Mutual information ranks all the features in a dataset and uses information gain to calculate the reduction in entropy, by evaluating

Scale	#Apps	#Users	#Darshan Logs	Science Domains	Application Names	Total Read Bytes (TB)	Total Write Bytes (TB)
Scale-1	10	11	36	Chemistry, Computer Science, Engineering, Materials, Nuclear Energy, Physics	'lalibe', 'xgc-es-cpp-gpu', 'hacc_p3m', 'cholla.paris-cuda', 'lbpm_random_force_simulator', 'PPP_Hierarchical.mpi.3d', 'tusas', 'nekrs', 'sigma.cplx.x', 'ramses3d'	1,295	416
Scale-2	10	14	85,078	Biology, Computer Science, Chemistry, Earth Science, Engineering, Physics	'xspecfem3D', 'pmemd.cuda.MPI', 'hf_summit_nvblas.x', 'python', 'hf_summit_oblate.x', 's3d.x' 'prog_ccm_ex_summit_nat.exe' 'xgc-es-cpp-gpu', 'lalibe'	24,805	19,465
Scale-3	12	39	474,132	Biology, Computer Science, Chemistry, Earth Science, Engineering, Machine Learning, Materials, Physics	'lalibe', 'ngp', 'ks_spectrum_hisq', 'dirac.x', 'epw.x', 'rmg-gpu', 's3d.x', 'converge-3.0.11_summit-test', 'python', 'vpicio_uni_h5.exe', 'xspecfem3D', 'conn-hvp-summit.py'	70,344	37,027

Table 3: Description of the darshan logs and applications selected for I/O trace generation.

Feature Name	Description	Feature Name	Description
POSIX_READS/WRITES/MMAP	Number of POSIX read/write/mmap operations	*POSIX_MAX_READ/WRITE_TIME_SIZE*	Size of the slowest POSIX read operation
POSIX_OPENS/STATS/SEEKS	Number of POSIX open/stat/seek operations	*POSIX_MEM/FILE_ALIGNMENT*	Memory/File alignment value which is chosen at compile time for memory and runtime for file
POSIX_BYTES_READ/WRITTEN	Total number of POSIX bytes that were read from/written to the file	*POSIX_MEM/FILE_NOT_ALIGNED*	Number of times that a read or write was not aligned in memory/file
POSIX_MAX_BYTE_READ/WRITTEN	Highest offset where a file was read/written	*POSIX_SIZE_READ/WRITE_**	Histogram of read/write access sizes.
POSIX_CONSEC_READS/WRITES	Number of reads/writes that were immediately adjacent to the previous access	*POSIX_STRIDE[1-4]_STRIDE/COUNT*	Size/count of 4 most common stride patterns
POSIX_SEQ_READS/WRITES	Number of reads/writes at a higher offset than the previous access	*POSIX_ACCESS[1-4]_ACCESS/COUNT*	4 most common access sizes and the count
POSIX_RW_SWITCHES	Number of times that access toggled between read and write in consecutive operations	*POSIX_FASTEST/SLOWEST_RANK*	The MPI rank with smallest/largest time spent in POSIX I/O
POSIX_F_*_START/END_TIMESTAMP	Timestamp that the first/last POSIX file open/read/write/close operation began/ended	*POSIX_FASTEST/SLOWEST_RANK_BYTES*	Number of bytes transferred by the rank with smallest/largest time spent in POSIX I/O
POSIX_F_READ_TIME	Cumulative time spent reading at POSIX level	*POSIX_F_FASTEST/SLOWEST_RANK_TIME*	The time of the rank which had the smallest/largest amount of time spent in POSIX I/O
POSIX_F_WRITE_TIME	Cumulative time spent in write, fsync, and fdatasync at the POSIX level	*POSIX_F_META_TIME*	Cumulative time spent in open, close, stat, and seek at the POSIX level
POSIX_F_MAX_READ/WRITE_TIME	Duration of the slowest individual POSIX read/write operation	*POSIX_F_VARIANCE_RANK_TIME/BYTES*	The population variance for POSIX I/O time/bytes transferred of all the ranks

Table 4: Description of the features provided in the darshan logs for the POSIX layer.



(a) MI Scores of Scale-1 jobs. **(b) MI Scores of Scale-2 jobs.** **(c) MI Scores of Scale-3 jobs.**

Figure 4: The mutual information (MI) scores for the top 10 features of the three scales of Summit jobs.

the gain of each variable in the context of the output variable. The reduction in entropy is measured by splitting a dataset according to a given value of a random variable. A larger information gain suggests a lower entropy group, and hence less surprise. Lower probability events have more information, higher probability events have less information. Entropy quantifies how much information there is in a random variable, or more specifically its probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy. Mutual information is calculated between two variables and measures the reduction in uncertainty for one variable given a known value of the other variable. The mutual information between two random variables X and Y can be stated formally as shown in Equation 2 where I(X ; Y) is the mutual information for X and Y, H(X) is the entropy for X and H(X | Y) is the conditional entropy for X given Y.

$$I(X;Y) = H(X) - H(X|Y) \tag{2}$$

A chi-square test is used to test the independence of two events. Given the data of two variables, we can get observed count O and expected count E. Chi-Square measures how expected count E and

observed count O deviates each other using Equation 3, where c is the degrees of freedom.

$$\chi_c^2 = \sum \frac{(O_i - E_i)^2}{E_i} \tag{3}$$

In feature selection, we aim to select the features which are highly dependent on the response. When two features are independent, the observed count is close to the expected count, thus has a smaller Chi-Square value. So high Chi-Square value indicates that the hypothesis of independence is incorrect. Therefore, higher the Chi-Square value the feature is more dependent on the response and it can be selected for model training. However, in the case of select k-best features using chi-square, there is a need to specify the number of features to be selected before-hand which is not feasible in our case.

The motivation for using feature selection methods is to identify the features which contribute towards I/O performance of HPC applications. However, feature generation methods used for dimensionality reduction and embedding, such as Principal Component Analysis (PCA), Multi-Dimensional Scaling (MDS), or Isomap combine multiple features and give outputs as abstract values which cannot be deciphered. Therefore, we use mutual information to rank features.

Figure 4 shows the mutual information scores for the top 10 features in the three scales of application runs on Summit. It is evident from the figure that different features form major contributors towards the I/O characterization of applications running in different scales. For example, when considering the top 50 features, Scale-1 jobs are more read-intensive than Scale-2 jobs because POSIX_SIZE_READ_1K_10K has a much higher score in Scale-1 compared to Scale-2 where POSIX_SIZE_WRITE_0_100 is more important.

4.3 Clustering

Not all features contribute majorly towards the I/O characterization for HPC application at a particular scale. Therefore, after all features are ranked for the three scales, clustering method is used for feature selection. We use the light-weight DBSCAN clustering algorithm (Density-Based Spatial Clustering of Applications with Noise) [17]. The DBSCAN algorithm uses two parameters.

- *minPts*: The minimum number of points clustered together for a region to be considered dense.
- *eps* (ε): A distance measure that will be used to locate the points in the neighborhood of any point.

The algorithm proceeds by arbitrarily picking up a point in the dataset (until all points have been visited). If there are at least 'minPts' points within a radius of 'ε' to the point then we consider all these points to be part of the same cluster. The clusters are then expanded by recursively repeating the neighborhood calculation for each neighboring point. DBSCAN therefore groups 'densely grouped' data points into a single cluster. It can identify clusters in large spatial datasets by looking at the local density of the data points. It also does not require the number of clusters to be told beforehand, unlike K-Means. Therefore, using DBSCAN on all the features, we first get the number of clusters. Then we select the subset of features (from mutual information ranked feature set) and apply DBSCAN so that we get the same number of clusters as the one using all the features, along with having 90% similarity in data points (i.e., 90% of the data points should belong to the same cluster as the cluster derived from all the features).

Table 5 shows the number of top-ranked features selected for each scale. Each darshan log represents a data point in the DBSCAN algorithm. Therefore, scales 1, 2 and 3 have 36, 85078, and 474132 data points respectively. The minimum number of top features that give at least 90% similar clusters are 75, 60, and 25 features for scales 1, 2, and 3 respectively. Therefore, the top n-ranked features (n = 75 or 60 or 25) are selected and fed as inputs to the I/O trace generation model described in the next section.

Scale	#Data Points	#Clusters	#Features Selected	Cluster Similarity(%)
Scale-1	36	6	75	91.67
Scale-2	85,078	31	60	96.62
Scale-3	474,132	39	25	93.15

Table 5: Minimum number of features that provide at least 90% similar clusters using DBSCAN algorithm in the three scales of HPC applications.

5 GENERATIVE MODELING

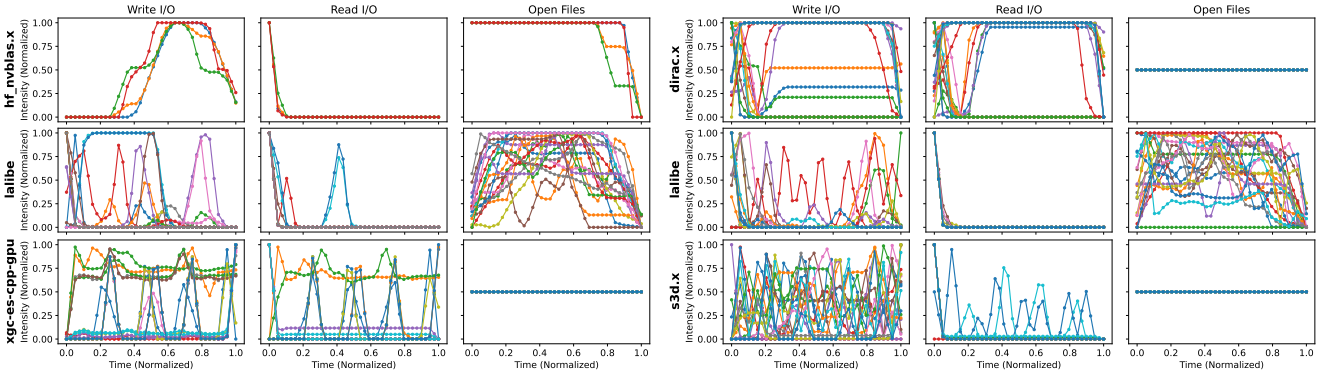
We build a generative model to reproduce darshan traces for multiple purposes; i) efficiently generating realistic-looking traces that can be used for other data-driven simulation environments (e.g., reinforcement learning), ii) saving costs by passing only models, not the full data, and iii) anonymizing traces to protect user privacy. We train a generative model by using our collection of observed data to regenerate new data similar to the observation. Recently, new algorithms and methods have been developed based on artificial neural networks (ANNs) or deep learning (DL) [24, 39]. In this paper, we propose a method using deep learning-based generative models to regenerate I/O traces.

In general, we can define a deep learning-based generative model as follows. We seek to find a neural network model which can take a set of values of size k, $x_i (i = 1, ..., k)$ as an input (also known as features or descriptors) and generate a number of n values of $y_j (j = 1, ..., n)$ as an output:

$$f_{NN} : (x_1, x_2, ..., x_k) \mapsto (y_1, y_2, ..., y_n) \tag{4}$$

where f_{NN} is a neural network or deep learning model. We choose the parameters of the f_{NN} by using a set of observations (known as training) to output values that should be close to the observed data. It is an unsupervised learning process in that we use the observation data for training without any labels. This paper focuses on finding a f_{NN} to generate a time series of I/O patterns based on the real-world, large-scale darshan traces collected in ORNL. Examples of our traces are shown in Figure 5. Our model generates I/O traces for read and write along with the number of open files in a time-series manner. Same application, for example, *lalibe* generates different traces for different scales of application run. Even for the same scale, applications generate different traces (shown by the large number of different line graphs) to exhibit I/O performance variability.

One of the important considerations in designing a generative model is the features of the input. To activate the model f_{NN} to



(a) Scale-2 (b) Scale-3

Figure 5: Example of various I/O patterns (write, read, file open) extracted from the ORNL's Darshan traces. Each line in the plot represent an instance of I/O patterns recorded with darshan. Each application shows a set of repeating patterns which we are trying to regenerate by using our proposed generative models.

generate traces, users will provide input. In generative modeling, the input is also called seed, as it serves as a seed value in developing new data. The seed values should be distinct enough (i.e., no collision) to be expressive to cover a wide range of input features. For such a reason, we use the features identified in Section 4 for the seed values for our generative models. Another aspect of input features to consider is the size. We observe the size of the seed (say k) plays an important role in designing the deep learning model. Using multiple seed values (k>1) improves the quality of the generative deep learning model, compared with a single value (k=1), as it increases the expressiveness of the model [38]. However, there is a limit in the effect for using large seed size. In practice, the size should be reasonable that one can reproduce.

Sensitivity problem with an over-fitted model: We have discussed the right number of feature sizes in the previous section specifically for our darshan traces, which are 75, 60, and 25 for the scales 1, 2, and 3 respectively. However, one caveat is that the number of seed values (input features) is still considerably large for it to be publicly released. Even worse, if the generative model is not well generalized, small deviations in the input feature set can cause unexpected outputs (traces) largely deviated from the original traces. Generally speaking, it is known as the over-fitting problem where the model gets very sensitive (or biased) to the statistical fluctuations in the input data and shows poor prediction or data generation performance with out-of-sample data (also known as unseen or testing data). The possibility of deviation is higher with increasing number of features.

We demonstrate the over-fitting problem with our data set in Figure 6. Our model gets easily over-fitted and shows poor performance in trace generation with random noises. Alternatively, one might consider sharing the inputs (feature dataset) along with the model. However, providing the exact inputs raises privacy concerns by diminishing the anonymization property of the HPC facility. Therefore, there is a need of a two-step generative model.

Two-step model: To overcome the issue of sensitivity or bias in the model, we develop a two-step generative model. Here, a

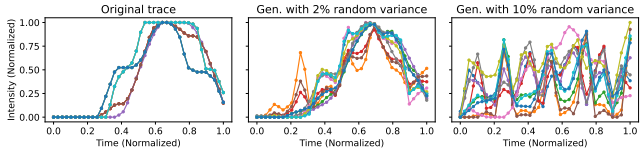


Figure 6: Sensitivity issue with an over-fitted model. We demonstrate how an over-fitted model can generate poor traces due to the small deviations in input. By adding 2% and 10% random noises in the input, the over-fitted model generates very different traces.

generative model (called the feature generator) generates the inputs for the second generator (called the trace generator).

We propose a generative workflow that consists of two generative models connected together (Figure 7). We call the two generative models the Feature Generator (FG) and the Trace Generator (TG), respectively. The FG will generate a set of seed numbers, called feature seed, to be used as an input to the TG. Taking the feature seed as input, the TG will generate trace data. In the following, we describe our proposed generative models.

Feature Generator (FG). It takes a single parameter {application ID and scale} as an input and generates a feature seed to feed to the TG. The main idea behind developing the FG is to help the tracer generator (TG) by providing distinctive and collisionless inputs. For training the FG, we choose a set of 75 or 60 or 25 features for scales 1, 2 or 3 respectively from Summit darshan traces based on the discussion in the previous section. The list of features and sizes vary depending on the job scale.

We envision the feature space as a probabilistic domain, and each application owns a distinct probabilistic distribution without overwrapping other applications. We seek a deep learning model to sample features based on the distribution. We use the normalizing flow method [39] for this type of generator. The normalizing flow is one of the neural network designs constructing complex probability distributions through a series of invertible mappings. At the end of

the training, we obtain a neural network model that can generate a probability distribution we can sample.

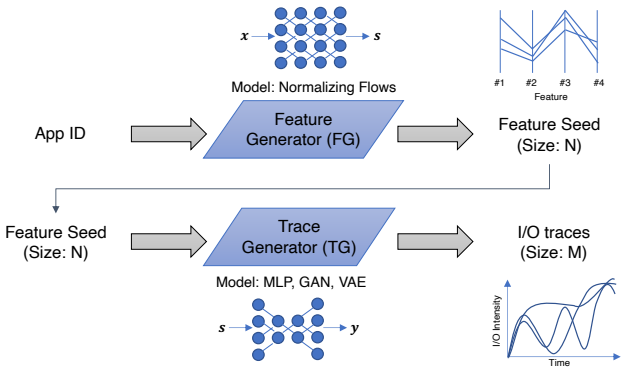


Figure 7: The deep-learning-based HPC workload trace generation workflow. We use two models, called the Feature Generator (FG) and the Trace Generator (TG); The FG generates a feature seed of length N for the TG and the TG, taking them as an input, generates time-series I/O traces of length M. We use the Normalizing Flow model for the FG and multi-layer neural networks for the TG.

Trace generator (TG). It generates a series of M numbers to represent I/O traces. Our goal is to develop a TG model to regenerate similar-looking I/O patterns observed from the real-world ORNL darshan traces we collected. Figure 5 shows a few examples of I/O patterns extracted from the ORNL darshan traces. We focus on generating various types of time-series I/O patterns from an application to demonstrate the changes of I/O in time. For example, one might ask, when do heavy writes occur? Some applications write all the data at the beginning of the job, while others dump all the data near the end of the execution. Some applications show periodic/bursty I/O behavior. The primary question here is if it is possible to build a model to regenerate such a time-series I/O trend.

We use a neural network to generate such patterns. To train our neural network model, we extract time-series I/O patterns per job for the selected applications from darshan's fine-grained logs as discussed in Section 3.3 and summarize them into a normalized vector of 20 values. Each row of the training data consists of feature seeds generated by the FG and the intensity values representing I/O patterns over time. We can use any generative neural network method for the TG, such as Generative Adversarial Network (GAN) or Variational Autoencoder (VAE). However, the accuracy mileage can vary. We present a detailed evaluation of different methods in the next section.

6 EVALUATION

This section evaluates the different algorithms used for the two-step deep generative models, the FG and the TG. Among many state-of-the-art generative neural network methods, we explore the following neural network architectures in this paper; Normalizing Flow [39] for the FG and Multi-Layer Perceptron (MLP) [47],

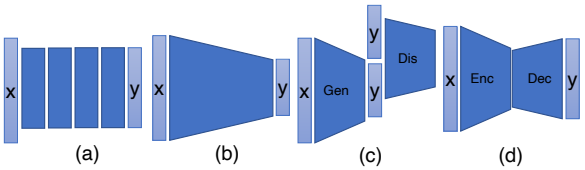


Figure 8: Schematic diagrams of deep generative models used in the paper: (a) Normalizing Flow, (b) Multi-Layer Perceptron (MLP), (c) Generative Adversarial Network (GAN), and (d) Variational Autoencoder (VAE).

Generative Adversarial Network (GAN) [19], and Variational Autoencoder (VAE) [24] for the TG. Figure 8 shows the schematic architectures. Brief descriptions of the models are as follows.

- Normalizing Flow: A neural network constructing complex distributions by transforming a probability density through a series of invertible mappings [39]. Normalizing flows provides a data-driven probabilistic sampler. This is a simplistic algorithm that is cost-effective and works best for the FG. However, the model is not able to handle the complicated stochastic traces needed to be generated in the TG. Therefore, we evaluate MLP, GAN and VAE for TG.
- MLP: It is a deep neural network architecture for multi-value regression. We integrate a residual network [20] with MLP to avoid the performance collapse with a deeper network.
- GAN: It uses two neural networks (generator and discriminator) working in an adversarial way to improve data generation accuracy.
- VAE: It is an extension of the autoencoder model with variational inference.

The darshan data sets are processed and categorized into three classes (scale 1-3) based on the I/O sizes for neural network training. All evaluations are performed on Summit. The I/O traces generated using our approach is validated against the baseline I/O traces generated from raw darshan logs inspired from the process described by Snyder et al. [46]. As both the I/O traces are time-series data, root-mean-squared-errors (RMSE) are used to get the accuracy of our model by showing how much the trace generated by our ML-model deviates from the trace generated from raw darshan logs.

6.1 Generative Power of TG only

Our first experiment is to solely measure the generative power of the TG (Trace Generator). We do not use the FG (Feature Generator) for this experiment to be able to compare these results with the results when FG is used, to better support the idea of the need for FG. For this purpose, we extract the N number of features directly from the darshan traces (i.e., not from the FG). Then, we train our data with three well-known deep generative models – MLP, GAN, and VAE – and measure the accuracies. Optimizing neural networks or hyperparameter tuning is not the main focus in this paper. However, we explore different number of layers and activation functions and use the optimal network architectures showing the best results. We summarize the architectures in Table 6.

Type	Model	#Layers	#Params	Input/Output
FG	NormalFlow	35	5K	1/upto 75
TG	MLP	16	82K	upto 75/20
	GAN	22	59K	upto 75/20
	VAE	25	57K	upto 75/20

Table 6: The list of neural network architectures used for evaluation and a brief information of the number of layers, parameters, and input/output sizes.

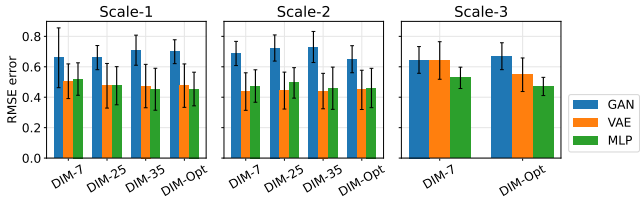


Figure 9: The RMSE errors of the generated traces in the TG only model. We compare three different generative models, GAN, VAE, and MLP, with three datasets (Scale 1-3). We add 2% random noises in the input. Overall, MLP shows the best performance compared with others.

Figure 9 shows how accurately different generative models can generate the darshan traces. We measure the generation errors by using root-mean-squared errors (RMSE) per model for each class (Scale-1, Scale-2, and Scale-3). We vary the number of features, ranging from 7, 25, 35, to 75, represented by DIM7, DIM25, DIM35, and DIMOpt (75 or 60 or 25 as discussed in Section 4.3). We add 2% random noises in the input to simulate a realistic use case. Overall, none of them achieves reasonable accuracy levels (say, below 0.2). However, we made two observations.

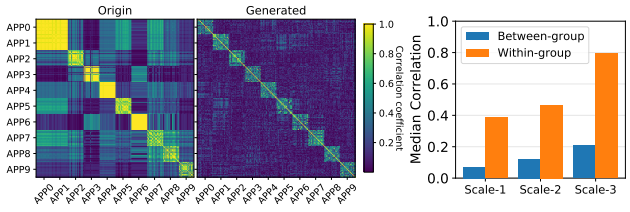
The first observation is the accuracy of different generative models. MLP showed the best accuracy compared to VAE and GAN. While GAN and VAE tend to find the average of the observation, MLP generates data closer to the training set. The second observation is that the size of input features (or dimensions) can affect the accuracy of the generative models. In the figure, the errors get smaller as the number of features increases. With the increased input dimensions, the expressiveness of the model can increase [13]. However, while a large number of features can help to build more accurate generative models, providing a large number of features as an input can be a difficult task for users. Users need to provide a set of similar numbers as an input to the model. Without the guarantee of the generalized model, the generative model can easily produce unexpected outputs. This is the reason why FG is required.

6.2 Efficiency of FG

The next experiment is to demonstrate the working of FG. We choose the Normalizing Flow [39] model to generate features for the TG. We compare with other generative models, such as GAN and VAE. We do not report a detailed comparison in this paper due to page limits. However, the normal flow shows better performance both in terms of accuracy and cost-effectiveness. One of the key properties that is pursued is the feature collision, where similar features from different applications mislead the TG to generate the same traces. We need a feature generator that can create features close to the training set where each application can have unique features sets different from the features from other application. Therefore, we measure how features from one application are different from the other applications by using the Pearson correlation score, defined by,

$$\rho(x, y) = \frac{\text{cov}(x, y)}{\sigma_x \sigma_y} \tag{5}$$

where, cov(x, y) is the covariance between two feature values, x and y, and σ represents the standard deviation.



(a) Pairwise correlations **(b) Median correlation**

Figure 10: Correlations of features generated from the feature generator (FG). (a) A pairwise correlation for Scale-1. The FG's normalizing flow model generates features with high correlations within the same application group but low correlations from other application groups, avoiding the feature collision problem. (b) Comparison of median correlation scores in the within-group and the between-group for the three data sets (Scale-1, Scale-2, and Scale-3).

Figure 10 shows the correlation coefficients of the features generated from the FG. Figure 10a shows all-to-all correlation coefficients for the large dataset. It is clear that FG generates features with high correlation within the same application groups and low correlations between other groups. Figure 10b shows the overall correlation coefficients (as median values) for all the data sets (Scale-1, Scale-2, and Scale-3).

6.3 Generative Power of FG+TG

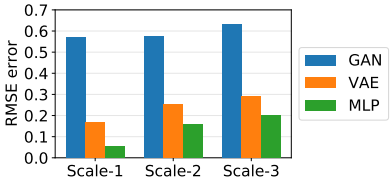
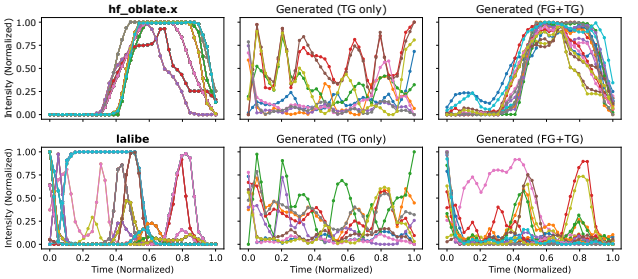
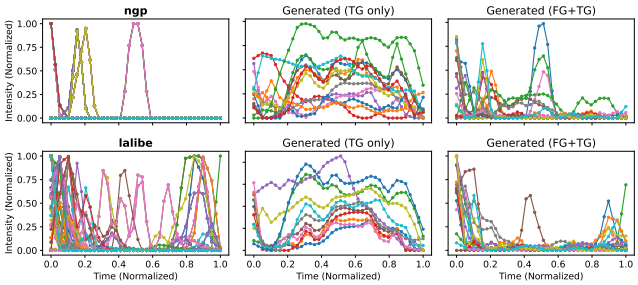


Figure 11: The generation errors of our two-step model (FG-TG) with the validation dataset. We apply different generative models (MLP, GAN, and VAE) for each data set scale (Scale-1, Scale-2, and Scale-3) and compare the differences measured with root-mean-squared-error (RMSE) between the original and the generated traces.



(a) Scale-2



(b) Scale-3

Figure 12: Example of generated write traces by using our proposed generative method. The first column has the original traces generated from raw darshan logs using a process similar to the approach described by Snyder et al. [46].

Finally, we experiment with the entire workflow, that is, using the FG and the TG together. We train the TG with only 90% of the data set and prepare 10% of the data for validation. The generation error from the TG is measured by using the features generated from the FG with the validation data set, which is not used for the TG training. The generation error measured in RMSE is shown in Figure 11. Compared with the RMSE errors by using on TG-only shown in Figure 9, the FG-TG workflow shows an improvement by dropping the RMSE from around 50% to less than 20% for all the data sets.

Figure 12 shows the examples of generated write traces from the FG-TG workflow for two applications. Three key observations can be made. First, our approach is able to generate traces for the same application (here, *lalibe*) for two different scales. Second, our approach is able to generate varied write patterns - periodic writes as well as one time bursty write. Third, for a particular scale, our approach is able to generate different traces which exhibit the realistic I/O performance variability in HPC clusters. The traces generated by our model is representative of the I/O traces on Summit as the model is trained using traces from raw darshan logs of applications running on Summit. In the future, we will train our model using darshan logs from other leadership-scale systems, so that we can compare the I/O traces from multiple HPC systems.

7 DISCUSSION

The aim of this work is to build an end-to-end ML workflow that is able to generate anonymized and realistic production based I/O traces from any HPC cluster. To this end, we use darshan traces collected for one year on Summit in ORNL and build a two-step generative model to generate time-series traces for read, write and the number of open files. In this section, we reiterate and discuss some of the design choices for the paper.

Why focus on workloads running on more than 92 nodes? The primary goal of this work is to help academicians who may not have access to large-scale HPC clusters and need publicly available I/O traces. Therefore, we focus on applications running only on more than 92 nodes. Moreover, Summit being a leadership scale system encourages users to run large-scale jobs by providing a longer walltime. This provides us a unique opportunity to work on applications that run longer and on large number of nodes, thereby leading to a large number of concurrent I/O operations on files that may lead to I/O congestion and I/O performance variability.

Why is a trace generator needed as a first step for raw Darshan files? Darshan logs provide per-file I/O data for all files accessed by the job during the job runtime, for example, the number of bytes read by a file, and the timestamp of the first I/O operation done by a file. Thus, darshan logs are different than I/O trace files (for example, strace) which provide timestamps for every I/O operation. Therefore, one of the most important tasks is to convert the darshan logs into time-series data. To do this, the approach described by Snyder et al. [46] is taken as this is the only work in literature that converts darshan logs into time-series traces. The time-series logs generated from raw Darshan files then become the training dataset as well as the baseline for the evaluations.

Why feature selection is important for different scales of applications? Prior literature that identify important features for I/O performance modeling do not consider the scale of an application run. In a HPC cluster, different scales of application runs generate different I/O access patterns, thereby making it imminent that different features are required to model I/O for applications running at different scales. Based on our experiments, we see that the scale of an application run might lead to the same feature being more important in one scale than the other scale. Moreover, for our proposed ML workflow, this is the first step to identify the list and size of important features in the training dataset for the feature generator.

Why is a two-step ML model needed to generate I/O traces? As discussed above, HPC application runs are prone to varied I/O performance. This suggests that ML models should be efficient to be able to generate different I/O traces for an application running at the same scale at multiple time instances. A two-step model ensures that I/O performance variability is taken into consideration. Moreover, it also helps in reducing bias/sensitivity that might be caused by overfitting the trace generator model. Having the feature generator also helps in anonymization and building a generic workflow as the only inputs that need to be provided are the application name and the scale of the application run.

Why is the I/O trace generation process not generic for all applications? I/O behavior of applications is highly variable. Some applications are write-intensive, some are read-intensive, while others have a combination of both reads and writes. Also, some applications can exhibit bursty I/O at different periods of time while

other applications might exhibit a uniform I/O pattern throughout their runtime. Additionally, different runs of the same application can have varying I/O patterns. This kind of variability in the I/O patterns makes it very difficult for trace generation approaches to be generic for all applications. In our approach, we try to solve the variability of I/O behavior for different runs of an application by providing only the application id and the scale of application run to the FG. In the future, we will try to classify the I/O patterns of all applications into different classes and generate I/O traces for each class of applications.

8 RELATED WORK

Extensive research on modeling, characterization and profiling of I/O behavior of HPC workloads has been presented in [15, 16, 27, 40, 44, 51, 52]. The application oriented profiling and characterization are the primary focus of the works in [25, 43]. Recently, machine learning techniques have been used to model IO pattern prediction [27, 28]. The important aspect of these machine learning models are that they try to take into account the stochastic nature of IO patterns. Since the availability of darshan data is generally limited to researchers from prominent institutes and laboratories, only some of these works could have used darshan datasets [12, 46]. However, datasets' privacy and security aspects make it challenging to share them in the public domain, thus limiting its impact on the broader scientific community. Therefore, our work in building machine learning generative models for trace generation will help the researchers who do not have access to large HPC systems logs.

Classification of HPC workloads using machine learning methods presented in [11, 35] is an exciting area of research for characterization of HPC applications' I/O behavior. There has also been some work in the areas of I/O trace generation. Snyder et al.[46] proposed three techniques for HPC workload generation. The authors of [10] have created an automated I/O kernel which generates the application traces as if the traces are generated from the actual application. The I/O kernel generation process begins with the trace files recorded by HDF5 instrumentation at each rank. The traces are then merged by order of I/O operations called by the application. A code generator will generate the I/O kernel from the combined dataset. Another work towards trace generation is [31], which utilizes ScalaTrace, a parallel application tracing framework to collect features that represent the run-time behavior of HPC workloads. The work is based on the understanding that the performance of an application is essentially a function of primary operations. Based on the above insight, they proposed application-specific performance benchmarks. The methodology takes MPI-based application as an input. The first task of the process is to use the ScalaTrace tool to record the applications' communication patterns, including intervening time. The collected trace is then fed to a benchmark generator. CONCEPTUAL [34] is a sophisticated program that is used to write a benchmark for application traces.

VAE is a variant of autoencoders that was first introduced in [41]. VAEs as a generative model has been applied in many fields such as speech recognition, computer vision, and timeseries classification. Usually, the goal of the VAE model is to reduce the gap between the distribution of generated and the original data [23, 39]. VAEs have also taken center stage in training deep learning models [48]. We did not find any prior work using VAE models in analyzing or modeling datasets in the storage and computer systems field. However, VAEs has been used in modeling and prediction of various kinds of time-series problems such as prediction of timeseries datasets related to IoT and financial timeseries, and pattern recognition for anomaly detection[9, 50].

9 CONCLUSION

Large-scale I/O traces are valuable in understanding applications' I/O behaviors and estimate system capacities. They are also essential to develop predictive methods for scheduling and system management. However, sharing large-scale I/O traces from leadership computing facilities, like ORNL, can cause security and privacy concerns. The cost of anonymizing such traces is huge. To overcome such issues, we develop an end-to-end ML workflow to generate I/O patterns for large-scale HPC workloads. We use Summit's darshan traces collected for a year and discuss how to utilize ML blocks towards building a complete generative model.

To this end, we first engineer and identify important features for different scales of application runs. Then these features are used to reduce the size of the darshan dataset and build a two-step generative model workflow to generate ORNL's I/O patterns for various applications running on Summit. The two-step model uses two deep generative models; the feature generator (FG) and the trace generator (TG). The FG generates the inputs (or features) to be used by the TG. The primary purpose is two-fold; i) to avoid feature collisions that can lower the performance of generation power of the FG and ii) to anonymize the features thus preventing security concerns. We compare and evaluate the effectiveness of the trace generation using different deep learning models for FG and TG. Our proposed generative model can generate I/O traces for different applications running at various scales and also generate different traces for an application in the same scale showcasing performance variability with less than 20% errors. In future, we plan to extend our model to generate I/O traces for different interfaces like MPI-IO as well as use our generative models as a surrogate to generate various I/O patterns for various learning tasks, such as I/O optimization and scheduling.

ACKNOWLEDGMENT

This work used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. We thank Sarah Neuwirth for Figure 3. We are also extremely thankful to the reviewers and our shepherd, Peter Dinda, for their valuable feedback.

AVAILABILITY

Our analysis and models are made available in a public repository[1]. The repository contains the entire workflow of the paper including the feature selection process and the database containing the mutual information scores for the features for all the three scales. The training modules for FG and TG are also given as jupyter notebooks. The pre-trained models stored in the 'data' folder can be used to generate traces of the applications.

[1] https://github.com/at-aaims/HPDC22_darshan_trace_generation

REFERENCES

[1] [n.d.]. Darshan - HPC I/O Characterization Tool. https://www.mcs.anl.gov/research/projects/darshan/. Accessed: July 17 2021.

[2] [n.d.]. Darshan Data. https://www.mcs.anl.gov/research/projects/darshan/data/. Accessed: January 5 2022.

[3] [n.d.]. Fugaku Supercomputer. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/. Accessed: April 20 2022.

[4] [n.d.]. Lustre Monitoring Tool. https://github.com/LLNL/lmt. Accessed: October 12 2021.

[5] [n.d.]. Perlmutter Supercomputer. https://www.nersc.gov/systems/perlmutter/. Accessed: April 20 2022.

[6] [n.d.]. Summit. https://www.olcf.ornl.gov/summit/. Accessed: July 17 2021.

[7] [n.d.]. Top 500 - November 2021. https://www.top500.org/lists/top500/2021/11/. Accessed: November 30 2021.

[8] Jiwoo Bang, Chungyong Kim, Kesheng Wu, Alex Sim, Suren Byna, Sunggon Kim, and Hyeonsang Eom. 2020. HPC Workload Characterization Using Feature Selection and Clustering. In *Proceedings of the 3rd International Workshop on Systems and Network Telemetry and Analytics*. 33–40.

[9] Wei Bao, Jun Yue, and Yulei Rao. 2017. A deep learning framework for financial time series using stacked autoencoders and long- short term memory. *Plos one* 12 (2017). https://doi.org/.https://doi.org/10.1371/journal.pone.0180944

[10] Babak Behzad, Hoang Vu Dang, Farah Hariri, Weizhe Zhang, and Marc Snir. 2014. Automatic generation of I/O kernels for HPC applications. *Proceedings of PDSW 2014 Held in Conjunction with SC 2014* (2014), 31–36. https://doi.org/10.1109/PDSW.2014.6

[11] Eugen Betke and Julian Kunkel. 2020. The Importance of Temporal Behavior When Classifying Job IO Patterns Using Machine Learning Techniques. *Lecture Notes in Computer Science* 12321 LNCS (jun 2020), 191–205. https://doi.org/10.1007/978-3-030-59851-8_12

[12] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24 / 7 Characterization of Petascale I / O Workloads. (2009).

[13] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the expressive power of deep learning: A tensor analysis. In *Conference on learning theory*. PMLR, 698–728.

[14] Emily Costa, Tirthak Patel, Benjamin Schwaller, Jim M Brandt, and Devesh Tiwari. 2021. Systematically inferring I/O performance variability by examining repetitive job behavior. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[15] Sheng Di, Hanqi Guo, Eric Pershey, Marc Snir, and Franck Cappello. 2019. Characterizing and Understanding HPC Job Failures over the 2K-Day Life of IBM BlueGene/Q System. *Proceedings - 49th DSN 2019* (2019), 473–484. https://doi.org/10.1109/DSN.2019.00055

[16] Wenrui Dong, Guangming Liu, Jie Yu, and You Zuo. 2015. Characterizing I/O workloads of HPC applications through online analysis. *2015 IEEE 34th International Performance Computing and Communications Conference* (2015), 15–16. https://doi.org/10.1109/PCCC.2015.7410353

[17] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*, Vol. 96. 226–231.

[18] Pablo A Estévez, Michel Tesmer, Claudio A Perez, and Jacek M Zurada. 2009. Normalized mutual information feature selection. *IEEE Transactions on neural networks* 20, 2 (2009), 189–201.

[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems* 27 (2014).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition*. 770–778.

[21] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[22] Mihailo Isakov, Eliakin Del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B Ross, and Michel A Kinsy. 2020. HPC I/O throughput bottleneck analysis with explainable local models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[23] Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. 2016. Improved variational inference with inverse autoregressive flow. *Advances in Neural Information Processing Systems* Nips (2016), 4743–4751. arXiv:1606.04934

[24] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[25] Zhengchun Liu, Ryan Lewis, Rajkumar Kettimuthu, Kevin Harms, Philip Carns, Nageswara Rao, Ian Foster, and Michael E. Papka. 2020. Characterization and identification of HPC applications at leadership computing facility. *Proceedings of ICS* (2020). https://doi.org/10.1145/3392717.3392774

[26] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. 2013. A multi-level approach for understanding I/O activity in HPC applications. In *2013 IEEE CLUSTER*. IEEE, 1–5.

[27] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan Wild. 2018. Modeling I/O performance variability using conditional variational autoencoders. In *2018 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 109–113.

[28] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M. Wild. 2017. Analysis and Correlation of Application I/O Performance and System-Wide I/O Activity. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*. 1–10. https://doi.org/10.1109/NAS.2017.8026844

[29] Michael P Mesnier, Matthew Wachs, Raja R Simbasivan, Julio Lopez, James Hendricks, Gregory R Ganger, and David R O'hallaron. 2007. Trace: parallel trace replay with approximate causal events. (2007).

[30] Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. 2016. Enabling parallel simulation of large-scale HPC network systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 87–100.

[31] Frank Mueller and Scott Pakin. 2011. Automatic Generation of Executable Communication Specifications from Parallel Applications Categories and Subject Descriptors. *ICS '11* (2011), 12–21.

[32] Mohammad Abu Obaida and Jason Liu. 2017. Simulation of HPC job scheduling and large-scale parallel workloads. In *2017 Winter Simulation Conference (WSC)*. IEEE, 920–931.

[33] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. 2016. Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328* (2016).

[34] Scott Pakin. 2007. The design and implementation of a domain-specific language for network performance testing. *IEEE Transactions on Parallel and Distributed Systems* 18, 10 (2007), 1436–1449. https://doi.org/10.1109/TPDS.2007.1065

[35] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. 2020. Job Characteristics on Large-Scale Systems. In *SC'20*.

[36] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. 2020. Understanding hpc application i/o behavior using system level statistics. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 202–211.

[37] Arnab K Paul, Ahmad Maroof Karimi, and Feiyi Wang. 2021. Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.

[38] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. 2017. On the expressive power of deep neural networks. In *international conference on machine learning*. PMLR, 2847–2854.

[39] Danilo Rezende and Shakir Mohamed. 2015. Variational inference with normalizing flows. In *International conference on machine learning*. PMLR, 1530–1538.

[40] Gonzalo P Rodrigo, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. 2015. HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems. *HPDC* (2015), 57–60. https://doi.org/10.1145/2749246.2749270

[41] David E. Rumelhart and James L. McClelland. 1987. *Learning Internal Representations by Error Propagation*. 318–362.

[42] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. 2020. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*. PMLR, 8459–8468.

[43] Hongzhang Shan, Katie Antypas, and John Shalf. 2008. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. *2008 SC* (2008). https://doi.org/10.1109/SC.2008.5222721

[44] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. 2017. Modular HPC I/O characterization with Darshan. *Proceedings of ESPT 2016 Held in conjunction with SC 2016* (2017), 9–17. https://doi.org/10.1109/ESPT.2016.006

[45] Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross, Christopher Carothers, Babak Behzad, Huong Vu Thanh Luu, and Surendra Byna. 2015. Techniques for modeling large-scale HPC I/O workloads. In *Proceedings of the 6th PMBS*. 1–11.

[46] Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross, Christopher Carothers, Babak Behzad, Huong Vu, Thanh Luu, and Surendra Byna. 2015. Techniques for Modeling Large-Scale HPC I/O Workloads. In *SC'15*. Austin, TX. https://doi.org/10.1145/2832087.2832091

[47] Donald F Specht et al. 1991. A general regression neural network. *IEEE transactions on neural networks* 2, 6 (1991), 568–576.

[48] Jakub Tomczak and Max Welling. 2018. VAE with a VampPrior. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 84)*, Amos Storkey and Fernando Perez-Cruz (Eds.). PMLR, 1214–1223. https://proceedings.mlr.press/v84/tomczak18a.html

[49] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C Roth. 2009. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. 26–31.

[50] Chunyong Yin, Sun Zhang, Jin Wang, and Neal N. Xiong. 2020. Anomaly Detection Based on Convolutional Recurrent Autoencoder for IoT Time Series. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2020), 1–11. https://doi.org/10.1109/tsmc.2020.2968516

[51] Xiang Yin, Yanni Han, Zhen Xu, and Jie Liu. 2021. VAECGAN: a generating framework for long-term prediction in multivariate time series. *Cybersecurity 2021 4:1* 4, 1 (jul 2021), 1–12. https://doi.org/10.1186/S42400-021-00090-W

[52] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled infiniband clusters. *Proceedings of IPDPS 2016* (2016), 1777–1784. https://doi.org/10.1109/IPDPSW.2016.178

[53] Ningning Zhu, Jiawu Chen, Tzi-Cker Chiueh, and Daniel Ellard. 2005. TBBT: scalable and accurate trace replay for file server evaluation. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (2005), 392–393.

[54] Yinhao Zhu, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris. 2019. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *J. Comput. Phys.* 394 (2019), 56–81.