A2FL: Autonomous and Adaptive File Layout in HPC through Real-time Access Pattern Analysis

Dong Kyu Sung[1], Yongseok Son[2], Alex Sim[3], Kesheng Wu[3], Suren Byna[3,4], Houjun Tang[3], Hyeonsang Eom[1], Changjong Kim[5], and Sunggon Kim[5,✉]

[1]*Seoul National University*, South Korea
[2]*Chung-Ang University*, South Korea
[3]*Lawrence Berkeley National Laboratory*, USA
[4]*The Ohio State University*, USA
[5]*Seoul National University of Science and Technology*, South Korea

***Abstract*—Various scientific applications with different I/O characteristics are executed in HPC systems. However, underlying parallel file systems are unaware of these characteristics of applications, and using a single fixed file layout for all applications can degrade the performance of HPC systems. In this paper, we propose A2FL, an autonomous and adaptive file layout adjustment scheme that optimizes parallel file system configurations by analyzing the access pattern of the applications. The key steps of `A2FL` are as follows: (1) `A2FL` initially intercepts the I/O operations of the application, recording their access patterns in real-time. (2) The access patterns are then transformed into a graphical representation used for predicting I/O performance and providing adjustment recommendations. (3) `A2FL` autonomously adjusts the file layout based on the prediction results, delivering an optimal file layout within the parallel file system. Moreover, we propose `A2FL-Compound` which analyzes an access pattern by dividing it into smaller components to optimize the file layout in a fine-grained manner. Our evaluations demonstrate that `A2FL` significantly enhances I/O performance, with improvements of up to 65.9× compared to the default file layout.**

***Index Terms*—HPC, I/O, Access Pattern, Machine Learning, Storage, Parallel File System**

I. Introduction

In high-performance computing (HPC) systems, many users execute large-scale applications that generate terabytes (TBs) to petabytes (PBs) of data [1]–[5]. To efficiently output such large volumes of data, they utilize multiple processes to perform I/O, exhibiting unique I/O behaviors [6]–[8]. For example, multiple processes in an application can perform I/O with different file-sharing strategies. As a "file-per-process" strategy [9], each process creates and performs I/O on its own file. On the other hand, in a "shared file" strategy [10], one or more processes can perform I/O on a single file. In addition, a single application can exhibit different I/O behaviors based on various configurations. Specifically, a user can adjust the application parameters, such as the shape of input data and the number of iterations; furthermore, the user can set the degree of parallelism (e.g., the number of compute nodes and processes) differently. Consequently, the configuration changes can alter the distinct I/O access patterns of the application, including intervals between I/O operations and the ordering of I/O operations. Therefore, a comprehensive understanding of applications and configurations is necessary to efficiently store large amounts of data.

To handle the diverse I/O behaviors of applications, HPC systems employ parallel file systems, such as Lustre, Spectrum Scale, Ceph, and BeeGFS [11]–[14]. By distributing I/O requests from applications to multiple storage servers and utilizing them simultaneously, parallel file systems can meet the requirements of I/O-intensive applications. The parallel file systems offer diverse configurations such as stripe count and size, which determine the number of storage targets to be used for parallel I/O and the size of each data block, respectively. In addition, some file systems provide fine-grained data distribution, such as Progressive File Layout (PFL), which increases the stripe count and size as a file size grows while an application is performing write operations on the file. While these configurations can enhance performance through parallelism, finding the optimal configuration can be challenging and may even result in performance degradation compared with the default configuration. Therefore, it is important to analyze I/O behaviors, such as the access patterns of the applications, to efficiently adjust the configuration of the file systems.

Many studies have analyzed the I/O characteristics of applications and adjusted the configuration of parallel file systems for I/O optimization. Kim et al. [15] design and develop a scheme to predict the I/O performance of HPC applications using integrated system logs that record the I/O characteristics of the applications. Behzad et al. [16] target HDF5-based applications and utilize a generic algorithm to auto-tune parameters across the entire I/O stack, spanning from applications to the storage server. CAPE [17] monitors the bandwidths of workloads and configures parallel file system settings, such as RPC congestion and I/O rate limits on the server side. Our study is in line with these previous studies in terms of analyzing I/O behavior during application executions and examining configuration parameters. However, our work focuses on real-time detection of the I/O access patterns of applications and enabling autonomous file layout adjustments.

In this paper, we propose `A2FL`, an autonomous and adaptive file layout adjustment scheme that optimizes file layout

✉ Corresponding author: Sunggon Kim

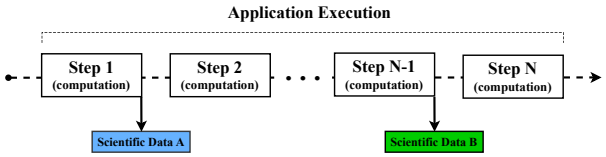


Fig. 1: HPC Applications with Multiple Steps.

by adjusting striping configurations based on real-time access pattern analysis of applications. To achieve this, A2FL first collects the access pattern of an application in real-time using an I/O tracing tool. Then, it creates images from the access pattern logs, which are used in conjunction with application and striping configurations to build machine learning (ML) models that predict the I/O performance of the application using different striping configurations. Finally, A2FL selects the striping configuration that maximizes the performance of the application and autonomously changes the file layout. We evaluate A2FL using a large-scale production HPC system with a state-of-the-art parallel file system equipped with NVMe SSDs. The evaluation results, based on real HPC application benchmarks, demonstrate that A2FL can improve I/O performance by up to 65.9× compared to the default file layout.

The contributions of the study are as follows:

- We analyze access patterns of HPC applications using a large-scale and state-of-the-art production HPC system.
- By employing a lightweight ML-based approach using images of access patterns, A2FL can predict and set optimized file layout for applications without user intervention.
- We evaluate A2FL in a real-world HPC system with HPC benchmarks, and demonstrate that A2FL can enhance performance through file layout adjustments.

To the best of our knowledge, our work is the first study to analyze the I/O access patterns of HPC applications in real-time for the purpose of autonomously adjusting and optimizing file layout during the runtime of the application.

II. Background and Motivation

A. HPC Applications and I/O Access Patterns

Many HPC applications [1]–[5], [18] perform computations and simulations on large-scale scientific data. While the access pattern of these applications can change due to their inherent I/O behavior, a single application can also exhibit diverse access patterns when using different configurations, making I/O performance difficult to predict and optimize in an HPC system [15], [19]. For example, an application can run with a different number of nodes and processes, which are user-adjustable parameters that change the degree of parallelism within the same application. Additionally, each process can be configured to write data into separate files in a file-per-process manner or into a single shared file, which can alter the I/O behavior. Moreover, the order of data blocks can be configured to be either contiguous or interleaved in terms of the logical file offset. Due to these various I/O access patterns, different applications, or even identical applications, can exhibit different performance characteristics, making striping configurations challenging to optimize based on a single rule or configuration.

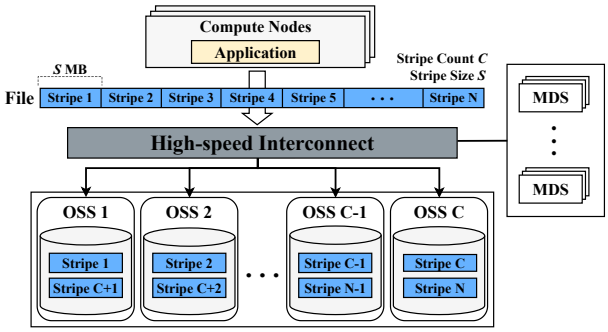


Fig. 2: Striping and File Layout in the Lustre File System.

While the I/O behavior can change between applications and executions due to differences in configurations, it does not typically change during a single application execution cycle. Figure 1 illustrates an example procedure of an HPC application during its execution. As depicted in the figure, multiple computation and I/O tasks exist within a single execution cycle and the applications frequently perform I/O operations in between computations to write the intermediate results of processing large data. Each computational state or unit is referred to as a "step", and intermediate I/O operations, such as storing the result through checkpointing, can be configured to occur after a certain number of steps. For example, many scientific applications [18], [20], [21] can be configured to repeat a certain number of steps for particle simulations, and checkpointing can also be configured to repeatedly occur after every specified number of steps. However, during execution, application parameters such as the number of nodes and processes remain fixed, as do I/O behaviors such as the number of files to be written during a single step. Thus, if the application's I/O behavior can be analyzed and the striping configuration can adapt to this behavior, I/O operations in subsequent steps can benefit from the optimized configurations.

B. Parallel File System

To handle the large I/O demands from various applications, HPC systems deploy parallel file systems as the underlying storage layer. Among many parallel file systems, Lustre [11] is widely used in many HPC systems. Figure 2 shows the architecture of the Lustre file system. As depicted in the figure, Lustre consists of metadata servers (MDSes) and object storage servers (OSSes). The MDSes manage metadata such as file names and directories, while the OSSes receive and store data blocks in a set of storage devices called object storage targets (OSTs).

To allow users to adjust the parallelism based on the I/O behavior, Lustre and many other file systems (e.g., Spectrum Scale, Ceph, and BeeGFS) provide a feature called file

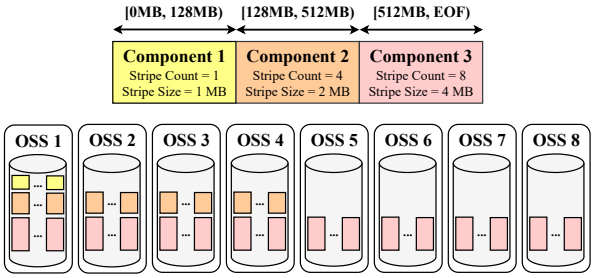


Fig. 3: Example of Lustre Progressive File Layout (PFL).

striping. Through striping, data from a single file is divided into multiple data stripes with a fixed size (stripe size) and distributed among the configured number of OSSes (stripe count). As shown in Figure 2, a file being written by the application is divided into multiple stripes. The file has been configured with a stripe size of S MB and a count of C. Therefore, the file is split into S MB stripes, which are then distributed and stored across C OSSes. Using large stripe size and count can improve sequential I/O performance and reduce metadata overhead but may be inefficient for small and random I/O operations. Conversely, selecting a smaller stripe size and count is appropriate for small, random I/O operations and ensures more balanced data distribution, though it may raise metadata overhead and potentially decrease sequential I/O performance. Thus, by giving different striping configurations, the file layout and data distribution change, and it is essential to consider the I/O pattern of the application carefully to choose the optimal configuration.

In addition to file striping, Lustre has recently introduced a feature called Progressive File Layout (PFL) that allows for more fine-grained adjustments to the striping. In contrast to traditional file striping, where the configured stripe count and size remain constant for an entire file, PFL enables the setting of stripe count and size independently for different components in the file. Figure 3 shows an example of a file with PFL configuration. As shown in the figure, there are multiple components (Component 1, 2, and 3) in a single file. The first 128MB of file data (Component 1) is divided into 1MB stripes and will be stored in a single OSS (OSS 1). The next portion of the file, up to 512MB (Component 2), will be written in 2MB stripes and distributed among 4 OSSes (OSSes 1 to 4). Finally, the remaining part of the file (Component 3) will consist of 4MB stripes and be distributed among 8 OSSes (OSSes 1 to 8). By configuring PFL, the striping options of a file can change dynamically without user intervention. However, this also necessitates a deep understanding of the I/O behavior of applications and fine-grained striping configuration based on access patterns.

III. Autonomous and Adaptive File Layout

In this section, we present the design of A2FL which analyzes diverse access patterns of applications in real-time and autonomously adjusts the striping configuration to enhance the file layout, improving the I/O performance of HPC applications.

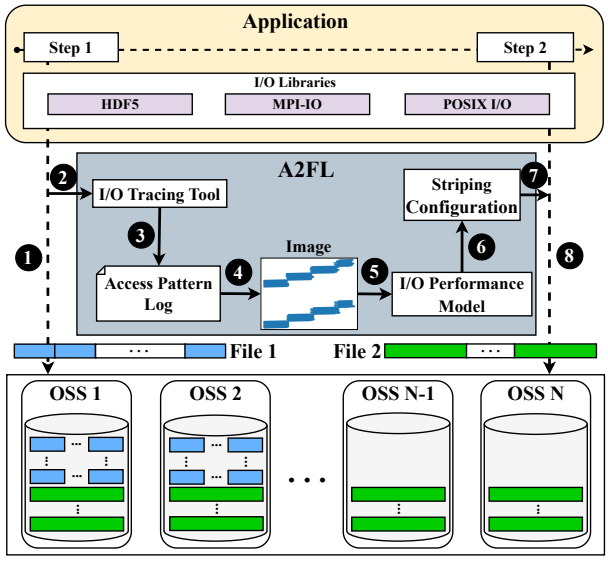


Fig. 4: Overall Procedure of A2FL.

A. Overall Procedure

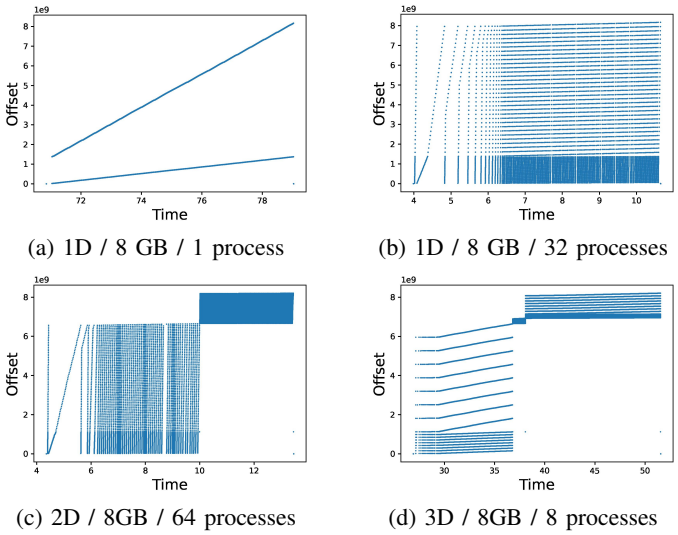
Figure 4 shows the overall procedure of A2FL after the user enables A2FL and executes an application. As shown in the figure, the application performs I/O operations on a file using various I/O libraries such as HDF5, MPI-IO, and POSIX I/O. In the initial step, the application writes a file (i.e., File 1) with the default striping configuration, including stripe count and size, represented by blue stripes (❶). To adjust the striping configuration, A2FL first intercepts and collects the I/O operations of the application using the I/O tracing tool (❷). Then, the traced access pattern (i.e., process ID, accessed file offsets, I/O request size, timestamps, etc.) is recorded as an access pattern log (❸). With the access pattern log for the file, A2FL generates a graph image, from which unique features representing the access pattern are extracted (❹). Then, the extracted features from the image, along with application configuration (e.g., the number of nodes/processes and file size) and striping configuration (e.g., stripe count and size), are used as input features to create an I/O performance model (❺). The I/O performance model is used to predict the performance of the application with a specific access pattern while varying the striping configuration. A2FL creates one model per application if the model for the application does not exists and trains the model to update with various access patterns from the application. The training of the model is performed periodically when multiple new access pattern logs are collected. By examining the predicted I/O performance values, A2FL selects the striping configuration that is predicted to have the best performance (❻). Finally, A2FL adjusts the optimized striping configuration for the application directory (❼), and subsequent I/O operations from the application can use the optimized file layout for the next file (i.e., File 2) (❽). Thus, model building, prediction, and selection of the optimal configuration are performed autonomously by A2FL without user intervention except for enabling A2FL at the start of execution.

B. Tracing I/O Access Pattern

To analyze the distinct access pattern of applications, it is important to first trace the access pattern using the I/O tracing tool. A2FL utilizes Darshan I/O characterization tool [22] for detailed access pattern tracing and Recorder [23] for real-time access pattern tracing. Darshan is developed by Argonne National Laboratory and is widely used in for monitoring I/O in many large-scale and production HPC systems. It can collect I/O information from various I/O libraries such as HDF5, MPI-IO, and POSIX I/O without requiring any code modification of the application. In this work, we focus on POSIX I/O because various I/O libraries ultimately use the POSIX I/O library to perform I/O operations on the parallel file system. To trace the detailed I/O information, the Darshan eXtended Tracing (DXT) tool can be enabled to collect the access pattern (e.g., process ID, accessed file offsets, request size, and I/O timestamps). DXT first stores the access pattern of an individual file in memory and then writes it to a log file after the application finishes its execution. The log file can then be parsed to retrieve the access pattern and the performance (i.e., throughput in MB/s). Thus, by using DXT, A2FL records the access pattern log for each file from the application. This captures the correlation between the access pattern, striping configuration, and performance. After collecting a large set of access pattern logs, A2FL uses this set as the training data for the machine learning (ML) model.

While DXT captures the detailed access pattern of the application, the collected access pattern is only available after the application execution. Thus, to analyze the access pattern of the current file in the currently running application and suggest an optimized file layout for the subsequent file, we need another tracking tool with real-time tracing. To do this, A2FL uses Recorder [23]. Similar to Darshan, Recorder is an I/O tracing tool that intercepts all I/O functions from libraries and interfaces, such as HDF5, MPI-IO, and POSIX I/O. However, we have modified Recorder to write access pattern information, such as timestamps and file offsets of I/O operations, into a log file at every invocation of the I/O function call rather than at the end of the execution. For example, *pwrite* is a POSIX I/O system call that takes a file descriptor, buffer address, data size, and file offset as parameters. Then, Recorder intercepts the function call, and the modified Recorder immediately records values such as the timestamp, file offset, request size, and process ID to the access pattern log to collect the I/O access pattern. Thus, the combination of DXT and Recorder allows A2FL to analyze the detailed access pattern after execution and perform real-time analysis of time and offset, respectively. After DXT or Recorder traces and collects access patterns, A2FL uses other tools such as SLURM [24] to obtain additional application configurations, such as the number of nodes and processes and includes them in the access pattern log.

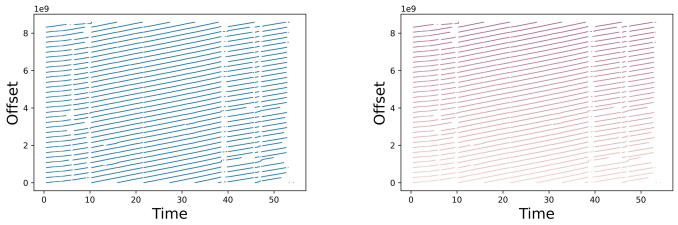
A2FL utilizes various tools to retrieve information that can determine the access pattern of an application. Since different applications have unique I/O access patterns depending on various I/O characteristics, it is important to include I/O patterns such as file offsets accessed by I/O operations and their timestamps and application configurations such as the numbers of nodes or processes to define a certain access pattern. In addition, the I/O tracing tools are capable of distinguishing multiple unique files even in a single step of the application execution. After tracing I/O traces of the application, one or more final access pattern logs can then be utilized in both performance prediction and striping configuration adjustment.



(a) 1D / 8 GB / 1 process (b) 1D / 8 GB / 32 processes

(c) 2D / 8GB / 64 processes (d) 3D / 8GB / 8 processes

Fig. 5: Access Pattern Images running AMReX-HDF5: Each image is generated from the same application with different configurations.



(a) Without hue. (b) With hue.

Fig. 6: Access Pattern Graph Images running H5bench-Write: Without and With Hue.

C. Creating Image of Access Pattern

With the traced I/O access pattern logs, A2FL can investigate various access patterns of applications and how each of them performs on different striping configurations. HPC applications perform I/O on a large scale and create a vast number of data points in a single access pattern at each step in application execution, consisting of various values of features (i.e., process ID, file offsets) by different timestamps. Generally, to analyze and transform time-ordered data, feature engineering tools [25], [26] are used to process and encode the data. However, it can be challenging to process and analyze such data in terms of computational costs.

On the other hand, processing access pattern logs by transforming them into graphical representations can be effective

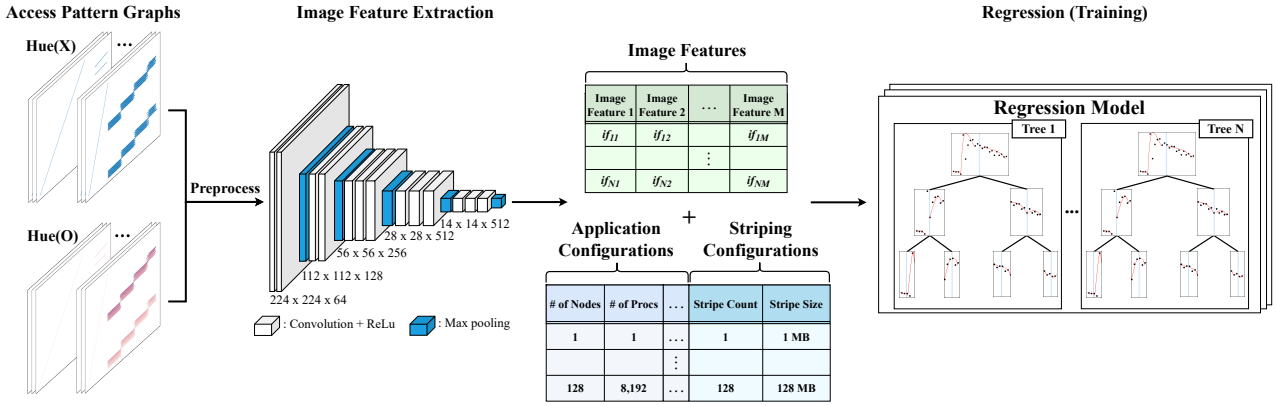


Fig. 7: Overall Procedure for Training the I/O Performance Model.

by summarizing and representing numerous data points of the single access pattern. Many previous studies [27]–[30] have shown that transforming univariate or multivariate time series data into an image can lower the complexity, improve the accuracy of the ML model, and reduce the computational costs. In terms of the accuracy in R^2 score, the image-based regression model and the encoding-based regression model can achieve up to 0.940 and 0.941, respectively, in our experiments. This shows that the image-based method can be exploited without losing any critical information on the I/O access pattern compared with the traditional encoding-based method and it can successfully detect the differences among various access patterns. However, the times for processing the log in the formats of an image and encoded data take up to 0.54 and 522 seconds, respectively, where the log is recorded from a 64GB file written by H5bench-Write [31] with a single process and it has 32,768 data points, the smallest access pattern log used in the experiment. This demonstrates that the encoding-based method can be inappropriate for real-time processing. Therefore, `A2FL` transforms access pattern logs into graph images to perform the prediction using ML models in real-time.

Figure 5 displays graph images generated by running AMReX-HDF5 [32] with various configurations of access patterns. The x-axis represents time, while the y-axis represents the file offsets accessed at that time. As shown in the figure, even if the application is identical, adjusting configurations such as the dimension of data and the number of processes can alter the access pattern. Since the access pattern impacts the optimal striping configuration, as explained in Section II-A, it is crucial to identify the access patterns of applications through analysis. Furthermore, as depicted in the figures, differences in the images confirm that distinct access patterns can be distinguished. This demonstrates that `A2FL` can effectively analyze the access pattern by examining these images.

However, some images of individual access patterns might have only marginal differences since A2FL generates 2D graph images using file offset and time. This can make it difficult to distinguish between such images from a certain application. To make access patterns more distinguishable using images, we add hue (color) to the data point based on the process ID that performs the I/O operation. Figure 6 shows an example of access pattern graph images with and without hue when running H5bench-Write. As shown in the figure, the two images have the same access pattern as the application, and the configurations are identical. However, with hue, as shown in Figure 6b, the images have data points with various colors which are designated by process ID. Depending on the application and its access pattern, images with hue can provide more information when timestamp and file offset are insufficient for distinguishing the access pattern. `A2FL` trains the model both with and without hue, selecting the model that performs better for each application.

D. Training the I/O Performance Model

With application configurations (e.g., the number of nodes/processes and file size) from SLURM and striping configurations (e.g., stripe count and size) from Darshan and extracted features from the image, `A2FL` builds and trains ML-based performance model that predicts the I/O performance of the application. This is to estimate how an application with a certain I/O access pattern performs with different striping configurations and to find the optimal file layout that maximizes performance.

Figure 7 shows the overall training procedure of the I/O performance model in `A2FL`. As shown in the figure, the images are first preprocessed to extract image features. `A2FL` resizes the height and width of the images to the same length (e.g., 256) and crops them at the center to remove edges that contain inconsistent data such as axis titles and ticks. Then, it converts the images into vectors and subsequently normalizes these vectors. This process is done to feed the image to the model as a vector to extract features. For the feature extraction, `A2FL` utilizes VGG16 [33], which is a widely used convolutional neural network (CNN) for image data processing. After the image vector is processed through VGG16, it will output a vector containing the extracted features. Finally, the application configuration and striping configuration of a single execution are fed as a single row alongside the image features to train the I/O performance model.

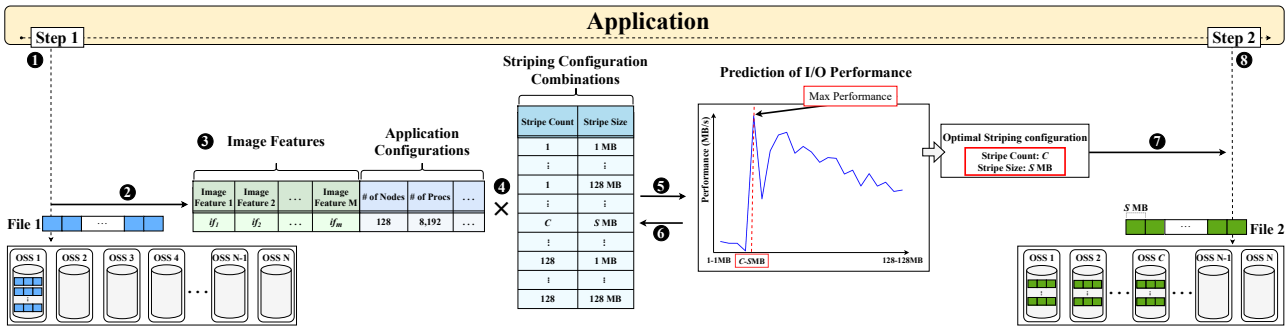


Fig. 8: Overall Procedure for I/O Performance Prediction and Optimal Striping Configuration Determination.

To build the I/O performance model, we create a model on a per-application basis. After a model for a specific application is built, we reuse and retrain the model for future application executions to reduce the cost and improve the accuracy of the model. When building the model for the first time, A2FL employs multiple regression models and identifies the best-performing model for each application. It is important to test different models because applications exhibit various access patterns, and input features from the application might impact the models differently. Consequently, the best model for accurately predicting performance may vary for different applications. To accomplish this, A2FL constructs the I/O performance models using the following regression models: Random Forest [34], Gradient Boosting [35], Histogram-based Gradient Boosting [36], Extreme Gradient Boosting [37], and Extra Trees [38]. Then, to determine the best-performing model, A2FL compares the models using the coefficient of determination or R^2 score. A high R^2 score is crucial for the trained model because it signifies a lower variance between predicted and actual performance. Consequently, A2FL can accurately predict performance trends based on various combinations of striping parameters.

E. I/O Performance Prediction and File Layout Adjustment

When an application is executed and a file is written during the first step, A2FL autonomously analyzes the file and optimizes the file layout of the application. Figure 8 illustrates the overall procedure for predicting I/O performance and determining the optimal striping configuration. As depicted in the figure, an application writes a file during step 1 (❶). After the file is written, A2FL traces the access pattern, creates an image, and extracts features from the image (❷). The input for the model includes the image features extracted from the currently running application and the application configuration (❸). Then, A2FL performs a Cartesian product to include multiple combinations of striping configurations in the input (❹). For example, A2FL can generate possible striping configuration combinations for the HPC system ranging from a stripe count of 1 to 128 and a stripe size of 1MB to 128MB, respectively. By creating more data points with multiple combinations of striping parameters, A2FL can explore several predicted performance values based on various striping configurations.

With the results of the Cartesian product, A2FL runs the model with the testing data (❺). As a result, the model produces a list of predicted I/O performance values. These predicted performance values will then depict a trend depending on the striping configurations, as shown on the right-hand side of the figure. After investigating the performance prediction results, A2FL can determine the single striping configuration (i.e., stripe count C and size S MB) that results in the maximum performance (❻). A2FL then adjusts the striping configuration of the application directory in the file system to modify the file layout and enhance I/O performance (❼). Subsequently, the application begins writing the next file in step 2 with the optimized file layout (❽).

F. Compound File Layout

Since A2FL optimizes the file layout by setting the single optimized striping configuration for the entire file, it cannot adapt to applications with varying access patterns within a single file. However, it collects the access pattern in real-time and can examine how performance changes during I/O. In other words, A2FL can track changes in access patterns as I/O operations are executed and adjust the file layout accordingly.

To explore multiple access patterns within a single file in a fine-grained manner, we propose A2FL-Compound, which collects multiple fine-grained logs divided by a certain period of time or data size and adjusts a compound striping configuration for a single file. For instance, A2FL-Compound can divide an access pattern log of a file into multiple components by 5-second intervals or 8GB-sized portions of the file. In this paper, we use the size as a unit to divide the access patterns, as I/O times can vary significantly between applications. For setting the striping configuration from A2FL-Compound, we use the PFL feature of Lustre to set different configurations to the different parts of the file.

Figure 9 shows the procedure of A2FL-Compound. As shown in the figure, an access pattern log is recorded from the I/O operations (i.e., Step 1) (❶). A2FL-Compound retrieves the log (❷) and divides it into smaller components based on a certain file size (e.g., 8GB), generating multiple graph images (❸). Using these images, A2FL-Compound generates testing data and performs predictions. The data preparation and running of the ML model for prediction are performed in the same way as explained in Sections III-D and III-E.

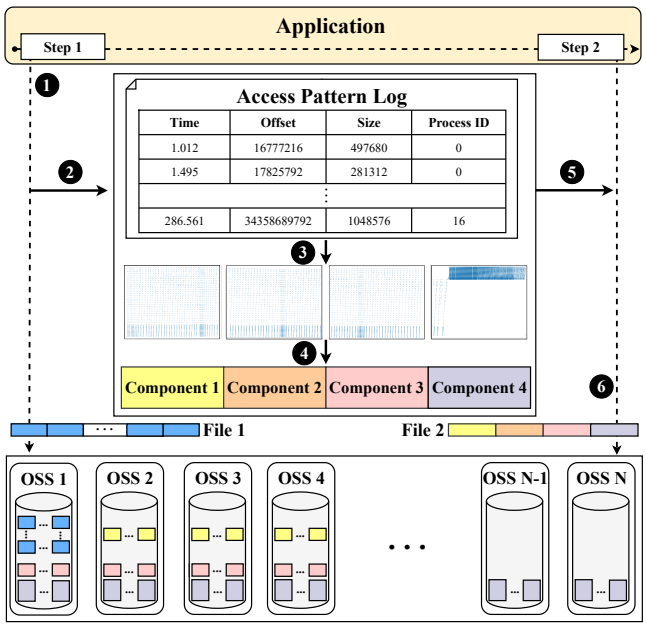


Fig. 9: Overall Procedure of A2FL-Compound.

The difference is that A2FL-Compound performs prediction and finds the optimal striping configuration for each divided component of the file rather than the entire file (❹). Then, A2FL-Compound applies the compound striping configuration for the file to the file system (❺). The next I/O operation from the application (i.e., Step 2) will use the new compound striping configuration to improve the I/O performance (❻). Therefore, A2FL-Compound can suggest a more fine-grained file layout than A2FL for adapting to a file with varying access patterns and optimizing the file layout.

IV. Evaluation

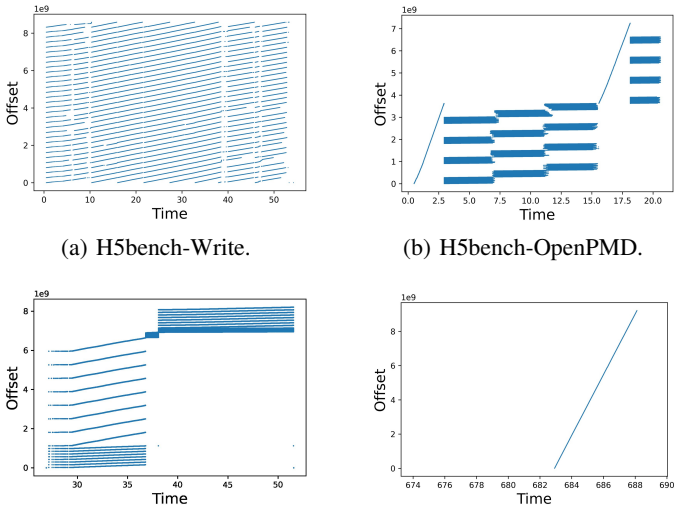
A. Experimental setup

For the evaluation, we employ Perlmutter supercomputer at NERSC [39]. The system comprises 3,072 CPU nodes and 1,792 GPU nodes, with each CPU node featuring 2 AMD EPYC 7763 (Milan) CPUs and 512 GB of main memory. As for the file system, Perlmutter operates with a Lustre file system, consisting of 16 metadata service nodes (MDSes) and 274 object storage servers (OSSes). Each OSS is equipped with 12 NVMe SSDs as object storage targets (OSTs). The Lustre file system of Perlmutter sets the default striping configuration to a stripe count of 1 and a stripe size of 1MB for all users.

B. Benchmarks

In terms of benchmark, we evaluate A2FL using four HPC benchmarks from two benchmark suites, H5bench [31] and AMReX [32].

- **H5bench-Write** is part of the H5bench suite [31] and is based on the particle physics simulation application VPIC [40]. It is configured to use multiple processes to write a shared file with interleaved file offsets, resulting in a non-contiguous access pattern.



(a) H5bench-Write. (b) H5bench-OpenPMD.

(c) AMReX-HDF5. (d) AMReX-Nyx

Fig. 10: Sample Access Pattern Images of the Benchmarks.

TABLE I: Configurations for A2FL in Small-Scale Experiments.

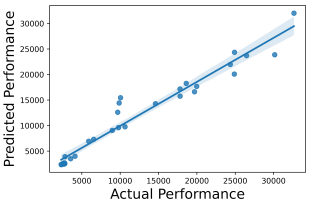
Benchmark	Dim	File Size (GB)	# Nodes	# Procs	Stripe Count
H5Bench-Write	1D	8, 32, 64	1, 4, 8	8, 32, 64	1, 8, 16, 32, 64
H5bench-OpenPMD	3D				
AMReX-HDF5	1-3D				
AMReX-Nyx	1D				

- **H5bench-OpenPMD** is part of the H5bench suite. It is a metadata-intensive benchmark that generates highly self-descriptive datasets. By default, OpenPMD primarily writes data in small sizes, mostly less than 1MB.
- **AMReX-HDF5** is an HDF5-based benchmark from the AMReX framework that simulates the I/O kernel of an application using adaptive mesh refinement. For AMReX-HDF5, we use a grid file to read the locations of grids relative to the entire problem domain, and we write a particle file as output. This results in the access pattern of the benchmark showing non-contiguous and composite I/O behaviors.
- **AMReX-Nyx** is a cosmological simulation code based on the AMReX framework. It is configured to write one or more plot files using the file-per-process method and sequential writes.

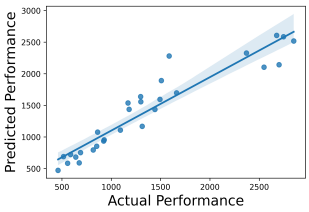
All these benchmarks are based on real HPC applications with diverse characteristics. Figure 10 displays access pattern graph images of these benchmarks, revealing distinct access patterns among them. This allows us to evaluate A2FL in a realistic HPC system.

C. A2FL

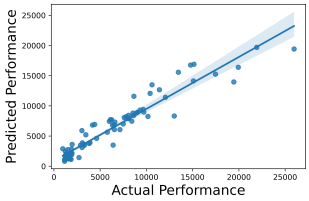
We first conduct small-scale experiments to evaluate the effectiveness of A2FL. To achieve this, we initially run benchmarks with various configurations. Table I shows the application and striping configurations for the training datasets. In the small-scale experiments, we conduct a strong scaling



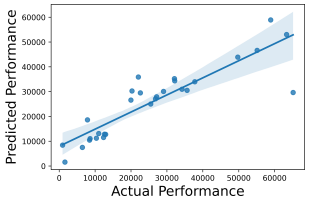
(a) H5bench-Write (R^2: 0.940, ExtraTrees, With Hue).



(b) H5bench-OpenPMD (R^2: 0.885, HistGradientBoosting, Without Hue).

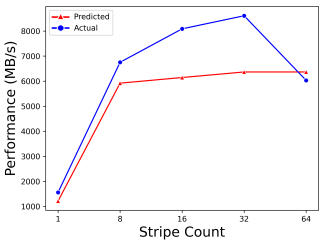


(c) AMReX-HDF5 (R^2: 0.915, XGBoost, Without Hue).

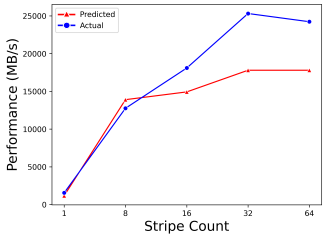


(d) AMReX-Nyx (R^2: 0.777, ExtraTrees, Without Hue).

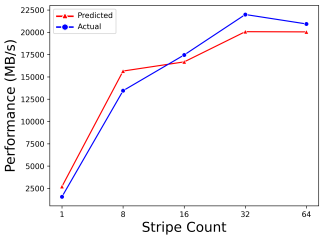
Fig. 11: Predicted and Actual Performance of the Benchmarks.



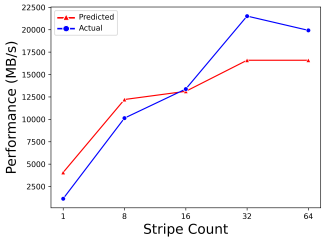
(a) 32GB 2D Particle File with 4 Nodes and 32 Processes.



(b) 64GB 2D Particle File: 8 Nodes and 32 Processes.



(c) 64GB 1D Particle File: 8 Nodes and 32 Processes.



(d) 64GB 3D Particle File: 8 Nodes and 32 Processes.

Fig. 12: Predicted and Actual Performance of AMReX-HDF5 with Varying Striping Configurations.

test with various file sizes. For I/O performance models of H5Bench-Write, H5bench-OpenPMD, AMReX-HDF5, and AMReX-Nyx, we use datasets with sizes of 150, 420, 450, and 150, respectively. To train the model for each benchmark, we split the data into training and testing using an 8:2 ratio.

After training, we assess the prediction model of A2FL, which is used to determine the optimal striping configuration. As explained in Section III-D, an accurate model is critical for determining the optimal file layout for the application since we choose the striping configuration based on the predicted performance. Thus, to evaluate A2FL, it is important to first evaluate the accuracy of the model.

Figure 11 shows the predicted performance using the I/O performance models of the benchmarks. The x-axis denotes the predicted performance and the y-axis denotes the actual performance from the experiment. Additionally, the caption includes information about the benchmark, such as the R^2 value for the model, the regression model used, and whether the image is created with or without hue. As shown in the figure, the models for H5bench-Write and AMReX-HDF5 have R^2 values of 0.940 and 0.915, respectively. Additionally, many data points are close to the linear diagonal lines, which denote perfect predictions. This is because they are both I/O-intensive and write most data in large blocks larger than 1MB. Thus, increasing the stripe count increases the performance as data blocks are written to multiple storage servers faster. This shortens the interval between the data points, which represent each request, making the image distinguishable across a range from a low stripe count to a high stripe count. Therefore, the trained model can model performance with various striping configurations, resulting in a higher R^2 value.

In contrast, the models for H5bench-OpenPMD and AMReX-Nyx have lower R^2 values of 0.885 and 0.777, respectively, along with fewer predicted values close to the linear diagonal lines. This is because H5bench-OpenPMD performs I/O operations with small sizes (e.g., 100B to 1KB). Moreover, in the case of AMReX-Nyx, each process writes a single file sequentially and independently, as shown in Figure 10d, which demonstrates a simple access pattern. Thus, changing the striping configuration for these access patterns does not alter the access pattern images. Thus, it is difficult for the model to differentiate one data point from another using similar images.

While overall performance prediction is important for evaluating the model, A2FL needs to predict the performance of a single application with a single access pattern while varying striping configurations. This is necessary to predict the trends of performance values that change based on different striping configurations and choose the optimal configuration. Figure 12 displays the predicted performance (Red Line) compared to the actual performance (Blue Line) using AMReX-HDF5 with various configurations. The x-axis represents the stripe counts ranging from 1 to 64, and the y-axis denotes the performance. As shown in Figure 12d, the results of I/O performance prediction can be accurate to a difference of 257 MB/s at a stripe count of 16. Conversely, it can be inaccurate to a difference of 7.5 GB/s at a stripe count of 32, as shown in Figure 12b. However, despite the differences between predicted and actual performances, the models can still determine the stripe count that maximizes performance. Thus, these results demonstrate that A2FL can select the optimal file layout that improves the performance the most for each access pattern in applications.

Finally, we evaluate the I/O performance improvement by adjusting the file layout based on the prediction model of A2FL. We perform evaluations for the configurations listed in Table I. Figure 13 showcases the performance of benchmarks using both the default and A2FL file layout. In the figure, "Default" represents the I/O performance during the first step

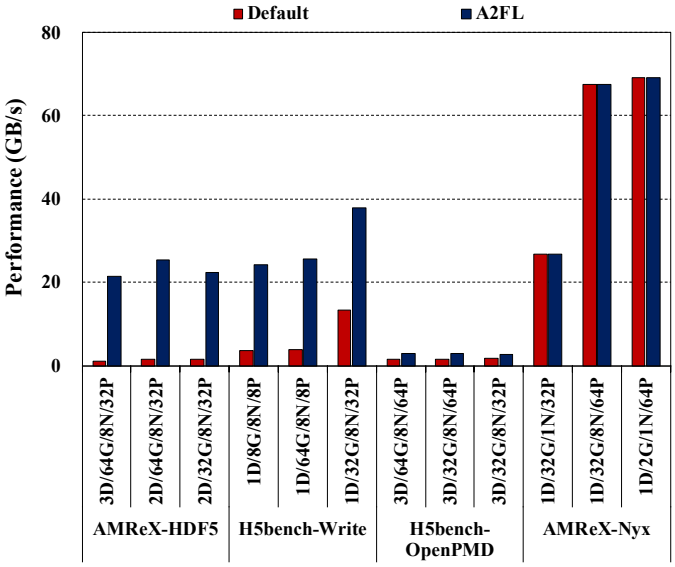
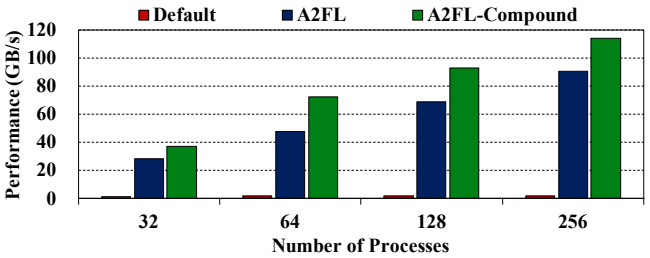


Fig. 13: Performance of A2FL in Small-Scale Experiments.

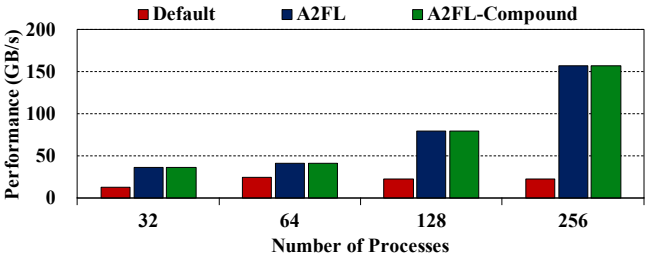
of the application with the default striping configuration which is stripe count of 1 and size of 1MB. We compare A2FL with the default since the majority of applications perform I/Os with the default striping configuration, according to the previous study [41]. The x-axis represents the benchmark type and its application configuration, including dimensions (i.e., D), file size (i.e., G), the number of nodes (i.e., N), and the number of processes (i.e., P). For example, "3D/64G/8N/32P" indicates an execution of a benchmark that writes a 64GB file in 3D using 8 nodes and 32 processes. For all benchmarks, we present the performance of the three configurations that lead to the largest performance improvements. As shown in the figure, A2FL improves the I/O performance of AMReX-HDF5 and H5bench-Write by up to 18.9× and 6.6×, respectively, compared with the default. In contrast, the I/O performance of H5bench-OpenPMD increases by up to 1.9× when using A2FL. This is because H5bench-OpenPMD is highly metadata-intensive and writes data in very small requests. Therefore, changing the file layout by distributing data among storage servers or sending data in larger data blocks does not have much impact on performance. Moreover, the performance of AMReX-Nyx does not improve even with the suggested file layout by A2FL. This is because it writes files on a per-process basis (file-per-process), and each process operates independently on its file. Increasing the stripe count does not provide significant performance benefits and leads to unnecessary complexity. Even though the performance does not improve, A2FL keeps using the default striping option as the I/O performance model expects it as the best stripe configuration for optimizing the file layout. Therefore, A2FL can successfully learn how each application performs depending on its various access patterns and modify file layout to improve the performance.

TABLE II: Configurations for A2FL-Compound in Small-Scale Experiments.

Benchmark	Dim	File Size (GB)	# Nodes(# Procs)
H5Bench-Write	1D	64, 128, 256, 512	8(32), 16(64), 32(128), 64(256)
AMReX-HDF5	2D	32, 64, 128, 256	8(32), 16(64), 32(128), 64(256)



(a) AMReX-HDF5.



(b) H5bench-Write.

Fig. 14: Performance of A2FL-Compound in Small-Scale Experiments.

D. A2FL-Compound

Although selecting an optimal striping configuration can enhance the I/O performance, we can further improve the performance by employing multiple striping configurations within a single file using A2FL-Compound. For the evaluation of A2FL-Compound, we use H5bench-Write and AMReX-HDF5 with configurations identical to those listed in Table II. For the prediction model, both A2FL and A2FL-Compound use the trained model from the previous evaluation. In the case of A2FL-Compound, we divide a single access pattern log into multiple components, each with an access pattern of 8GB file components.

Figure 14 shows the performance of AMReX-HDF5 and H5bench-Write with default, A2FL, and A2FL-Compound. As shown in Figure 14a, the I/O performance of AMReX-HDF5 with A2FL is higher than that of the default configuration. With 256 processes, the performance is improved by up to 52.3×. By using A2FL-Compound, the performance is further enhanced, reaching up to 65.9× that of the default configuration on 256 processes. This is because AMReX-HDF5 exhibits non-contiguous and complex I/O behavior, and the fine-grained file layout adjustment of A2FL-Compound can improve performance compared to a single striping configuration.

In contrast, H5bench-Write exhibits a lower degree of performance improvement compared to AMReX-HDF5. As

TABLE III: Striping Configurations of File Components for PFL Settings.

PFL 1		PFL 2	
File Component	Stripe Count/Size	File Component	Stripe Count/Size
[0GB, 8GB)	1 / 1MB	[0MB, 4MB)	1 / 1MB
[8GB, 32GB)	8 / 1MB	[4MB, 512MB)	2 / 1MB
[32GB, 64GB)	16 / 1MB	[512MB, 1GB)	4 / 1MB
[64GB, 256GB)	32 / 1MB	[1GB, 10GB)	8 / 1MB
[256GB, 512GB)	64 / 1MB	[10GB, 100GB)	16 / 1MB
[512GB, EOF)	128 / 1MB	[100GB, EOF]	32 / 1MB

TABLE IV: Configurations for `A2FL-Compound` Running H5Bench-Write in Large-Scale Experiments.

# Procs	File Size (GB)	Stripe Count	Stripe Size (MB)
1,024	128	1, 8, 16, 32, 64, 128	1, 2, 16, 32
2,048	256		
4,096	512		
8,192	1,024		
16,384	2,048		

shown in Figure 14b, the I/O performance of H5bench-Write can be improved by up to 7.0× compared to the default configuration However, `A2FL-Compound` does not lead to any additional improvement compared with `A2FL`. This is because H5bench-Write is a highly I/O-intensive application, and all processes efficiently write data in an asynchronous manner. Therefore, performance remains consistent throughout writing a file, and the optimal striping configuration remains unchanged, eliminating the need for a compound file layout from `A2FL-Compound`.

In addition to small-scale experiments, we validate `A2FL` and `A2FL-Compound` in large-scale experiments and examine their ability to optimize file layout with the extreme scalability of large HPC applications. Moreover, to evaluate the effect of analyzing the access pattern of the application for optimizing the striping configurations in different components of the file, we also compare `A2FL-Compound` with PFL. Since Perlmutter does not provide a default PFL setting, we use two different PFL settings: 1) manually configured PFL and 2) PFL from another large-scale production HPC system. Table III shows the file components and corresponding striping configurations of two PFL settings. We use H5bench-Write, which exhibits the highest scalability among the benchmarks used in this paper. Table IV shows the configurations for the large-scale experiments using H5bench-Write. We use 64 processes per node to run the benchmark and conduct a weak scaling test in which file size increases as the number of processes increases. For the evaluation of a large-scale experiment, we use the dataset with a size of 150 to build a new model of H5bench-Write different from small-scale experiments.

Figure 15 shows the performance improvement achieved by `A2FL` and `A2FL-Compound` in the large-scale experiment. In the figure, "PFL 1" and "PFL 2" represent the performances with two different PFL settings shown in Table III. As shown in the figure, `A2FL-Compound` improves the file

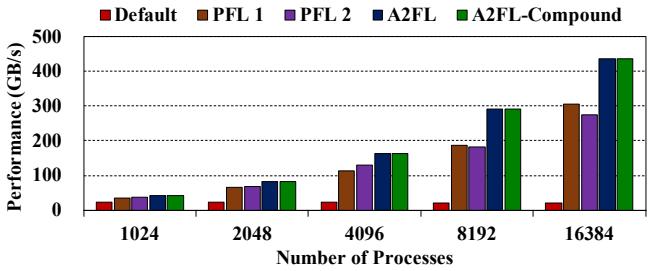


Fig. 15: Performance of `A2FL-Compound` in Large-Scale H5Bench-Write Experiments.

layout of H5bench-Write with very large numbers of nodes and processes. With 16,384 processes, `A2FL-Compound` achieves 436 GB/s, increasing the I/O performance by up to 21.8×, compared with that of the default. Moreover, `A2FL-Compound` outperforms PFL 1 and PFL 2 by 1.43× and 1.6×, respectively. Similar to the small-scale experiment, there is no performance difference between `A2FL`, and `A2FL-Compound` because H5bench-Write is an I/O-intensive application with a straightforward access pattern. Therefore, the experimental results in the large-scale experiments demonstrate that `A2FL-Compound` can successfully model the performance of applications in a large-scale production HPC environment and autonomously adjust the file layout to maximize performance in a fine-grained manner.

E. Overheads

To evaluate the overheads of `A2FL`, we measure the costs of creating images, I/O performance model training, and prediction.

1) Creating Images: For evaluating the overhead of creating images, we measure the time for creating images out of access pattern logs. To create each access pattern image of a 64GB file for H5bench-Write, H5bench-OpenPMD, AMReX-HDF5, and AMReX-Nyx, it requires up to 1.51, 2.09, 0.6, and 1.01 seconds, respectively. `A2FL` needs a group of images that can be used to train the model. Since this step is performed offline in batch, it does not affect the performance of the application during execution. In addition, `A2FL` only needs a single image out of an access pattern log to predict optimized striping configuration for the next step during the application execution. This short period of time can be overlapped with computation or simulation steps during the application executions. Therefore, the overhead of creating images is negligible.

2) Model Training: For evaluating the overhead of model training, we measure the time for training the I/O performance model. The training of each model for H5bench-Write, H5bench-OpenPMD, AMReX-HDF5, and AMReX-Nyx in the small-scale experiment takes 22.61, 59.49, 59.64, and 21.02 seconds, respectively. In the large-scale experiment, the training of the model for H5bench-Write takes up to 30.35 seconds. These training steps can be performed offline regardless of the application execution. Thus, it does not affect the performance of the application.

3) Prediction: For evaluating the overhead of prediction, we measure the time for predicting optimized striping configuration during application execution. The prediction of each model for H5bench-Write, H5bench-OpenPMD, AMReX-HDF5, and AMReX-Nyx in small-scale experiments requires 0.12, 0.26, 0.29, and 0.11 seconds, respectively. In the large-scale experiment, the prediction of the model for H5bench-Write takes up to 0.31 seconds. Although prediction is performed during application execution to set an optimized file layout for I/O in the next step, it can be completed immediately. Therefore, the overall overheads of A2FL are considered minimal.

V. Related Works

A. ML-Based Modeling and Prediction of I/O Characteristics

There have been many previous studies that utilize I/O access patterns of HPC applications to predict I/O performance or behavior using ML-based modeling to enhance application performance and system utilization. Omnisc'IO [42] employs a grammar-based model to track I/O operations and retrieve metrics such as I/O time, offset, and request size in MPI-IO and POSIX I/O libraries. Using the model, it predicts future I/O operations for optimization purposes like prefetching, caching, and scheduling. Kim et al. [15] propose an ML-based prediction framework that utilizes integrated system and application-level logs to predict the I/O performance of HPC applications. Schmidt et al. [43] use artificial neural networks to learn and predict file access times in the Lustre file system. Among various neural networks, they employ an ANN-based model to achieve the lowest prediction error for analyzing the performance of the parallel file system. Behzad et al. [44] design an I/O performance model that predicts I/O costs based on tunable I/O parameters to identify configurations that enhance the I/O performance of applications. Our study aligns with these prior studies by utilizing I/O-related information from applications and constructing an ML-based model to predict I/O performance. In contrast, we concentrate on file access patterns, such as accessed file offsets and timestamps, to construct an I/O performance model and utilize it in real-time to recommend an optimized file layout on the parallel file system.

B. Adjusting Application and File System Configuration

There have been many studies that suggest and adjust configurations for applications and file systems to improve the performance. Tipu et al. [45] employ artificial neural networks to forecast read and write bandwidths. By constructing two models for read and write operations, they identify distinct sets of parameters from MPI-IO and Lustre file systems to enhance the read and write bandwidths of the application. Yang et al. [46] propose AIOT, an I/O path optimization tool that provides parameter-tuning strategies to improve I/O performance and resource utilization. It utilizes the I/O behavior of applications and the status of the file system to optimize load-balancing and interference. Behzad et al. [16], [47] propose a framework that identifies optimal configuration parameters from HDF5, MPI-IO, and parallel file systems using genetic algorithms. This framework reduces parameter spaces and suggests configurations by collecting and learning parameter values over multiple runs. CAPE [17] is a parameter tuning system that monitors the throughput of storage servers and adjusts parallel file system configurations, such as RPC congestion window size and I/O rate limit. It employs a Q-learning-based approach to perform real-time tuning, enabling continuous operation and dynamic configuration modifications. Similar to these studies, our work analyzes HPC I/O applications and determines optimized configurations to enhance I/O performance. However, our focus is on determining the optimal striping configuration (i.e., stripe count and size) in real-time in a fine-grained manner for the parallel file system, while also emphasizing autonomous adjustment without requiring user intervention.

VI. Conclusion

In this paper, we propose A2FL, an autonomous and adaptive file layout for large-scale distributed systems. A2FL's approach encompasses several key steps: (1) **I/O Access Pattern Tracing**: A2FL first dynamically traces the I/O access patterns of applications during runtime. (2) **Feature Extraction and Model Training**: Using the traced access pattern logs, A2FL generates graphical representations and extracts unique features of access patterns to train an I/O performance model. (3) **Performance Prediction**: Leveraging the trained model, A2FL predicts I/O performance, enabling it to identify the striping configuration that offers the maximum performance enhancement. (4) **Fine-Grained File Layout Optimization**: For more precise optimization, we introduce A2FL-Compound, which further enhances file layout by subdividing the file into smaller components and adjusting striping configurations individually. Our comprehensive evaluation results demonstrate that A2FL improves I/O performance by up to 65.9× compared to the default file layout using HPC benchmarks. We firmly believe that A2FL is a valuable tool for HPC users since it automatically analyzes the I/O access patterns of diverse applications to improve utilization of the parallel file system and boost I/O performance — all without necessitating any code modifications or user intervention.

VII. Acknowledgement

We thank the anonymous reviewers for their insightful comments and suggestions. This research used the resources of the National Energy Research Scientific Computing Center (NERSC). This work was supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-05CH11231, with program managers Hal Finkel and Margaret Lentz, National Research Foundation of Korea (NRF) (RS-2022-00166541, 2022R1A4A5034130, 2021R1F1A1063438), the Korea Institute of Science and Technology Information (P23008), BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, SNU) funded by NRF (4199990214639).

REFERENCES

[1] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki *et al.*, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001, 2009.

[2] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia *et al.*, "Scalable earthquake simulation on petascale supercomputers," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–20.

[3] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[4] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "Bdcats: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[5] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.

[6] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. Navaux, M. A. Dantas, and Y. Denneulin, "A checkpoint of research on parallel i/o for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–35, 2018.

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

[8] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable i/o forwarding framework for high-performance computing systems," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[9] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs-a temporary distributed file system for hpc applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 319–324.

[10] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2007.

[11] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[12] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, no. 19, 2002.

[13] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[14] J. Heichler, "An introduction to beegfs," *Introductionto BeeGFS by ThinkParQ. pdf*, 2014.

[15] S. Kim, A. Sim, K. Wu, S. Byna, Y. Son, and H. Eom, "Towards hpc i/o performance prediction through large-scale log analysis," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 77–88.

[16] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel i/o complexity with auto-tuning," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, 2013, pp. 1–12.

[17] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. Long, "Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2017, pp. 1–14.

[18] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[19] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A year in the life of a parallel file system," in *SC18: International conference for high performance computing, networking, storage and analysis*. IEEE, 2018, pp. 931–943.

[20] M. T. Henry de Frahan, J. S. Rood, M. S. Day, H. Sitaraman, S. Yellapantula, B. A. Perry, R. W. Grout, A. Almgren, W. Zhang, J. B. Bell *et al.*, "Pelec: An adaptive mesh refinement solver for compressible reacting flows," *The International Journal of High Performance Computing Applications*, vol. 37, no. 2, pp. 115–131, 2023.

[21] A. Almgren, M. B. Sazo, J. Bell, A. Harpole, M. Katz, J. Sexton, D. Willcox, W. Zhang, and M. Zingale, "Castro: A massively parallel compressible astrophysics simulation code," *Journal of Open Source Software*, vol. 5, no. 54, p. 2513, 2020. [Online]. Available: https://doi.org/10.21105/joss.02513

[22] 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)*. IEEE, 2016, pp. 9–17.

[23] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–8.

[24] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[25] M. Christ, N. Braun, J. Neuffer, and A. W. Kempa-Liehr, "Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package)," *Neurocomputing*, vol. 307, pp. 72–77, 2018.

[26] J. Faouzi and H. Janati, "pyts: A python package for time series classification," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 1720–1725, 2020.

[27] A.-A. Semenoglou, E. Spiliotis, and V. Assimakopoulos, "Image-based time series forecasting: A deep convolutional neural network approach," *Neural Networks*, vol. 157, pp. 39–53, 2023.

[28] C.-L. Yang, C.-Y. Yang, Z.-X. Chen, and N.-W. Lo, "Multivariate time series data transformation for convolutional neural network," in *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2019, pp. 188–192.

[29] N. Hatami, Y. Gavet, and J. Debayle, "Classification of time-series images using deep convolutional neural networks," in *Tenth international conference on machine vision (ICMV 2017)*, vol. 10696. SPIE, 2018, pp. 242–249.

[30] C.-L. Yang, Z.-X. Chen, and C.-Y. Yang, "Sensor classification using convolutional neural network by encoding multivariate time series as two-dimensional colored images," *Sensors*, vol. 20, no. 1, p. 168, 2019.

[31] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "h5bench: Hdf5 i/o kernel suite for exercising hpc i/o patterns," in *Proceedings of Cray User Group Meeting, CUG*, vol. 2021, 2021.

[32] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves *et al.*, "Amrex: a framework for block-structured adaptive mesh refinement," *The Journal of Open Source Software*, vol. 4, no. 37, p. 1370, 2019.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[34] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[35] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[36] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[37] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou *et al.*, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[38] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[39] "Nersc perlmutter," https://www.nersc.gov/systems/perlmutter/, accessed: 2023-06-30.

[40] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin *et al.*, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *SC'12: Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis. IEEE, 2012, pp. 1–12.

[41] S. Kim, A. Sim, K. Wu, S. Byna, T. Wang, Y. Son, and H. Eom, "Dca-io: A dynamic i/o control scheme for parallel and distributed file systems," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 351–360.

[42] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Using formal grammars to predict i/o behaviors in hpc: The omnisc'io approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2435–2449, 2015.

[43] J. F. Schmidt and J. M. Kunkel, "Predicting i/o performance in hpc using artificial neural networks," *Supercomputing Frontiers and Innovations*, vol. 3, no. 3, pp. 19–33, 2016.

[44] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel i/o autotuning with performance modeling," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 253–256.

[45] A. J. S. Tipu, P. Ó. Conbhuí, and E. Howley, "Artificial neural networks based predictions towards the auto-tuning and optimization of parallel io bandwidth in hpc system," *Cluster Computing*, pp. 1–20, 2022.

[46] B. Yang, Y. Zou, W. Liu, and W. Xue, "An end-to-end and adaptive i/o optimization tool for modern hpc storage systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1294–1304.

[47] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 4, pp. 1–27, 2019.