Predicting and Comparing the Performance of Array Management Libraries

Donghe Kang[1], Oliver Rübel[2], Suren Byna[2], Spyros Blanas[1]
The Ohio State University[1] , Lawrence Berkeley National Laboratory[2]
{kang.1002, blanas.2}@osu.edu, {oruebel, sbyna}@lbl.gov

***Abstract*—Many applications are increasingly becoming I/O-bound. To improve scalability, analytical models of parallel I/O performance are often consulted to determine possible I/O optimizations. However, I/O performance modeling has predominantly focused on applications that directly issue I/O requests to a parallel file system or a local storage device. These I/O models are not directly usable by applications that access data through standardized I/O libraries, such as HDF5, FITS, and NetCDF, because a single I/O request to an object can trigger a cascade of I/O operations to different storage blocks. The I/O performance characteristics of applications that rely on these libraries is a complex function of the underlying data storage model, user-configurable parameters and object-level access patterns. As a consequence, I/O optimization is predominantly an ad-hoc process that is performed by application developers, who are often domain scientists with limited desire to delve into nuances of the storage hierarchy of modern computers.**

This paper presents an analytical cost model to predict the end-to-end execution time of applications that perform I/O through established array management libraries. The paper focuses on the HDF5 and Zarr array libraries, as examples of I/O libraries with radically different storage models: HDF5 stores every object in one file, while Zarr creates multiple files to store different objects. We find that accessing array objects via these I/O libraries introduces new overheads and optimizations. Specifically, in addition to I/O time, it is crucial to model the cost of transforming data to a particular storage layout (memory copy cost), as well as model the benefit of accessing a software cache. We evaluate the model on real applications that process observations (neuroscience) and simulation results (plasma physics). The evaluation on three HPC clusters reveals that I/O accounts for as little as 10% of the execution time in some cases, and hence models that only focus on I/O performance cannot accurately capture the performance of applications that use standard array storage libraries. In parallel experiments, our model correctly predicts the fastest storage library between HDF5 and Zarr 94% of the time, in contrast with 70% of the time for a cutting-edge I/O model.

I. Introduction

Many scientific datasets are naturally represented as multi-dimensional arrays. To access these scientific datasets, applications often use array management libraries that support array-centric APIs. Different array management libraries use different data storage representations. For example, one such array management library is HDF5, which has been in active use for two decades. HDF5 stores arrays, metadata, and attributes, in one file with an inner structure that is opaque to the application. Another library that is becoming prominent is Zarr. Zarr stores a chunk, a fixed-size partition of an array, in a separate file, such that storing a large array requires many files which is challenging for parallel file systems. HDF5 and Zarr represent the two extremes of array storage mechanisms.

Although I/O performance modeling has long been a topic of active research, performing I/O using an array management library poses additional challenges. The performance of an array management library is impacted by various system-specific or user-configured factors, such as the logical array shape, the physical storage layout and the application access pattern to arrays. The impact of these factors varies in different array management libraries. For example, the same dataset is faster to read from HDF5 than Zarr when the array is partitioned into many chunks, but HDF5 becomes slower than Zarr when multiple processes concurrently write to the array due to the I/O contention. Existing I/O models [1]–[3] are only focusing on the storage device, but are not accurate when the device I/O time is only a small fraction of the total time it takes to return data to the application. Thus, it is not trivial for users (often domain scientists) to select the most efficient array storage configuration for their application.

Towards addressing the challenge, this paper introduces a performance model that estimates the I/O performance for array management libraries over parallel file systems. To the best of our knowledge, no prior work has systematically considered how the choice of an array management library and its configuration impacts I/O performance. In addition to choosing the most efficient array management library for a given access pattern, an accurate I/O performance model also guides users to configure the logical and physical storage layout for better performance. The model presented in the paper is based on the cost of data movement to memory and between compute nodes and storage devices. The factors in the model are the number of accessed data chunks, the data size, the number of I/O requests, number of used data servers (devices) in the parallel file system, and number of parallel processes. The model derives these factors from the array storage layout and access pattern. The layout and pattern can be easily intercepted from the applications and array management libraries using existing I/O characterization tools such as Darshan or an HDF5 VOL driver.

We evaluate the cost model on two parallel file systems, Lustre and GPFS, in three HPC clusters that represent facilities of different sizes and configurations. The access patterns are derived from two scientific applications, a neuroscience application and an accelerator modeling simulation code. The model correctly predicts the relative performance of HDF5

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00097

IEEE computer society

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

and Zarr in 97 out of 111 evaluation experiments, with a root mean square error (RMSE) of 0.29 for the model.

The main contributions of this paper are:

1) We build an end-to-end cost model to compare the performance of two popular array management libraries, HDF5 and Zarr. We decompose the cost into the memory copy cost, software cache read cost and data server I/O cost.
2) We propose a model to predict the parallel I/O cost for both HDF5 and Zarr that captures the contention for the metadata server and the data servers in the underlying parallel file systems.
3) We use the linear regression to train the model and evaluate the model based on two scientific applications. The evaluation is conducted on three HPC clusters with different sizes and parallel file systems, to better reflect the diversity of HPC infrastructure. The memory copy and software cache read operations take as much as 90% and 87% of the execution time, which corroborates that exclusively focusing on the data server I/O cost is insufficient. In parallel experiments, our model correctly predicts the fastest library between HDF5 and Zarr 94% of the time versus 70% of the time for a cutting-edge I/O model.

The rest of this paper is organized as follows. Section II briefly introduces parallel file systems and the two array management libraries compared and modeled in this study (HDF5 and Zarr). We then present the cost prediction model in Section III. We train and evaluate the model in Section IV and V. We discuss the related work to the I/O and memory operation cost prediction and then conclude.

II. Background

A. Parallel file systems

Parallel file systems, such as Lustre and GPFS are widely used on HPC clusters. These file systems have architectural similarities in terms of having separate metadata and data servers and transferring data to clients through the network. A file is partitioned into stripes and each stripe is stored in a single data server. The metadata server stores the location of each stripe for a file. We refer to the mapping between stripes and data servers as the *file layout*. There are two parameters that control the file layout: (1) the *stripe size*, or the size of a stripe in bytes, and (2) the *server count*, or the number of data servers that will be used. In GPFS, all files in the same file system use the same stripe size and server count, whereas in Lustre two files in the same file system can have different stripe sizes and server counts. Accessing many files in a parallel file system is inefficient due to the frequent communication with metadata servers and the lack of prefetching in data servers.

B. Array management libraries

Array management libraries, such as HDF5, Zarr, TileDB, FITS, and NetCDF, expose array-centric APIs. Scientific applications access arrays through these libraries. An array is partitioned into fixed-shape *chunks*. Chunks are the I/O units between the library and the underlying file system. Applications access an array through hyperslab selections. A hyperslab is a logically contiguous region of array cells or a regular pattern of cells. Section III-A describes the data access procedure after a hyperslab selection. This paper focuses on the HDF5 and Zarr libraries, which use two different file layouts: All chunks of an HDF5 array are stored in one file, whereas each chunk of a Zarr array is stored as a separate file.

1) HDF5: The Hierarchical Data Format 5 (HDF5) is an array management library which stores multiple arrays and their metadata in one file. Chunks in an array are stored in the file based on row-major order. HDF5 locates chunks in a file through the *chunk index*. The chunk index records the offsets and lengths of chunks. The index is partitioned into index blocks, and each index block indexes a fixed number of chunks. An index block is stored near the chunks it indexes.

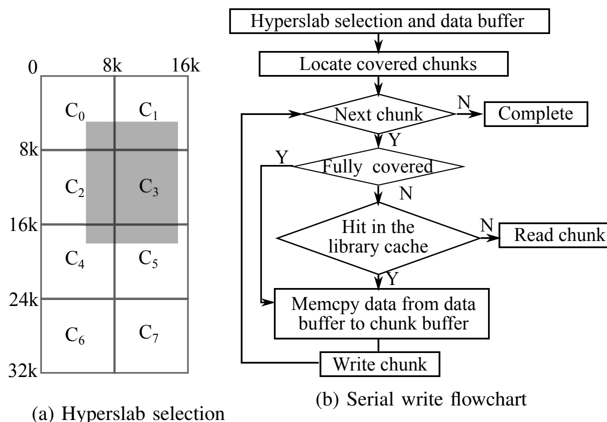
2) Zarr: The physical representation of a Zarr array is a folder. The folder contains one metadata file and one or more chunk files. The metadata file stores the array shape, chunk shape and data type. The metadata file is accessed before opening or creating an array. A chunk file stores the raw data of a chunk. The chunk file name encodes the logical position of the chunk in the array. Locating relevant chunks requires converting index offsets to chunk file names at query time.

III. Modeling array management libraries

This section builds a cost model to predict the performance of HDF5 and Zarr. Section III-A first introduces the common I/O path that both HDF5 and Zarr use to read and write arrays through hyperslab selections. Based on this I/O path, Sections III-B and III-C introduce the serial and the parallel cost model, respectively. We discuss the model applicability, usage, and limitations in Section III-D.

A. Array management library behavior

The HDF5 and Zarr array management libraries have a common interface to read and write arrays through hyperslab selections. Figure 1a shows an example that accesses a $32,000 \times 16,000$ array with chunk shape $8,000 \times 8,000$



(a) Hyperslab selection
(b) Serial write flowchart

Fig. 1: Hyperslab selection and write flowchart

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

through a hyperslab selection, denoted as the grey rectangle. The user inputs a hyperslab selection and a *data buffer*. The data buffer holds the cell values read from or written to the array. The data buffer is serialized on row-major order.

HDF5 and Zarr have a common I/O path in serial programs. HDF5 and Zarr first identify the chunks covered by the selected hyperslab, e.g. the chunks C_0 to C_5 in Figure 1a. HDF5 and Zarr use different mechanisms to locate chunks. HDF5 locates chunks via the chunk index, while Zarr locates chunks based on the chunk file names. These chunks are then read or written sequentially. The I/O unit is an entire chunk. The I/O path in the parallel file system has three steps: opening or creating a file, transferring data through the network, and extracting or flushing data from or to storage devices in data servers. We denote the time of the three steps as the *metadata time*, *network time*, and *process time*. Both libraries and the file system use caches, called *library cache* and *software cache* respectively, to hold recently accessed data.

In the read path, when a chunk is found in the library cache, the selected cells in the chunk are copied from the library cache to the data buffer. When the chunk is not in the library cache, HDF5 and Zarr use POSIX I/O to retrieve the chunk from the file system. If the chunk is in the software cache, HDF5 and Zarr fetch the chunk from the software cache to a buffer, called *chunk buffer*, and then copy the selected cells to the data buffer. Otherwise, HDF5 and Zarr read the chunk from data servers to the chunk buffer and copy selected cells.

In the write path, when a chunk is fully covered by the hyperslab, HDF5 and Zarr copy the chunk from the data buffer to the chunk buffer and then write the chunk. When a chunk is not fully covered, HDF5 and Zarr read the chunk into the chunk buffer, copy the data buffer to the chunk buffer, and write the chunk. Figure 1b shows the flowchart to write an array.

The I/O path is different when multiple processes concurrently access an array. The parallel I/O path of Zarr is same as the serial I/O path. In the parallel I/O path, HDF5 does not use the library cache. For each chunk covered by the hyperslab, HDF5 uses one or more MPI I/O operations to access the selected cells in the chunk. An I/O operation accesses the cells which are contiguous in both the data buffer and the serialized chunk. A chunk and the buffer are serialized on row-major order. Parallel HDF5 and Zarr do not guarantee sequential consistency: the results of hyperslab selections concurrently writing and reading an array are undefined [4].

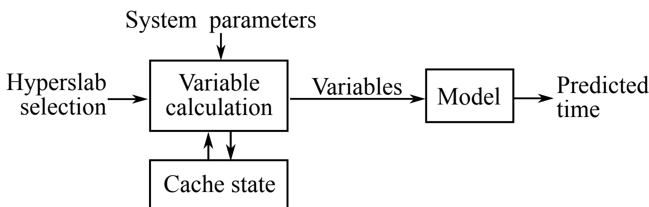


Fig. 2: Prediction procedure

TABLE I: Variables and descriptions

Variables	Description
t	Predicted execution time
t_{memory}	Predicted memory copy time
t_{rdisk}	Predicted data server read time
t_{wdisk}	Predicted data server write time
t_{cache}	Predicted software cache read time
s_{chunk}	Chunk size in bytes
n_{memcpy}	Number of memory copy operations
s_{memcpy}	Memory copied data size
n_{rdisk}	Number of chunks read from data servers
n_{rseek}	Number of seeks to read chunks in HDF5
n_{rindex}	Number of index blocks read in HDF5
n_{wdisk}	Number of written chunks
n_{wseek}	Number of seeks to write chunks in HDF5
n_{windex}	Number of index blocks written in HDF5
n_{cache}	Number of chunks read from software cache

B. Serial cost model

The serial cost model predicts the end-to-end execution time of reading or writing an array through a sequence of hyperslab selections in the serial HDF5 and Zarr. Figure 2 shows the procedure to use the model. Users input the hyperslab selections and system parameters, such as the sizes of the library cache and the software cache, to compute the variables in the model. The variable calculation maintains the status of caches. The model then predicts the execution time based on these variables.

Existing analytical I/O cost models [1], [2], [5] only predict the time to access data servers. In comparison, we find that other operations in the I/O path, specifically the memory copy operation and software cache accesses, take considerable time. We thus divide the end-to-end execution time into three components: **memory copy time**, **data server access time** and **cache read time**. The cache read time refers to the time reading chunks from the software cache. Table I shows the notation. The predicted end-to-end execution time, t, is the sum of these components:

$$t = t_{memory} + t_{rdisk} + t_{wdisk} + t_{cache} \tag{1}$$

The memory copy time in HDF5 and Zarr is proportional to the number of memory copy operations n_{memcpy}, and the size of copied data s_{memcpy}. We assume that a memory copy operation has some constant startup cost plus a cost that is proportional to the size of the data that are copied during the operation. Therefore, the memory copy time is modeled as:

$$t_{memcpy} \propto n_{memcpy} + s_{memcpy} \tag{2}$$

For the cache read time, the model assumes that there is some constant start-up cost plus a cost that is proportional to the read size. Hence, the cache read time is modeled as a linear function of the the number of chunks n_{cache}, and the size of the data read from the cache $n_{cache} \times s_{chunk}$:

$$t_{cache} \propto n_{cache} + n_{cache} \times s_{chunk} \tag{3}$$

When the requested data are not in the cache, a read operation will need to access the data servers. For every accessed

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

chunk, Zarr must first communicate with the metadata server before opening the relevant chunk file. The model assumes that the metadata time of a read operation is constant, whereas the network time and the process time have constant start-up cost and are proportional to the read size. The data server read time in Zarr is thus modeled as:

$$t^{Zarr}_{rdisk} \propto n_{rdisk} + n_{rdisk} \times s_{chunk} \quad (4)$$

HDF5 reads multiple chunks from a single file, hence HDF5 will only open the file once per read operation. Hence, we model the data server read time in HDF5 as the sum of the network time and the process time. The network time is proportional to three variables: (1) the number of chunks, (2) the size of each chunk, and (3) the number of index blocks that need to be read. The process time of random reads in an HDF5 file is higher than sequential reads. The model uses the number of *seek* operations to capture the overhead of random reads. A seek operation happens when the next chunk to be read does not immediately follow the last chunk that was read from the HDF5 file. The process time is thus proportional to the number of seek operations, the number and size of read chunks and the number of read index blocks. The data server read time in HDF5 is thus modeled as:

$$t^{HDF5}_{rdisk} \propto n_{rseek} + n_{rdisk} + n_{rdisk} \times s_{chunk} + n_{rindex} \quad (5)$$

The data server write time is predicted similarly to the data server read time. The write time in Zarr, t^{Zarr}_{wdisk}, is linearly proportional to the number of written chunks and the written data size, as shown in Formula 6. The write time in HDF5, t^{HDF5}_{wdisk}, is linearly proportional to the numbers of seeks, written index blocks and written chunks and the written data size as shown in Formula 7.

$$t^{Zarr}_{wdisk} \propto n_{wdisk} + n_{wdisk} \times s_{chunk} \quad (6)$$

$$t^{HDF5}_{wdisk} \propto n_{wseek} + n_{wdisk} + n_{wdisk} \times s_{chunk} + n_{windex} \quad (7)$$

Careful readers may have noticed that the HDF5 model adds variables to the Zarr model. However, this does not necessarily mean that the read time of HDF5 is larger than the read time in Zarr, as the HDF5 and Zarr formulas use different coefficients for the same variable.

Variable Calculation. This subsection shows how to compute the variables in Table I. The inputs of the variable calculation are the hyperslab selection and system parameters. A hyperslab selection is the offset and size of the hyperslab in each dimension. The system parameters are the sizes of the library cache and the software cache. Computing the HDF5 model requires one more system parameter, which is the number of chunks indexed by an HDF5 index block, called the index block size.

Users need to input the hyperslab selection and the system parameters to the variable calculation. Users who do not know these inputs can run the application and collect this information systematically through the HDF5 virtual object layer (*VOL*). The *VOL* is a new abstraction in HDF5 that allows intercepting and injecting I/O operations without modifying applications. Users can implement a *VOL* plugin to log the hyperslab selections, the configured HDF5 cache size and index block size, and the free memory size. Prior works [6], [7] have used this mechanism to log the access pattern.

Algorithm 1: Computing variables for HDF5 read

input : h, a hyperslab
s_{index}, the number of chunks indexed by an index block
output: n_{rdisk}, n_{rseek}, n_{rindex} and n_{cache}

```
1  id_last ← -1;
2  Initialize output variables with 0;
3  foreach chunk c covered by h do
4      id ← the row-major order of c in the array;
5      if id ∉ C_lib ∧ id ∈ C_u then
6          n_cache ← n_cache + 1;
7      else if id ∉ C_lib ∧ id ∉ C_u then
8          n_rdisk ← n_rdisk + 1;
9          if id ≠ id_last + 1 then
10             n_rseek ← n_rseek + 1;
11         index ← ⌊ id / s_index ⌋;
12         if index ∉ C_index then
13             n_rindex ← n_rindex + 1;
14             Insert index to C_index;
15         Insert id to C_u;
16     Insert id to C_lib;
17     id_last ← id;
```

Calculating the input variables correctly depends on the state of the cache. For this reason, the model tracks the cache state separately for the library cache, the software cache and the accessed HDF5 index blocks. The contents of the library cache and the software cache are represented as two finite sets, denoted as C_{lib} and C_u, respectively. The sizes of the two sets are the capacity of the two caches. All the accessed HDF5 index blocks are tracked in the C_{index} set. The three sets only hold a unique ID for each chunk and index block, instead of the actual data.

Due to space limitations, we only show how to compute the variables for the HDF5 read operation. Algorithm 1 shows the variable calculation algorithm. The inputs are a hyperslab selection and the index block size. The algorithm first identifies all the chunks covered in the hyperslab. First, Algorithm 1 increments the number of cache read operations when a chunk is not in the library cache but is found in the software cache (lines 5-6). When a chunk is in neither cache, the number of chunks read from data servers, n_{rdisk}, is increased by 1. If the current chunk does not immediately follow the last chunk, the number of seeks is incremented (lines 9-10). If the index block has not been accessed before, the number of accessed index blocks is incremented (lines 11-13). Finally, the cache state is updated (lines 14-16) before processing the next chunk.

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

C. Parallel cost model

The parallel cost model predicts the execution time of HDF5 and Zarr when multiple processes independently and concurrently access an array. The parallel cost model focuses on the use case where (1) each chunk is accessed by only one process, (2) the stripe size is the chunk size, and (3) each process runs in a different node. Many scientific applications, such as GCRM and VORPAL [8], have this access pattern.

When multiple processes concurrently read or write to the parallel file system, the model predicts the I/O time by estimating resource contention. The parallel cost model assumes that (1) the metadata server sequentially processes the file open and creation requests, (2) the network is not the bottleneck, and (3) a data server sequentially processes the I/O requests it receives. The inputs to the parallel cost model are the hyperslab selections for each process and the number of accessed data servers.

We model the contention for the metadata server and the data server differently. The degree of contention for the metadata server is the number of processes that are accessing it. To model the contention for data servers, we track the number of processes concurrently accessing a data server, denoted as cf. Data servers have higher contention in HDF5 than Zarr when chunks are stored in a subset of all data servers. Consider the example shown in Figure 3 where three processes P_0, P_1, P_2 read sixteen array chunks $C_0, C_1, ..., C_{15}$ in a round-robin pattern, and the chunks in Zarr are stored in all n data servers while the HDF5 chunks are striped to two data servers S_i and S_j. Formula 8 computes the degree of contention cf with p processes and c data servers. The c variable in HDF5 is the server count, while c in Zarr is the total number of data servers.

$$cf = 1 + \frac{p-1}{c} \quad (8)$$

The parallel execution time of a process consists of the memory copy time, software cache read time, and data server access time. The parallel cost model predicts the software cache time similarly as the serial model, as processes run in separate nodes and access different chunks.

Formula 9 models the data server read time in HDF5, which consists of the network time $n_{rrequest} + s_{rrequest}$, and the process time $n_{rrequest} \times cf + s_{rrequest} \times cf$. The process time is multiplied by cf to account for contention in data servers. The data server write time is predicted similarly as the read time. The memory copy time does not need to be modeled

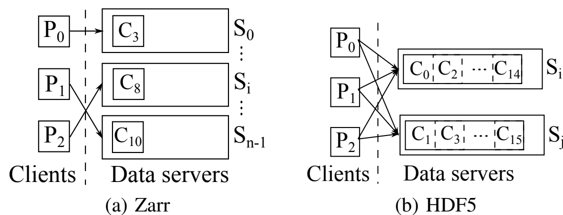


Fig. 3: Data layout and access pattern

TABLE II: Variables and descriptions of parallel cost model

Variables	Description
p	Numer of processes
c	Number of data servers accessed in the application
cf	Number of processes concurrently accessing a data server
$n_{rrequest}$	Number of data server read requests
$s_{rrequest}$	Read data size from data servers
$n_{wrequest}$	Number of data server write requests
$s_{wrequest}$	Written data size to data servers

because HDF5 does not use the chunk buffer in the parallel I/O path.

$$t_{rdisk}^{HDF5} \propto n_{rrequest} + s_{rrequest} + n_{rrequest} \times cf + s_{rrequest} \times cf \quad (9)$$

Formula 10 models the data read time in Zarr. The read time consists of the network time, process time, and metadata time. The metadata time $n_{rrequest} \times p$, is multiplied by p due to contention in the metadata server. The memory copy time of Zarr in the parallel cost model is predicted similarly as the serial cost model.

$$t_{rdisk}^{Zarr} \propto n_{rrequest} \times p + n_{rrequest} + s_{rrequest} + n_{rrequest} \times cf + s_{rrequest} \times cf \quad (10)$$

Variable Calculation. Given the hyperslab selections of each process, we compute the variables $n_{rrequest}$, $s_{rrequest}$, $n_{wrequest}$ and $s_{wrequest}$. The model only considers the access pattern where a chunk is only accessed by a single process.

With parallel I/O, Zarr works similarly as the serial version. The $n_{rrequest}$ and $n_{wrequest}$ of Zarr equal the n_{rdisk} and n_{wdisk} in Table I. Chunks are the I/O units in the parallel Zarr, such that the $s_{rrequest}$ and $s_{wrequest}$ are derived by multiplying the n_{rdisk} and n_{wdisk} with the chunk size s_{chunk}.

An MPI-I/O operation in parallel HDF5 directly reads or writes array cells which are contiguous in a chunk and the data buffer. For example, in Figure 1a, an MPI-I/O operation accesses the selected cells in each row of a chunk. We initialize the four parameters as 0. For each read chunk, we increase the $n_{rrequest}$ and $s_{rrequest}$ by the number of MPI-I/O operations and the data size of the selected cells in the chunk. The $n_{wrequest}$ and $s_{wrequest}$ are computed in the similar way.

D. Discussion

1) Generalizability: The cost model can be applied to different array management libraries. There are three common array storage mechanisms: storing everything in a file (S1), storing arrays in separate files (S2), and storing a chunk in a file (S3). The HDF5 and Zarr represent the two extremes, S1 and S3, respectively. The cost models in this paper can also be applied to arrays stored based on the S2 mechanism. The HDF5 cost model can predict the serial cost. Processes in a parallel application often access either an array or separate arrays. The HDF5 and Zarr parallel cost models apply to the two access patterns respectively due to the same file access patterns.

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

The model considers HPC clusters with a fast network to slow, shared storage. Many HPC clusters use Infiniband to connect nodes in the cluster and hard disks to store data in the parallel file system. The generalizability of the model to faster storage devices that may be locally-attached to compute nodes, such as NVM, is an open question.

2) Usage: It is challenging for users, who are often domain scientists, to identify efficient I/O configurations. The I/O performance is dependent on various factors, like the hardware status, file system, and array access pattern. Without a prediction model, users have to implement the I/O modules in applications on different array management libraries and manually tune I/O parameters. This procedure is cumbersome, error-prone, and heavily relies on a user's prior experience.

Together with suitable search heuristics, the model provides a systematic method to identify the most efficient array management library and I/O configurations for an application. The solution depends on the array access pattern which can be intercepted automatically. If a dataset is replicated in different file formats and file system configurations, the cost model can predict the execution time on different replicas.

3) Limitations: Although the model in this paper only captures linear effects, we acknowledge that recent advances in I/O technology may require introducing non-linearity in I/O performance modeling. More complex I/O models and training methods could more accurately predict the performance of each phase in the I/O path. Given that the end-to-end time is modeled as the sum of the time of individual phases (cf. Formula 1), one could easily replace the model for a particular phase with a more complex one, after deriving the input parameters of the model through a variable calculation as shown in Algorithm 1. A strength of linear models is their interpretability, compared to more complex models such as models based on neural networks.

IV. Training the cost model

The coefficients in the cost models must be identified before prediction. It is difficult and error-prone to profile the file systems in supercomputers to directly derive these coefficients. The I/O path in HPC clusters is complicated and domain scientists usually cannot measure the detailed performance of each step in the path. We train the model trough linear regression to derive these coefficients. We measure the performance of reading and writing synthetic arrays by hyperslab selections and ensure that all parameters in the cost model are varied. The models are trained on multiple HPC clusters.

The shape of the synthetic array to train the serial cost model is $8,388,608 \times 128$ and each cell is a 4-byte integer. The initial chunk shape is 65536×1 and extended by 2 to 128 times on the two dimensions respectively. We define three types of hyperslab selection, including reading or writing the full array, a column and 1,000 rows randomly selected from the array. A chunk can be accessed more than once to read or write the 1,000 rows, such that the software cache read time is trained.

Processes read or write separate synthetic arrays to train the parallel cost model. Except for the chunk shape, we vary the array size from 2 GB to 8 GB by changing the length of the first dimension of the arrays. The number of processes varies from 1 to 64 or 20 and the stripe count of HDF5 files stored in Lustre file systems varies from 1 to 64 or 4, depending on the sizes of HPC clusters. Applications cannot change the stripe count in GPFS file systems.

It is worth noting that these synthetic arrays are longer in the first dimension than in the second dimension. This shape is common in many scientific arrays, where a column is an attribute and a row is the observation values in a timestamp.

V. Experimental evaluation

This section presents an evaluation of the cost models for HDF5 and Zarr on three HPC clusters with different hardware and software capabilities. These clusters include: a large cluster, called **Cori** located at the National Energy Research Scientific Computing Center (NERSC); a medium cluster, called **Owens**, located at the Ohio Supercomputing Center (OSC); and a small cluster, called **RI2**. The medium and the small clusters are located at the Ohio Supercomputer Center (OSC). Table III shows the configuration of the three clusters. We use HDF5 version 1.10.2 and Zarr version 2.3.2 in these experiments. We have conducted experiments in Section V-B and Section V-C on Cori. The results presented in Section V-D are from the runs on the Owens and RI2 clusters. We repeat each experiment 10 times and report the median of the measured values, with error bars denoting the 25 and 75 percentiles. We consider the following questions:

1) What is the relative error of the cost model? How does the model behave in different access patterns and on different HPC clusters?
2) When is Zarr faster than HDF5 or vice versa? What is the performance trend when I/O parameters, like the chunk shape, are changed? Can the model correctly pick the fastest among HDF5 and Zarr in each case?

A. Datasets and access patterns

We use two scientific datasets to evaluate the cost model. The first one is a 9 GB neuroscience dataset storing the brain observations. The array shape is $35,660,170 \times 64$ and chunk shape is $69,649 \times 1$. A column represents an instrument and a row represents a timestamp. The array size is increased to 128 GB by extending the array length in the first dimension to evaluate the parallel cost model. We figure out four common access patterns, which are reading or writing 1) the whole array, 2) a set of rows, 3) a column and 4) a set of hyperslabs. We randomly select 1,000 rows and 1,000 hyperslabs with shape $1,000 \times 8$, located in the leftmost 8 columns.

TABLE III: Cluster configuration

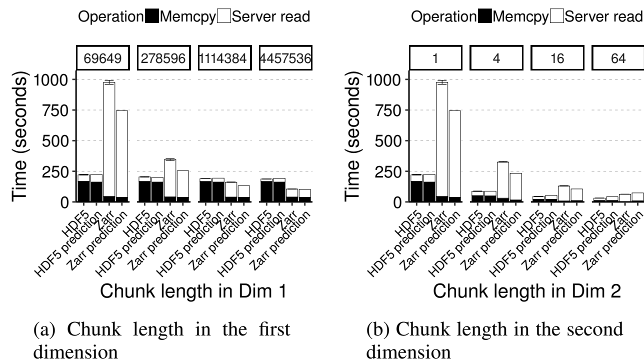
	Cori	**Owens**	**RI2**
Compute nodes	9,688	822	48
CPU	Xeon Phi 7250	Xeon E5-2680	Xeon E5-2680
Memory per node	96 GB	128 GB	128 GB
File system	Lustre	GPFS	Lustre
Storage servers	248	36	4

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

The second dataset is a 3D array from the VORPAL application, which is an acceleration modeling and computation plasma framework developed by Tech-X Corporation. The array shape is $2,400 \times 2,000 \times 3,000$ and the chunk shape is $60 \times 100 \times 300$. Multiple processes write the array, each of which writes a partition of logically contiguous chunks.

B. Serial cost model

1) Read all: We evaluate the serial cost model by reading the 9 GB neuroscience dataset through HDF5 and Zarr. We extend the initial chunk shape by 4, 16 and 64 times in the two dimensions respectively. Figure 4a shows the time breakdown of the real and predicted execution time when the chunk length in the first dimension is increased. The data server read time decreases from 52 seconds to 19 seconds in HDF5 and from 930 seconds to 65 seconds in Zarr. Increasing the chunk size saves the server read time because the number of I/O operations is decreased. The memory copy time is constant and at most takes 90% of the execution time. Hence, only predicting the I/O time is insufficient. Zarr has higher variance because it more frequently communicates with the metadata server, which is a limited resource that is shared by all programs in the cluster.



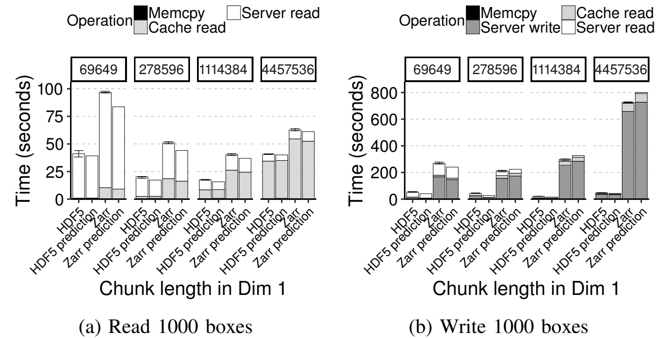
(a) Chunk length in the first dimension

(b) Chunk length in the second dimension

Fig. 4: Read experiment. The memory copy operation takes as much as 90% of the execution time. The model correctly predicts the relative performance between HDF5 and Zarr.

Figure 4b shows the breakdown of the real and predicted execution time when the chunk length in the second dimension is increased from 1 to 64. The real memory copy time decreases from 168 seconds to 12 seconds in HDF5 and from 44 seconds to 0.003 seconds in Zarr due to fewer memory copy operations. Zarr does not copy data when the elements of a chunk are contiguous in the data buffer, such that Zarr only spends 0.003 seconds in the memory copy phase when the chunk length equals array length in Dim 2.

2) Read and write boxes: This experiment reads and writes the 9 GB neuroscience dataset by $1,000$ hyperslabs with shape $1,000 \times 8$, located in the leftmost 8 columns. We extend the chunk shape in the first dimension and report the real and predicted execution time in Figure 5. The execution time is composed of the memory copy time, data server read and write time and software cache read time. As shown in Figure 4a, the server read time decreases as the chunk shape is extended due to fewer Lustre read operations. The cache read time increases and is at most $7\times$ larger than the server read time because more data are read from the software cache. Models ignoring the software cache are not accurate. The server write time in Figure 5b ranges from 14 seconds to 35 seconds in HDF5 and from 155 seconds to 675 seconds in Zarr, which is a trade-off between the number of write operations and written data size.



(a) Read 1000 boxes

(b) Write 1000 boxes

Fig. 5: Read and write 1000 boxes. The cache read takes as much as 87% of the execution time. The relative performance is correctly predicted.

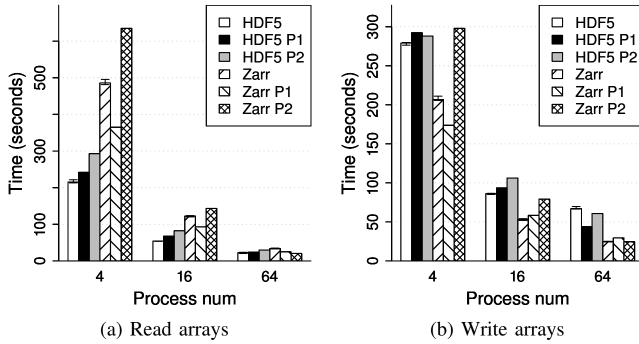
C. Parallel cost model

In this section, we compare the parallel cost model with a cutting-edge I/O cost model on Titan [3]. The model on Titan does not consider the startup time of I/O operations, which is considerable when accessing many chunks. We denote the parallel cost model as *P1* and the cost model on Titan as *P2* in the following figures. Section V-C1 and V-C2 evaluates the models on the neuroscience dataset and Section V-C3 uses the VORPAL application. Chunks in an array are equally partitioned. The VORPAL dataset is stored as an array, while a partition of the neuroscience dataset is stored as an array. A compute node runs a single process, accessing a partition.

1) Process Number: We fix the stripe count as 4 and the stripe size as the chunk size to store the HDF5 files. A chunk file in Zarr is stored in a data server. The number of processes ranges from 4 to 64. Figure 6 shows the real and predicted execution time. The HDF5 is faster than Zarr to read arrays, but slower to write arrays. Both HDF5 and Zarr spends less time as the number of processes is increased by 16 times. The speedups of HDF5 and Zarr are 9.7 and 14 in Figure 6a and 4.16 and 8.28 in Figure 6b. Zarr has higher speedups than HDF5 because Zarr utilizes all the data servers in Luster but HDF5 only uses 4 data servers to read and write chunks. The parallel cost model, *P1*, correctly predicts the performance order between HDF5 and Zarr, but the model on Titan *P2* only correctly predicts the order in 4 out of the 6 experiments.

2) Stripe Count: In this experiment, we use 16 processes to read HDF5 arrays while increasing the stripe count from

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.



(a) Read arrays (b) Write arrays

Fig. 6: Parallel I/O experiment. Zarr is at most $2.7\times$ faster than HDF5 in parallel write. Our model (*P1*) correctly predicts the relative performance of HDF5 and Zarr , while an I/O model from prior work (*P2*) correctly predicts 4 experiments.

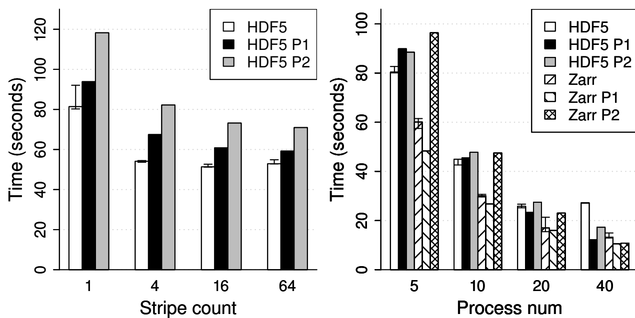


Fig. 7: Striping experiment. Execution time decreases due to lower resource contention.

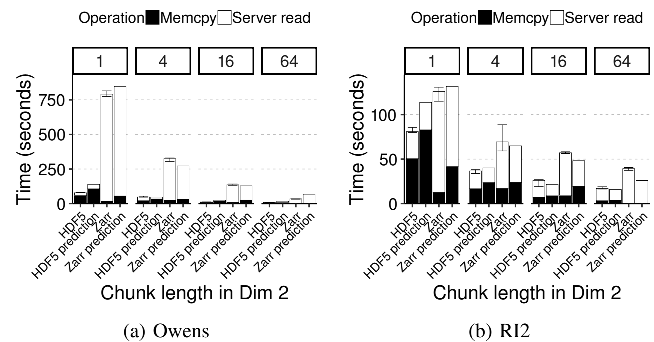
Fig. 8: VORPAL. *P1* and *P2* correctly predict the performance order in all and 3 experiments.

1 to 64. Figure 7 shows the real and predicted HDF5 execution time. The real HDF5 execution time decreases from 81 seconds to 54 seconds when the stripe count is increased from 1 to 4 and almost does not change when we continue to increase the stripe count. The predicted time of *P1* and *P2* decreases from 93 seconds to 59 seconds and from 118 seconds to 71 seconds respectively. Reading data from more data servers takes less time due to lower resource contention.

3) VORPAL: The VORPAL application shows the robustness of the parallel I/O cost model. Arrays to train the model and in other experiments are 2D long tables and an HDF5 process accesses a contiguous region in the file. However, chunks accessed by a process in the VORPAL application are scattered in the HDF5 file because the array is partitioned on the fast changing dimension. We set the stripe count as 40 and vary the number of processes. Figure 8 shows the results. The HDF5 execution time decreases from 80 to 25 seconds when the number of processes is increased from 5 to 20 and does not change later. Zarr always spends less time than HDF5. The *P1* predicts the order of HDF5 and Zarr correctly in all the experiments while *P2* wrongly predicts the order when the number of processes is 5.

D. Evaluation on medium- and small-sized clusters

In this section, we evaluate the serial and the parallel cost models on Owens and RI2 clusters located at OSC. The parallel file system in RI2 is Lustre and that in Owens is GPFS. Figure 9 shows the results of reading the 9 GB neuroscience dataset to evaluate the serial cost model. The chunk length in the second dimension is varied from 1 to 64. The memory copy operation is the bottleneck, taking as much as 76% of the execution time. Both the memory copy time and the server read time decrease as the chunk length is increased due to the fewer memory copy and I/O operations. The predicted costs also decrease. The model correctly predicts the relative performance order between HDF5 and Zarr.



(a) Owens (b) RI2

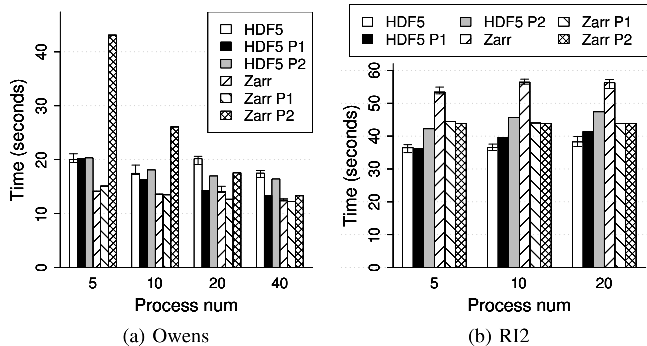
Fig. 9: Time for reading an array with variable chunk shape. The memory copy operation respectively takes as much as 76% and 62% of the execution time on Owens and RI2. The model correctly predicts the relative performance order between HDF5 and Zarr.

Figure 10 shows the results of evaluating the parallel cost model through the VORPAL application. In contrast with the Cori cluster, the execution time on the two clusters does not change significantly when more processes read the array. The predicted time of the parallel cost model has the same trend as the execution time. Zarr is faster than HDF5 on Owens, but slower on RI2. The parallel cost model correctly predicts the relative performance between HDF5 and Zarr, while the cost model on Titan correctly predicts in 2 out of the 7 experiments.

E. Model accuracy

Figure 11 reports the predicted and real execution time of HDF5 and Zarr in all the serial and parallel experiments, conducted on the three clusters. The serial experiments evaluate the model on 4 access patterns, which are accessing the whole array, a set of rows, a column and a set of hyperslabs. Section V-B reports the experiments of accessing a whole array and a set of hyperslabs. The parallel experiments vary the number of processes and stripe counts. The model correctly predicts the fastest library between HDF5 and Zarr in 97 out of the total 111 experiments. As shown in Figure 12, the model correctly predicts the fastest library in 94% of the 47 parallel

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.



(a) Owens (b) RI2

Fig. 10: VORPAL on Owens and RI2. *P1* and *P2* correctly predict the relative performance in all and 2 experiments.

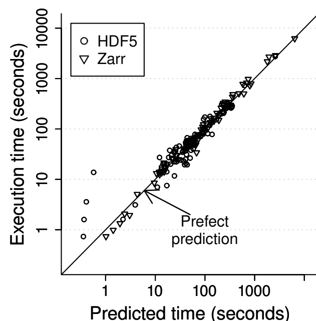


Fig. 11: The real and predicted time of P1. The RMSE is 0.29.

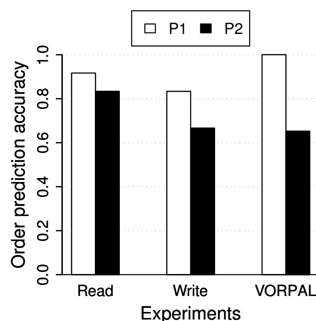


Fig. 12: Accuracy of predicting the fastest library in parallel experiments.

experiments, in contrast with 70% for the cost model on Titan. The RMSE of our cost model in all experiments is 0.29.

Takeaways. The experiments show five main takeaways. (i) The three phases in the I/O path, memory copy, cache read and data server I/O, spend respectively up to 90%, 87% and 99% of the end-to-end execution time in different access patterns. Hence, models only focusing on a specific phase are not enough to predict the execution time. (ii) HDF5 is at most $15\times$ faster than Zarr in the serial applications, while Zarr is up to $2.7\times$ faster than HDF5 for parallel write. It is crucial for users to identify the most efficient array management library based on the access patterns in scientific applications. (iii) The execution time decreases as the number of processes increases on the Cori cluster, while not changing on the Owens and RI2 clusters. Experience of tuning configurations on a specific cluster is not portable to other clusters. (iv) The cost model, with RMSE 0.29, correctly predicts the relative performance of HDF5 and Zarr in 97 out of 111 evaluation experiments.

VI. RELATED WORK

I/O cost prediction. Predicting the I/O performance is well studied in the HPC and database communities. Previous works proposed various cost models with different granularity and accuracy. We categorize these models into 3 types.

Analytic models. The analytic models characterize underlying file systems with a few features and explicitly define formulas to estimate the I/O cost based on these features. Hass et al. [1] decompose the end-to-end time of an I/O request in hard disks into multiple unit operations, including seek, latency and page transfer, and estimate the costs of different *ad hoc* joins by multiplying the unit time of each operation with the number of instances of the operation in the joins. Wu et al. [5] assume the I/O cost is composed of a constant latency and the data transferring time which is proportional to the transferred data size. Gulati et al. [2] builds a linear I/O cost model, depending on the to the average outstanding I/Os, I/O size, read-write ratio, and randomness of the I/O requests. These works profile the file systems to determine the coefficients in the models. However, profiling the parallel file systems in supercomputers is challenging for domain users. These works model the performance of serial applications.

Song et al. [9] estimate the resource contention on storage nodes and network to predict the parallel I/O performance in parallel file systems. However, the Infiniband in cutting-edge supercomputers is fast enough to avoid resource contention compared with the slow storage devices, like HDD. Xie et al. [3] predicts the I/O performance in Titan supercomputer. The model does not consider the startup time of I/O operations, which is considerable when array management libraries access many chunks. We use the model as the baseline in our experiments to evaluate our parallel I/O cost model.

ML models. Machine learning techniques are also used to predict the I/O performance. These models abstract the I/O activities as a vector from the applications' view. Ganapathi et al. [10] predict the performance metrics of database queries through the *KCCA* algorithm. A query plan, a tree where nodes are basic operators, is vectorized as the number of instance count and cardinality sum for each operator in the tree. Zhang et al. [11] use the transform regression algorithm to predict the performance of XML queries. The features in the regression model are the number of visited XML nodes, the returned XML elements, page requests and the elements inserted into the buffer. It is not trivial to capture and represent the I/O characteristics of array query workloads as vectors.

Abstract quantity. These models abstract a set of I/O operations as a unit and predict the I/O performance based on the number of unit instances. Chaudhuri et al. [12] and Luo et al. [13] predict the database query cost by counting the *GetNext()* function instances and the size of processed tuples respectively. Mozafari et al. [14] and Duggan et al. [15] assume that each I/O request or disk page access spend constant time.

Memory cost prediction. Zhang et al. [16] predicts the memory access cost based on the read and written data size in memory. Byna et al. [17] and Rahman et al. [18] predict the memory access cost based on cache misses in hierarchical memories. These models are either oversimplified or dependent on many variables. Constructing complicated models is challenging and not necessary to predict the performance of array management libraries. The memory access cost

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.

dominates the execution time of array management libraries when there are many *memory copy* operations.

These I/O and memory cost models mentioned above do not include the full I/O path in array management libraries. Steps in the path dominate the execution time in different scenarios. These models are dependent on detailed variables, like the number of disk seek operations. I/O and memory characterization tools only gets the values of these variables in an execution of an application, which limits the model usage.

VII. Conclusions

Many scientific applications use array management libraries to read and write arrays. Whereas a substantial body of prior work has focused on modeling I/O performance, we find that I/O accounts for as little as 10% of the execution time when accessing array objects via popular I/O libraries. Models that only focus on I/O therefore cannot accurately capture new overheads (such as transforming data to a particular layout) and new optimizations (such as library-level caching) when accessing data through popular I/O libraries.

This paper builds a new analytical cost model to predict and compare the end-to-end performance of two popular I/O libraries, HDF5 and Zarr. HDF5 and Zarr embrace different storage principles: HDF5 stores all chunks in one file, while Zarr stores each chunk as a separate file. We systematically evaluate the model using two scientific applications on three HPC clusters, equipped with the Lustre and GPFS file systems. The model correctly predicts the fastest library between HDF5 and Zarr in 94% of the parallel experiments, in contrast with 70% for a cutting-edge model that only focuses on I/O performance. Automatically tuning the I/O configuration of scientific applications based on the cost model is a promising avenue for future work.

Acknowledgments

This work is supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract Numbers DE-AC02-05CH11231; the National Institute Of Mental Health of the National Institutes of Health under Award Number R24MH116922; and the National Science Foundation grant SHF-1816577. This research used resources of the National Energy Research Scientific Computing Center, which is DOE Office of Science User Facilities supported by the Office of Science of the U.S. Department of Energy under Contract Number DE-AC02-05CH11231. The content is solely the responsibility of the authors and does not necessarily represent the official views of any federal agency. We would like to thank Kristofer Bouchard and his team for providing us with example ephys data used in the performance evaluation.

References

[1] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla, “Seeking the truth about ad hoc join costs,” *The VLDB Journal*, vol. 6, no. 3, pp. 241–256, Aug. 1997.

[2] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar, “Basil: Automated I/O load balancing across storage devices,” in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, ser. FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 13–13.

[3] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, “Predicting output performance of a petascale supercomputer,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: ACM, 2017, pp. 181–192.

[4] The HDF Group, “Enabling a strict consistency semantics model in parallel HDF5,” 2012.

[5] T. Wu, J. Chou, S. Hao, B. Dong, S. Klasky, and K. Wu, “Optimizing the query performance of block index through data analysis and I/O modeling,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 12:1–12:10.

[6] D. Kang, V. Patel, A. Nair, S. Blanas, Y. Wang, and S. Parthasarathy, “Henosis: Workload-driven small array consolidation and placement for HDF5 applications on heterogeneous data stores,” in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 392–402.

[7] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, “Data elevator: Low-contention data movement in hierarchical storage system,” in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016, pp. 152–161.

[8] B. Behzad, S. Byna, Prabhat, and M. Snir, “Pattern-driven parallel I/O tuning,” in *Proceedings of the 10th Parallel Data Storage Workshop*, ser. PDSW '15. New York, NY, USA: ACM, 2015, pp. 43–48.

[9] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, “A cost-intelligent application-specific data layout scheme for parallel file systems,” in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 37–48.

[10] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, “Predicting multiple metrics for queries: Better decisions enabled by machine learning,” in *2009 IEEE 25th International Conference on Data Engineering*, March 2009, pp. 592–603.

[11] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang, “Statistical learning techniques for costing XML queries,” in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05. VLDB Endowment, 2005, pp. 289–300.

[12] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, “Estimating progress of execution for SQL queries,” in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 803–814.

[13] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, “Toward a progress indicator for database queries,” in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 791–802.

[14] B. Mozafari, C. Curino, A. Jindal, and S. Madden, “Performance and resource modeling in highly-concurrent OLTP workloads,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 301–312.

[15] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, “Performance prediction for concurrent database workloads,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 337–348.

[16] C. Zhang and C. Ré, “Dimmwitted: A study of main-memory statistical analytics,” *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1283–1294, Aug. 2014.

[17] S. Byna, Xian-He Sun, W. Gropp, and R. Thakur, “Predicting memory-access cost based on data-access patterns,” in *2004 IEEE International Conference on Cluster Computing*, Sep. 2004, pp. 327–336.

[18] S. M. F. Rahman, Q. Yi, and A. Qasem, “Understanding stencil code performance on multicore architectures,” in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 30:1–30:10.

Authorized licensed use limited to: The Ohio State University. Downloaded on July 30,2020 at 23:23:11 UTC from IEEE Xplore. Restrictions apply.