Battle of the Defaults: Extracting Performance Characteristics of HDF5 under Production Load

Bing Xie*†, Houjun Tang*‡, Suren Byna‡, Jesse Hanley†, Quincey Koziol‡, Tonglin Li‡, Sarp Oral†
† Oak Ridge National Laboratory
‡ Lawrence Berkeley National Laboratory

***Abstract*—Popular parallel I/O libraries, such as HDF5, provide tuning parameters to obtain superior performance. However, the selection of effective parameters on production systems is complex due to the interdependence of I/O software and file system layers. Hence, application developers typically use the default parameters and often experience poor I/O performance. This work conducts a benchmarking-based analysis on the HDF5 behaviors with a wide variety of I/O patterns to extract performance characteristics under the production workload. To make the analysis well controlled, we exercise I/O benchmarks on POSIX-IO, MPI-IO, and HDF5 using the same I/O patterns and in the same jobs. To address high performance variability in production environments, we repeat the benchmarks across I/O patterns, storage devices, and time intervals. Based on the results, we identified consistent HDF5 behaviors that appropriate configurations and operations on dataset layout and file-metadata placement can improve performance significantly. We apply our findings and evaluate the tuned I/O library on two supercomputers: Summit and Cori. The results show that our solution can achieve more than 10× performance speedup than the default for both of the systems, suggesting the effectiveness, stability, and generality of our solution.**

I. Introduction

In high-performance computing (HPC), many domain scientists manage their data via I/O libraries, such as HDF5 [1], ADIOS [2], PnetCDF [3], etc. These libraries sit between applications and the underlying storage systems, support a variety of data structures, and maneuver toward high throughput across multiple I/O layers. For instance, HDF5 provides parallel I/O services via MPI-IO interfaces, stores files on large-scale parallel file systems such as Lustre [4] and GPFS [5], and allows end users to handpick the specific configurations for HDF5 internal operations (e.g. HDF5 metadata cache size and management), MPI-IO operations (e.g. through MPI_Info object), and file systems (e.g. data layout properties). We discuss the tunable parameters in details in Section II.

Although given the tuning options, most users still adhere to the default configurations set by the libraries and systems, due to a lack of expert knowledge on I/O middleware layers. Unfortunately, such configurations, typically selected based on heuristics rather than performance perspectives from applications, can hardly address the needs of specific applications, especially for those that demand high performance. Instead of delegating the performance-tuning tasks to the end users, we choose to set the default configurations agilely rather than heuristically. To this end, we study how the applications behave via I/O libraries and how the I/O middleware layers interact with each other and with the underlying file systems.

To achieve this overarching goal, we take HDF5 as an example to characterize the behavior of I/O libraries in production HPC environments. As a popular parallel I/O library, HDF5 organizes application data as *datasets* and metadata attributes describing the datasets together. The layout and placement of HDF5 datasets and metadata are configurable parameters. In this work, we profile the performance of these generic tuning parameters in HDF5, identify the best configurations across I/O patterns, and implement our solution as the default configurations in the HDF5 library. Different from previous works on tuning I/O performance of specific systems and/or specific applications [6], [7], [8], [9], [10], [11], we profile the behaviors of the target I/O library based on its design principles and search for the solution generally applicable to various I/O patterns and supercomputer systems.

This work takes a benchmarking approach to extract the HDF5 behaviors on the Summit and Cori supercomputers. We conduct controlled experiments for I/O benchmarks, exercise various I/O patterns on POSIX-IO, MPI-IO, and HDF5, and profile I/O performance across compute nodes, storage devices and time intervals. In the approach, we swept the parameter space using $3,328$ benchmark settings with above 100 repetitions each, and eventually identified a set of effective tuning parameters in HDF5 on dataset layout and placement, and operations on file metadata. Based on our empirical study of measurements, we updated HDF5 with new system-specific parameters that lead to superior I/O performance. We summarize the contributions below:

- We introduced a benchmarking approach to understand the I/O performance of large-scale production systems.
- We identified efficient HDF5 alignment and file metadata optimizations and built the chosen configurations as the default in HDF5. Our solution is adopted by OLCF for production use and is publicly available[1].
- We evaluated the tuned HDF5 on Summit and Cori. The results show that the tuned HDF5 achieves more than 10× performance speedup on both Summit and Cori, suggesting that our solution is consistently effective across systems.

*Equal contribution.

[1]https://github.com/xiexbing/hdf5-work

II. Background

A. Parallel I/O in HDF5

The Hierarchical Data Format version 5 (HDF5) [1] is a self-describing file format and the supporting library. By using HDF5, scientists can manage various data structures or objects and their corresponding metadata within a single *HDF5 container* or *file*. Across HPC facilities, HDF5 is widely used in scientific applications [6]. This work is built upon the HDF5-1.10.6 release, which is the default module version on Summit and Cori at the time of this writing.

1) HDF5 file: Each HDF5 file contains two types of data: *application data* and *file metadata*. Typically, application data are organized and stored as datasets, with each dataset having its own data structure. Besides datasets, the file also contains file metadata, including application metadata (e.g., groups and attributes), file infrastructure information, dataset metadata (e.g., B-trees maintaining dataset locations) and a superblock (e.g., file identifier). In this work, we categorize file metadata into two classes: *superblock* and *descriptive metadata*. In an HDF5 file, when more datasets are created, the total size of file metadata increases. Specifically, the size of superblock is 96 bytes; in typical uses (1—100 datasets in a file), the total size of file metadata is less than 100KB.

Moreover, in a file, the locations of the superblock, the descriptive metadata and datasets can be calculated before file creation. In particular, the superblock is always located at the beginning of a file and the locations of the descriptive metadata and datasets are both configurable. This tuning allows aligning the metadata and data with file system page boundaries, leading to better performance on some file systems. To set the location of datasets, users can set an *alignment value*. At runtime, HDF5 places a dataset starting from the alignment value or from its multiples depending on the lengths of datasets in the file. By default, the alignment value is set as 2KB to avoid file fragmentation. However, we find that, from the performance perspective, this default setting is not optimal for larger datasets (§III and §IV).

For the descriptive metadata, users may configure a file in the way that the entirety of this metadata is placed after the superblock of an HDF5 file and before all of the datasets, or by default, let the datasets interleaved with the metadata, such as shown in Figure 1. Moreover, HDF5 caches the entire file metadata and offers end users the option to defer the metadata flushing till file close.

2) HDF5 parallel write operations: In Figure 1(a), we show an example of writing to an HDF5 file in parallel with 9 MPI processes. In particular, the file contains three datasets, in which each of the 9 concurrent processes (Rank 0 - 8) write a different data block in one of the three datasets of the file. The write operation progresses in three steps. At first, each process writes its data block in its dataset. In the 2^{nd} and the 3^{rd} steps, the processes write the descriptive metadata and superblock.

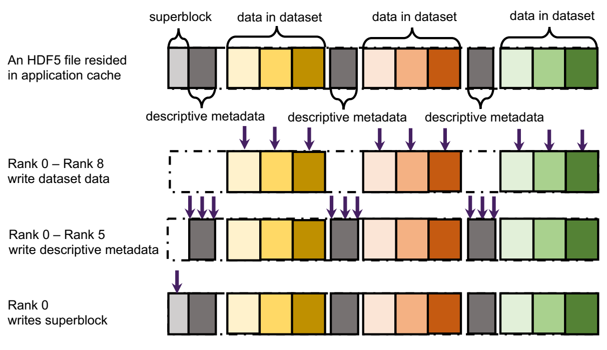
Specifically, in the last two steps, users may choose to write the metadata with multiple processes (default option), or alternatively, with collective MPI-IO calls and write with one process. In particular, with multi-process metadata write option, after the slowest process completes the last byte of its data chunk, a subgroup of the 9 processes each update a different part of the descriptive metadata with independent write calls. The group size is determined by the data size of the overall dataset metadata in an HDF5 file. In Figure 1(a), Rank 0 - 5 write dataset metadata. And after the slowest process completes its write on its corresponding metadata, Rank 0 updates the superblock independently indicating the completion of the write operation. On the other hand, when *HDF5 collective metadata I/O* is enabled, all 9 processes participate in a single collective MPI write call to update the descriptive metadata and superblock in a sequence, with only Rank 0 performing the write.

3) HDF5 parallel read operations: As is shown in Figure 1(b), we illustrate HDF5 read operations using the file discussed in §II-A2. In summary, a parallel HDF5 read operation progresses in three steps; the first two steps read the superblock and the metadata; in the 3^{rd} step, the 9 participating processes each read its data chunk from the parallel file system.

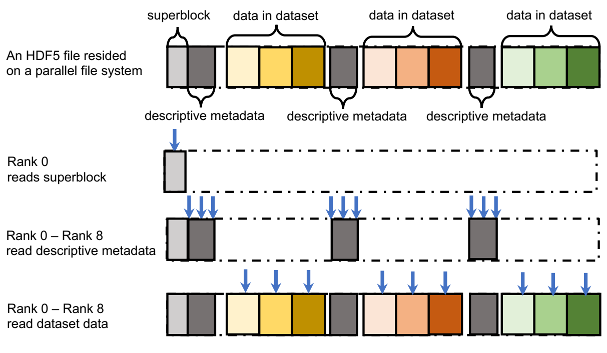
Similar to the write operation, users can choose to read the metadata with multiple processes (default) or a single process. With the default mode, Rank 0 first reads the superblock with an independent MPI-IO read call and broadcasts its completion to the other 8 processes. Next, the 9 processes each initiate an independent MPI-IO read call to retrieve a specific portion of the metadata, describing the file infrastructure and the dataset that contains its data chunk. Alternatively, when users choose collective metadata I/O, Rank 0 reads both the superblock and the metadata, and broadcasts the completion about the entire file metadata to the other 8 processes.

B. I/O Systems on Production HPC Systems

1) GPFS on Summit: Summit, the fastest supercomputer in the world between 2018 and June 2020, is an IBM-built supercomputer and housed at the Oak Ridge Leadership Computing Facility (OLCF). It consists of 4,608 compute nodes, with each node containing 2 IBM POWER9 processors (2×22 CPUs) and 6 NVIDIA V100 accelerators (6 GPUs). Summit is connected to Alpine, the center-wide IBM Spectrum Scale file system (GPFS). Alpine provides roughly 250 PB of usable storage capacity and 2.5TB/s peak I/O bandwidth. Each Summit node runs a GPFS client software stack and provides I/O services for both metadata and data. Alpine is a single POSIX namespace comprised of 154 Network Shared Disk (NSD) servers. Each NSD server manages one GPFS Native RAID (GNR) and serves as both a storage server and a metadata server. As a GPFS deployment, Alpine absorbs file data in parallel. Shown as Figure 2(a), for each file, GPFS partitions its data into a sequence of equal-size data blocks and distributes the block-sequence across an NSD-sequence in a round-robin way. Users have no control on the block size (*GPFS block size*) nor the NSD sequence. Instead, a default GPFS block size is determined at the creation time of a GPFS filesystem. The NSD sequence starts from a randomly-chosen NSD server and may span over the entire server pool by

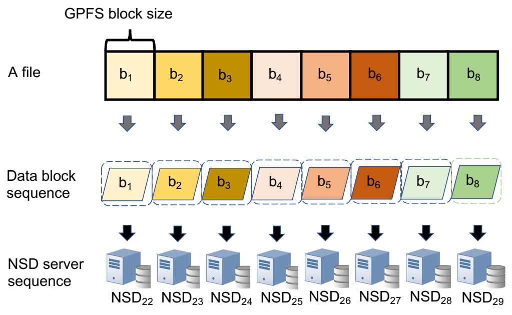


(a) HDF5 write

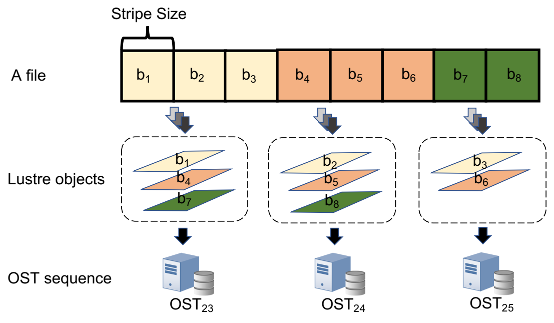


(b) HDF5 read

Fig. 1: Simple examples of parallel HDF5 write and read operations with 9 MPI processes



(a) GPFS striping policy



(b) Lustre striping policy

Fig. 2: Data striping policies on GPFS and Lustre deployments.

following the system-configured NSD order. In Alpine, the default GPFS block size is configured as 16MB.

We find that when issuing large bursts (≥16MB) to write-share an HDF5 file in parallel, the write performance is sensitive to the dataset layout on disks: the inappropriate configurations on alignment settings may cause a mismatch and lead to the significant performance loss (§III-C).

Moreover, GPFS manages file system fragmentation with *subblocks*. When a file size is smaller than or is not multiples of the GPFS block size, the file will be stored in one/more subblocks or in a number of full blocks plus some subblocks. At the system side, GPFS merges/migrates the subblocks to the local and/or remote NSDs to form full blocks and alleviate the fragmentation accordingly. In Alpine, the subblock size is 16KB. In our benchmarking analysis, we find that, for some small writes (256KB), the appropriate configurations on alignment values may benefit significantly from the subblock arrangement in Alpine. We return to this issue in §III-C.

2) Lustre on Cori: We evaluate our benchmarking results on Cori. Cori is a Cray XC40 supercomputer launched at NERSC, comprised of 2,338 Haswell compute nodes, and connected to a Lustre-based file store, called Cori Scratch. Cori Scratch provides I/O services to Cori and the other computational systems at NERSC, with 30 PB of usable disk space and above 700GB/s peak I/O bandwidth.

On Cori, each compute node runs a Linux operating system, which invokes the local Lustre kernel modules for two file system services, Object Storage Client (OSC) and Metadata Client (MDC). At the file system side, Cori Scratch is a single POSIX namespace with five Metadata Servers (MDSes) and 248 Object Storage Servers (OSSes). In particular, each MDS hosts a different part of the namespace; each OSS manages one Object Storage Target (OST), with each OST configured as a Grid-RAID. For the Lustre on Cori Scratch, each client is configured to connect to a single MDS and 248 OSSes.

Different from GPFS, Lustre allows users to set its data layout on disks with configurable parameters. Figure 2(b) presents a simple example about the Lustre striping policy on a single-shared file. In summary, a file is partitioned into a sequence of equal-sized data blocks; the data blocks are distributed across a sequence of OSTs in a round-robin way. Here, the block size, the length of the OST sequence, and the OST start index are the three configurable parameters in Lustre, called *stripe size*, *stripe count* and *starting OST*.

On Cori Scratch, the default stripe count is 1 and the stripe size is 1MB. This setting works well for small files writing by serial programs, but parallel programs accessing the same file from multiple processes could result in poor performance. NERSC allows users to set their own striping and provides a set of recommendations. In particular, for single-shared files, the NERSC categorizes the files into five groups based on file size and recommends each group 1 out of 4 striping configurations [12]. We discuss the impact of these recommendations for different I/O patterns in §IV.

III. HDF5 I/O Profiling on Summit

A. Overview

This work profiles the I/O performance of Summit supercomputer and builds the analysis upon two groups of IOR benchmarks: dataset-layout and file-metadata experiments. The first group studies the HDF5 performance with various dataset layouts, i.e. various alignment settings on a file system; the second group focuses on understanding HDF5 behavior with various file metadata configurations. Since HDF5 uses MPI-IO API for parallel I/O and MPI-IO in turn uses POSIX-IO API for performing I/O operations, we measure performance of all these layers. We chose IOR because it allows testing POSIX-IO and MPI-IO directly. Comparing HDF5 with other APIs shows its overhead to write self-describing metadata as well as any other overheads in writing HDF5 datasets. Our I/O profiling is done primarily on Summit, we do see similar trends with the Lustre parallel system on Cori, and verified the tuning parameters are effective with applications runs on it. In summary, we design the experiments by following three strategies.

- To ensure the experiments are well controlled, we run IOR with consistent and strategically chosen I/O patterns. In particular, in the dataset-layout experiments, the same patterns exercise the POSIX-IO, MPI-IO, and HDF5 APIs within the same job submissions. We create job scripts to run the IOR benchmark with different I/O APIs and various HDF5 configurations that perform a set of I/O patterns. As they run on the same groups of compute nodes and experience similar system conditions, their relative performance provides insights on HDF5 I/O performance tuning.
- To obtain good coverage of the I/O behaviors of scientific applications, in the two groups of experiments, we vary the IOR parameters on compute nodes, processes per node (PPN), I/O burst sizes, and I/O configurations such as alignment size, metadata modes, and stripe settings.
- To address high variability on the target supercomputing systems, we run each experiment repeatedly for at least 100 times in multiple job submissions, across different compute nodes, storage devices, and time intervals.

B. Benchmarking Method

We develop a performance analysis method using a statistical benchmarking methodology proposed for supercomputer I/O systems under production load [13], [14], [15], [16], [17]. Specifically, we design the controlled experiments on POSIX-IO, MPI-IO, and HDF5 with various configurations for performance tuning. We performed two groups of experiments that each executes IOR within a benchmarking harness to coordinate simultaneous I/O bursts from multiple cores and nodes. Each experiment produces a set of I/O sampling data.

After the job starts, it reads a job description file, which specifies the IOR executions with a multi-level for-loop. Each loop varies the values of an IOR parameter on I/O APIs (in the data-set layout experiments), or I/O patterns (e.g., read/write data sizes, number of cores in use), or HDF5 configurations. We submit each such job many times and execute one at a time through job dependency setting to avoid self-interference.

Each job includes several IOR executions. Each execution simulates a typical I/O pattern: in a job, the synchronous processes read/write a single shared file. In particular,a number of benchmark processes each issue a *file_open()*, a read or write system call, and a *file_close()* in a sequence. The processes are synchronized with MPI barriers before *file_open()* and after *file_close()*. To avoid read-cache effects in I/O reads, each of the read calls changes its ordering for readback. To avoid write-cache effects in I/O writes, each of the write calls is followed by an *fsync()* to flush data to the disks.

For each IOR execution, we report the end-to-end performance from the minimum of *file_open()* to the maximum of *file_close()* among the bursts; for each process in an IOR execution, we analyze the I/O bandwidths and the times on *file_open()* and *file_close()*. In addition, for each process in an IOR execution, we also measure the times of the HDF5 I/O calls about dataset and file metadata operations. We extract these times from Darshan DXT logs.

C. Dataset Layouts and File System Defaults

1) Experimental Setup: We first perform the dataset layout experiments, in which we search for the best file data layout and parallel I/O settings in HDF5 by exercising IOR with POSIX, MPI-IO, and HDF5. We used the default spectrum-mpi/10.3.1.2-20200121 MPI module and HDF5 1.10.6 version on Summit.

As a high-level I/O library, HDF5 implements parallel I/O services using the lower-level MPI-IO operations and MPI-IO internally uses POSIX calls. In this group of experiments, we profile the performance of these three I/O APIs under the same I/O patterns and in the same IOR jobs. We use the performance measurements of POSIX and MPI-IO as the baseline to evaluate the effectiveness of different alignment settings on HDF5.

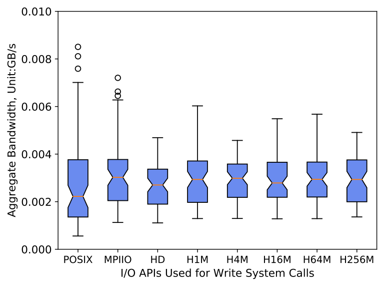
To identify the best dataset layout, we conduct three experiments: *baseline experiment*, *multi-block experiment*, and *multi-core experiment*. Each experiment evaluates the efficiency of the parallel I/O pipelines with a different type of I/O pattern observed in production use. Table I presents the experiments and their parameters in detail. Specifically, in each experiment, we vary the number of compute nodes in use (N), processes per node (PPN), the number of data blocks in each read/write burst (T), and the aggregate data size per node.

- **Baseline experiment.** N coordinated processes each run on a single core from a single node and write an equal-sized single data block with data size of W-bytes.
- **Multi-block experiment.** N processes each run on a single node and in write T equal-sized blocks (in the same system call) sequentially with W-bytes of aggregate data per node.
- **Multi-core experiment.** $PPN \times N$ processes each write a single block. We use PPN cores per node and produce W-bytes of aggregate data per node.

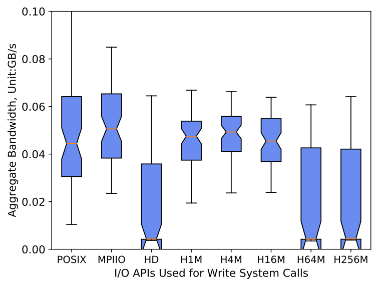
For each setting in an experiment, we collect 102–157 sample points across compute nodes, storage devices and

Experiment \ Parameter	N	PPN	T	Aggregate data size per node (W)
Baseline experiment	2, 8, 32, 128	1	1	1KB, 16KB, 256KB, 1MB, 16MB, 256MB, 1GB
Multi-block experiment		1	4, 8, 16	16KB, 256KB, 16MB, 256MB, 1GB
Multi-core experiment		4, 8, 16	1	

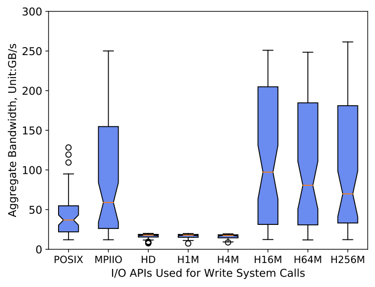
TABLE I: Parameters and Values used in the dataset-layout experiments III-C.



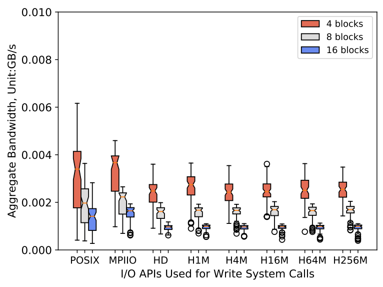
(a) W=16KB baseline



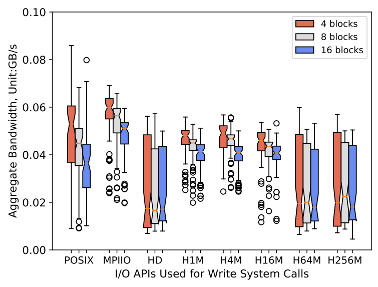
(b) W=256KB baseline



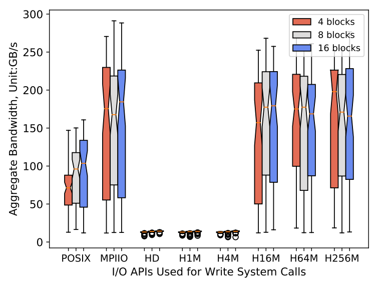
(c) W=1GB baseline



(d) W=16KB multi-block



(e) W=256KB multi-block



(f) W=1GB multi-block

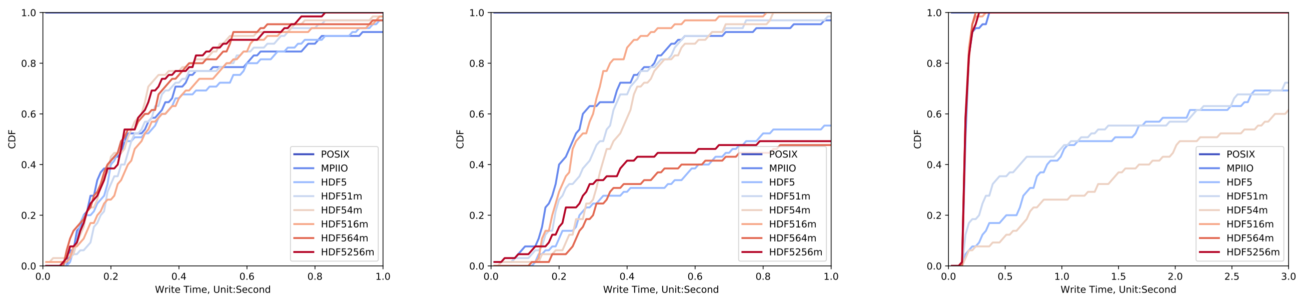
Fig. 3: Aggregate Bandwidths observed from the baseline and multi-block experiments using 128 nodes. We report the HDF5 results based on the specific alignment values. For example, HD means the HDF5 runs with the default alignment setting; H16M means the HDF5 runs with alignment value set as 16MB.

time intervals. Due to the space limitation, this work mainly presents the figures generated by the IOR instances executed on 128 nodes and application performance on 512 nodes. For all experiments with all settings, we see consistent and stable patterns when scaling out. We provide all of the experimental results in a GitHub repository[1].

2) Write Performance Analysis: We summarize and report the results in Figures 3, 5(a), 4 5(d). As is shown in Figures 3(c) and 3(f), for large writes ($W \geq 16MB$), HDF5 reports two types of performance patterns: the low-bandwidth pattern (for $\leq$ 4MB alignment settings, e.g. HDF51m, HDF54m) and the high-bandwidth pattern (for $\geq$ 16MB alignment settings, e.g. HDF516m, HDF564m, and HDF5256m). Specifically, for the large bursts on all write experiments, the high-bandwidth patterns report 3×—13× speedup. To gain a better understanding of the root causes of the results, we look into the write-call internals: for the baseline and multi-core experiments, we focus on the write time of each burst generated by each participating core/process; for the multi-block experiment, we analyze the write time of the data block generated by each of the write calls from an MPI rank (extracted from Darshan DXT logs).

Figure 4(c) reports the cumulative distribution function (CDF) of the performance of $1GB$ bursts with one process per node on 128 nodes ("baseline" experiment), representing the results of individual processes/blocks on ≥16MB bursts across write scales and three experiments. It is clear that, for the low-bandwidth settings, each burst/block of a process reports a longer write time. Relative to the GPFS striping policy, a burst/block is a sequence of 16MB data chunks, when the starting offset of the first chunk mismatches to the GPFS block size. Each of the 16MB data chunks might be considered as a full block and accordingly be placed on two NSD servers and its performance is determined by the slower server. Or, relative to the GPFS subblock policy, each 16MB chunk might be considered as two non-full blocks and accordingly be partitioned into a number of 16KB subblocks and distribute among the NSDs in the system. In both of the cases, the layout mismatch leads to the poor write performance.

As is shown in Figures 3(a), 3(b), 3(d) and 3(e), for small writes ($W > 16MB$), HDF5 also delivers two types of performance patterns, one for burst sizes 1KB, 16KB and 1MB, the other one for burst size 256KB. Specifically, for 1KB, 16KB and 1MB bursts, all of the alignment settings report similar write performance; for 256KB bursts, 1MB, 4MB and 16MB alignment settings deliver better performance.



(a) W=16KB baseline (b) W=256KB baseline (c) W=1GB baseline

Fig. 4: CDFs of the write times of a specific process in the baseline experiments. Each figure reports the write times of Rank 12 across the runs for W=16KB, 256KB and 1GB, respectively.

Similar to the analysis of large writes, we look into the write time of each burst/block of each process. Figures 4(a) and 4(b) report the CDFs of 16KB and 256KB bursts on a single process from the baseline experiments on 128 nodes, representing the results of individual processes on 16KB and 256KB bursts across write scales and experiments. Specifically, the performance behaviors of 256KB bursts are similar to the large bursts discussed above: each block/process associated with the 1MB—16MB alignment settings delivers similarly high performance as POSIX and MPI-IO do.

It implies that the 256KB bursts might hit another layout mismatch issue on GPFS subblocks. We conclude that, in HDF5, small writes ($< 16MB$) benefit most from the 1MB—16MB alignment settings. We leave further study on the small-size alignment settings around GPFS subblock size (e.g., 8KB, 16KB, 32KB, etc.) as our future work.

Moreover, we find that, for I/O writes with 128 nodes, MPI-IO and HDF5 receive higher peak bandwidth than POSIX does. We find that the root cause is on file open and close. Figures 5(a) and 5(d) report the performance for the three I/O APIS on different write scales. It suggests that, when the concurrent file open/close scales out, POSIX reports progressively poorer performance. It might originate from the file locking in NFSv4 adopted by the GPFS file system deployed on Alpine. In the NFSv4 semantics, MPI-IO and HDF5 may benefit from the share reservation mechanism, which is not supported in POSIX calls in the default implementation. We plan to investigate this further in the near future.

In summary, we conclude that, HDF5 write performance is sensitive to the alignment setting. Specifically, in consideration of the performance of the file fragmentation risk in HDF5 files, we conclude that large bursts ($\geq$16MB) bursts benefit most from 16MB alignment setting and small write bursts benefit most from 1MB alignment setting on Alpine.

3) Read Performance Analysis: We report the read performance in Figure 5, 5(b) shows that for the 1KB—256MB bursts across the target read scales and experiments, the HDF5 runs with all alignment settings reach the similar median and peak performance as POSIX and MPI-IO do with $<$ 15% variances. Moreover, 1GB bursts with 64MB and 256MB alignment settings report >30% better median and peak performance than the other settings (Figure 5(e)), but the advantage decreases when a core initiates more read calls (Figure 5(c)) or when more cores are used per node (Figure 5(f)). Relative to the tight boxplots shown in Figure 5(b), we conclude that, the good performance on 64MB and 256MB alignment settings in the baseline experiment may indicate read bottlenecks within Summit, which is irrelevant to the dataset layout on Alpine. In summary, we conclude that the read performance of HDF5 is insensitive to dataset layout.

D. Tuning File Metadata Operations

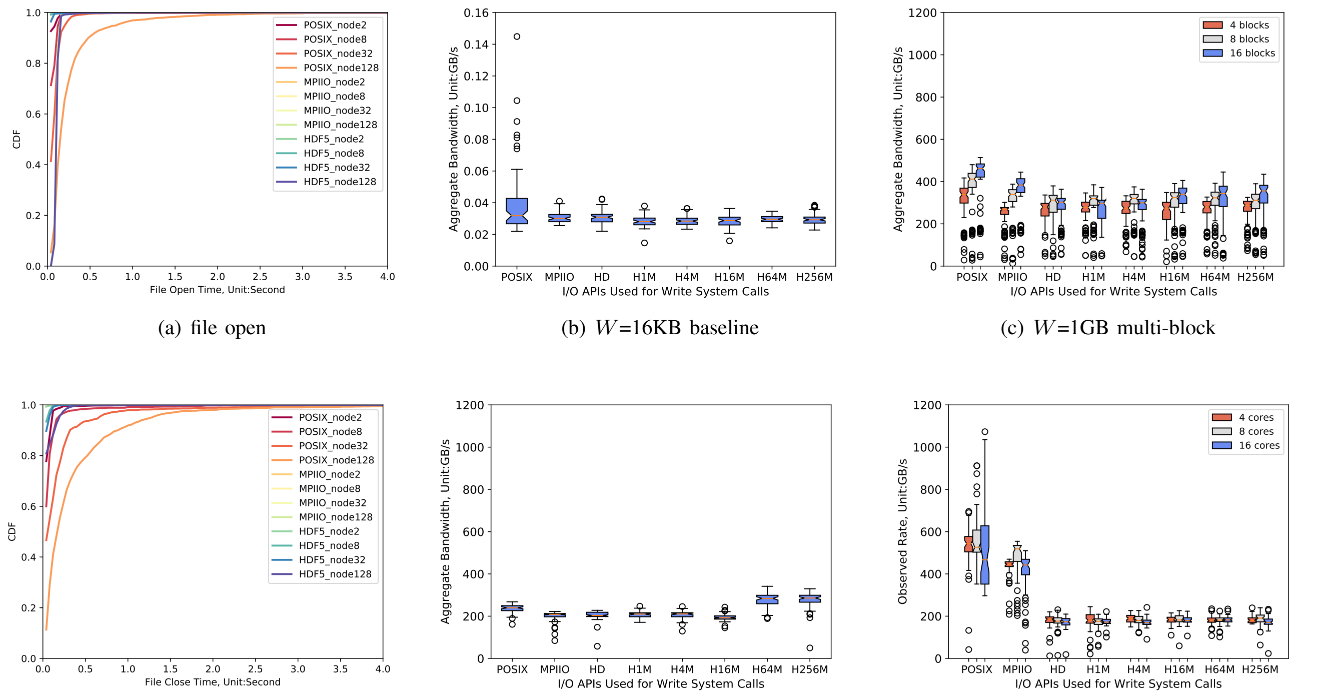
We design this experiment to search for the best configurations on file metadata by varying the HDF5 default configurations and burst sizes.

1) Experimental Setup: Table II presents the parameters and values used in this experiment. Specifically, we fix the alignment settings as 1MB and 16MB for small (<16MB) and large ($\geq$16MB) bursts (discussed in §III-C), respectively.

N coordinated processes read/write D datasets in an HDF5 file. Each process runs on a single core from one node and reads/writes a W-bytes data block in each of the D datasets. Thus, the total data size per node/core is $W \times D$; the total data size of a file is $W \times D \times N$. Moreover, in each IOR execution, we vary the locations of the descriptive metadata in HDF5 files (§II-A1). We alternate the options on file-metadata read/write between default and collective metadata I/O. For HDF5 write, we enable and disable the option of flushing the file-metadata cache at file close.

2) Performance Analysis: We first analyze the time consumption on file-metadata in HDF5 read and write. Specifically, across HDF5 configurations and read/write scales, superblock read consumes $\leq$ 0.03 seconds; more than 76% and 82% of the superblock writes complete within 1 and 2 seconds, respectively. It suggests when reading/writing large bursts, the performance impact from superblock is relatively small.

To investigate the descriptive metadata cost, we take the default configuration on metadata cache (for HDF5 write), vary the metadata locations, and alternate the read/write between default and collective metadata I/O. Figures 6(b) summarizes the read cost. In particular, for the descriptive metadata on read, the read cost increases with the number of datasets; for



(a) file open (b) W=16KB baseline (c) W=1GB multi-block

(d) file close (e) W=1GB baseline (f) W=1GB multi-core

Fig. 5: CDFs of file open and file close (5(a) and 5(d)) and **Aggregate read performance** for 16KB and 1GB bursts in the baseline and multi-core experiments (5(b), 5(c), 5(e), 5(f)). In Figures 5(a) and 5(d), we combine the HDF5 performance across different alignment configurations as different alignment settings do not affect the performance of file open and close.

Parameter / Experiment	N	D	Aggregate data size per node (W)		L	C	F
Fie-metadata experiment	2, 8, 32, 128	2, 8, 32, 128	16KB, 256KB, 1MB,	16MB, 64MB	0, 1	0, 1	0, 1
			alignment setting				
			1MB	16MB			

TABLE II: Parameters and values used in the file-metadata experiment. We define N, W, D in Section III-D. Moreover, L presents the locations of the other metadata in an HDF5 file; 0 means the default configuration where the other metadata is interleaved with datasets; 1 means the configuration where the entire file metadata is placed at the file start. C suggests the use of collective I/O on file metadata; 0 is the default configuration with collective I/O disabled; 1 means collective I/O is enabled. F denotes the option on deferring the file-metadata cache flushing in HDF5 write; 0 means flushing the cache in the HDF5 write calls; 1 means flushing the cache at file close.

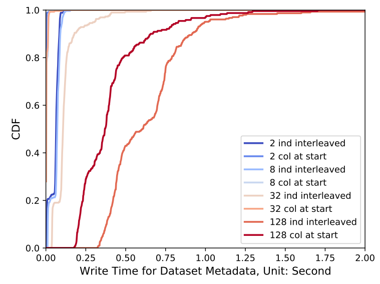
any given number of datasets, better performance is observed with configurations that place the all file metadata at the beginning of the file and use collective I/O on metadata read.

Relative to the HDF5 read process (§II-A3), when reading the descriptive metadata with multiple processes, the coordinated MPI ranks form groups, and each process group shares the metadata about a dataset. This sharing results in worse performance when more datasets are created or more ranks participate in the read. Comparatively, when using collective I/O on this metadata read, Rank 0 fetches the metadata with a sequence of read calls, which avoids the locking conflicts on read-sharing and benefits further on requesting fewer NSD handlers when the entire metadata is placed at the file start and on the same GPFS block.

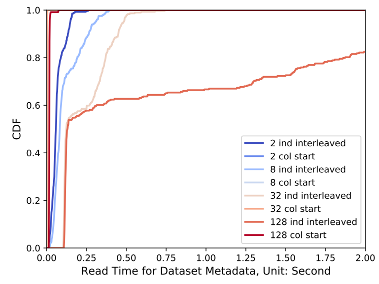
Figure 6(a) presents the write cost on the descriptive metadata. Similar to the read performance analysis, collective metadata I/O delivers better performance on the small writes about the descriptive metadata than the default option. This advantage is further improved when the entire metadata is placed at the beginning of the file.

Next, we evaluate the operations on the metadata-cache management in HDF5 write. Specifically, we alternate the operations on the metadata cache flushing together with the other HDF5 configurations and measure the end-to-end performance variations across the number of datasets, burst sizes and read/write scales. Figure 6(c) reports the aggregate write performance of $64MB$ bursts on 128 nodes, representing the results of all bursts across all scales. It shows that the HDF5 write benefits from the write cache flushing at file close.

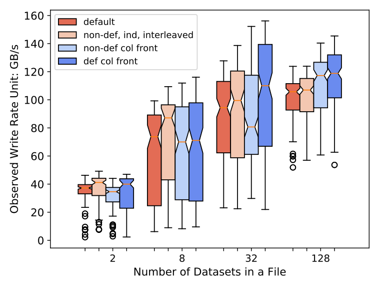
In summary, we conclude that enabling collective I/O on file metadata write, placing the metadata at the file start, and flushing the metadata cache at file close, delivers the best HDF5 write performance. Moreover, placing the file metadata at the file start and using collective I/O on metadata read operations provides optimized HDF5 read performance.



(a) **CDFs of the write cost on the descriptive metadata**



(b) **CDFs of the read cost on the descriptive metadata**



(c) **Aggregate write bandwidth** observed from $64MB$ dataset writes on 128 nodes

Fig. 6: CDFs of the write and read cost on the descriptive metadata (6(a) and 6(b)) and **Aggregate write bandwidth** observed from 64MB dataset on 128 nodes (6(c)). In Figures 6(a) and 6(b), each line presents the metadata cost for a number of datasets with two file-metadata configurations: 1. using default or collective metadata I/O, 2. being placed at file start or interleaved with datasets. Moreover, we combine the descriptive metadata performance across various burst sizes as different sizes do not affect the performance of descriptive metadata. In Figure 6(c), for each number of datasets, we report the HDF5 performance with three file-metadata configurations: 1. defer the metadata cache flushing at file close or not, 2. choose default or collective metadata I/O, 3. place the metadata at file start or interleaved with datasets.

IV. Evaluation with Tuned Parameters

We experimented on two production systems (§II-B), and measured the tuned HDF5 performance on three representative I/O kernels extracted from production codes (§IV-A) and on the extensive I/O patterns generated by IOR (§IV-B). For the experiments on Summit/Alpine, we compare the performance of the default and the tuned HDF5. For the experiments on Cori/Lustre, we evaluate the performance of three settings: the default HDF5 with the NERSC-recommended Lustre striping, the tuned HDF5 with the NERSC-recommended Lustre striping, and the tuned HDF5 with the tuned Lustre striping. Here, we set the tuned Lustre striping as *stripe size*=16MB and *stripe count*=128 in consideration of the Alpine GPFS striping.

A. Evaluation with Scientific Workloads

We ran the experiments with three I/O kernels. For each kernel on a target system, we configured the kernel to read/write multiple timesteps per run. On Summit, we use 3,072 (512×6) compute cores from 512 compute nodes and with 6 cores per node; for Cori, we use 4,096 (512×8) compute cores from 512 compute nodes with 8 cores per node.

1) Scientific Workloads: **VPIC-IO kernel**[2] is an I/O benchmark extracted from VPIC [18], a plasma physics code. It presents the data structures of a space-weather simulation in a multi-dimensional space. In this kernel, each MPI rank writes 256MB data to a single shared HDF5 file for 8M (8×2^{20}) particles. In each experimental run, VPIC-IO kernel generates 2.3TB and 3TB data on Summit and Cori, respectively.

BD-CATS-IO[2] is an I/O kernel that represents the I/O read patterns used to analyze the particle data produced by applications such as VPIC and Nyx [19]. In BD-CATS-IO, each MPI rank reads a different part of each dataset in an HDF5 file. In our experiments, each run reads the data generated by VPIC-IO, i.e., 2.3TB data on Summit and 3TB data on Cori.

AMReX Benchmark[3] is an I/O benchmark developed to evaluate the write performance of the I/O patterns generated by the adaptive mesh refinement (AMR) codes [20]. Different from the regular write patterns as in VPIC-IO, AMR data patterns are more dynamic: in an AMReX run, different MPI ranks generate the data with different sizes in a timestep and the total data size per timestep varies across timesteps. The benchmark generates two patterns for single and multiple mesh refinement levels.In our experiments on the target systems, each AMReX run produces $1.1TB$ (single level, 3 timesteps) and $2.7TB$ (multiple levels, 5 timesteps) of data in total.

2) Performance Analysis: In Figures 7(a) and 7(d), we show the benchmark execution time of the three kernels on Summit and Cori, respectively. It is clear that, the tuned HDF5 delivers 1.1×—12× speedup on the mean performance across the production codes on the target systems, showing the effectiveness and generality of our tuned parameters in HDF5.

In particular, VPIC-IO benefits most from the tuned HDF5 and shows 12× and 7× (with tuned Lustre) speedup on Summit and Cori, respectively. Because the 256MB per-core data in size is a multiple of 16MB full blocks (which is equal to the tuned block size on Summit's GPFS and Cori's Lustre stripe size), maximum benefit comes from the perfect data-layout alignment between the tuned HDF5 and the underlying file systems. Comparatively, we observed less speedup in the AMReX runs, suggesting that when the write bursts include non-full blocks and these blocks cannot utilize the bandwidth of the resources efficiently, and leading to the reduced end-to-end benefit accordingly.

Moreover, on Cori, the runs with the tuned HDF5 and tuned Lustre deliver the best performance for all three applications. Relative to the fact that each code reads/writes >200MB per-core data, we conclude that, large reads/writes obtain better performance when *stripe size* and *stripe count* set larger. It also indicates that, for Lustre file systems, a fine-tuned striping may

[2]https://sdm.lbl.gov/exahdf5/ascr/software.html

[3]https://github.com/AMReX-Codes/amrex/tree/development/Tests/HDF5Benchmark

deliver even better performance than the default configuration does. We leave additional study as future work.

B. Evaluations with IOR

We generated I/O patterns using the IOR template proposed in the file-metadata experiment (§III-D). On the target systems, we use 512 compute nodes with M cores per node and read/write D datasets in an HDF5 file. In each pattern, the $512\times M$ MPI ranks each read/write a W-byte different part in each of the D datasets; the aggregate data size of a file is $512\times M\times W\times D$. We generate I/O patterns with the randomly-chosen file sizes between 0.1GB and 1TB and the randomly-chosen number of datasets (D) between 1 and 200. We set the randomly-chosen number of cores (M) between 1 and 16 on Summit and set M=1 on Cori.

We ran each of these patterns repetitively, between at least 3 times (up to 6 times in a few cases) on each system to show performance variance. For each pattern on a target system, we summarize the mean and the max of the read/write performance for each HDF5 settings and normalize it to the corresponding measure on the pattern with the default HDF5.

In Figure 7(b), we report the read performance on Summit. Specifically, with the tuned HDF5, 73% and 85% of the patterns deliver $1.5\times$—$57\times$ speedup on the mean and max of the read performance. We separate the time consumption in HDF5 read internals and find that, for a pattern, the cost on file metadata takes up to 58% of the total read time and the tuned HDF5 outperforms the default on this metadata $30\times$—$100\times$. It is clear that the tuned file-metadata configurations contribute most to this performance improvement.

Figure 7(c) shows the write performance on Summit. With the tuned HDF5, 77.5% of the patterns deliver $1.03\times$—$1.8\times$ speedup on the mean write performance, suggesting the consistent improvement of the tuned HDF5. However, we also noticed that, on some patterns, the tuned HDF5 performs worse than the default and it becomes more noticeable when we compare the max performance. We look into the HDF5 write internals and find that, the tuned file-metadata configurations still achieve $42\times$—$103\times$ better performance on the metadata write, but the achievement is less than 7% of the total write times. Moreover, the results also suggest that, for small writes, the performance improvement from the alignment setting is limited. In consideration of the highly variable environment and the limited repetitions per pattern, we conclude that, small writes can benefit from the tuned HDF5 but its performance is also largely determined by the system conditions.

Figure 7(e) reports the read performance on Cori. It suggests that the tuned HDF5 plus the default Lustre performs the best. Specifically, 92.6% and 84.3% of the patterns with this HDF5 delivers $1.05\times$—$17.3\times$ and 1.04—$14.2\times$ speedup on the mean and max performance, respectively. We look into the HDF5 read internals and find that, the cost of file metadata in the default HDF5 settings ranges in 20%—67% of the total read time, and with the tuned HDF5, this cost is reduced to less than 5%. It suggests that, the tuned file-metadata configurations contribute most to good performance.

Figure 7(f) reports the write performance on Cori, showing that the tuned HDF5 and Lustre delivers the best performance. In particular, 90.3% and 87% of the patterns with this HDF5 reaches $1.2\times$–$44\times$ and $1.2\times$–$39\times$ speedup, respectively. We look into the write internals and find that the write performance is dominated by the dataset write. Clearly, the appropriate alignment setting plus the large *stripe size* and *stripe count* maximize the write performance on the Cori's Lustre.

In summary, we conclude that for small and medium I/O, our tuned HDF5 delivers good performance across I/O patterns and file systems.

V. Related Work

First, researchers proposed various optimization mechanisms in various I/O libraries. Among the HDF5 tuning efforts, Byna et al. [7] work on tuning configurations with the HDF5 sub-filing feature and initiates parallel writes to multiple HDF5 files instead of a single-shared one. Howison et al. [8] tune the HDF5 performance on Lustre file systems with chunked dataset alignment setting and metadata flush. Several studies explored tuning of various parameters using genetic algorithms and performance modeling [9], [10], [11].

Second, at the MPI-IO layer, ROMIO [21] introduced data sieving and collective I/O, to improve the I/O performance that allows users to configure various "hints" of the techniques to tune the performance. Liao et al. [22] proposed several improvements to the ROMIO collective buffering algorithm.

Third, parallel file systems such as Lustre allows users to customize their data layout on storage devices. Different layout policies can lead to different I/O performance [13], [16]. For instance, Yu et al. [23] characterized, tuned, and optimized I/O performance on a Lustre file system. Similarly, Byna et al. [24] showed the performance improvements with properly set Lustre parameters matching the access patterns of VPIC.

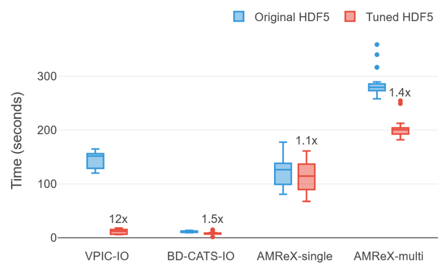
Compared to these tuning studies that focused on optimizing specific I/O benchmarks, our study explored setting the default parameters by using benchmark patterns on a system and obtained tuned parameters that can be set as new defaults for HDF5 on different systems.

VI. Conclusion

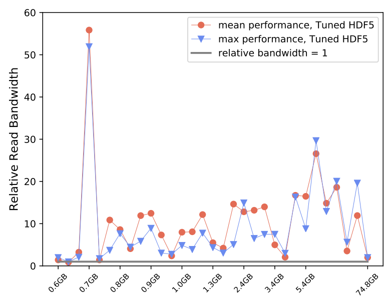
To find tuning parameters that provide high I/O performance with HDF5, we have conducted a benchmarking analysis with controlled experiments to profile the I/O behavior under production loads Using this approach, we identified a set of tuned default parameters for HDF5 and demonstrated these defaults achieve significant performance benefits for various I/O patterns on on multiple HPC systems. We plan to extend this work towards achieving automatic and runtime tuning by using file system load and other I/O software parameters.

Acknowledgment

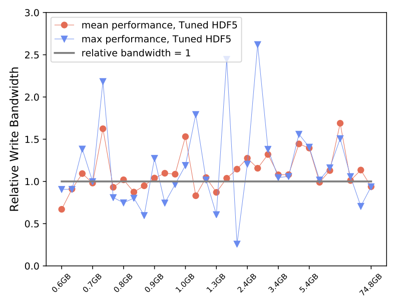
This research is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of



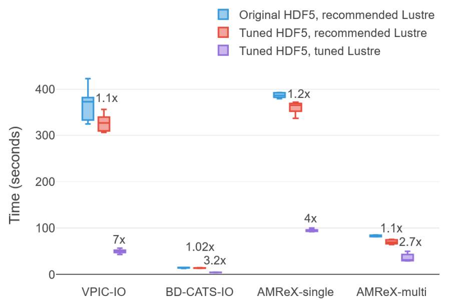
(a) Execution time with 512 nodes on Summit



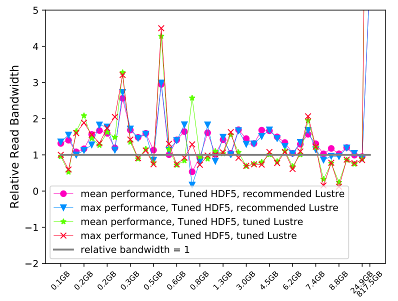
(b) Relative read performance on Summit



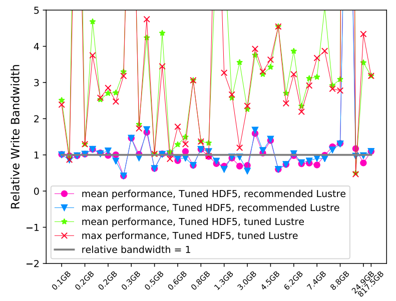
(c) Relative write performance on Summit



(d) Execution time with 512 nodes on Cori



(e) Relative read performance on Cori



(f) Relative write performance on Cori

Fig. 7: Performance comparison with different I/O kernels and IOR runs on Summit (top three) and Cori (bottom three)

the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 and resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] The HDF Group, "HDF5," https://www.hdfgroup.org/solutions/hdf5/.

[2] ADIOS team at ORNL, "The Adaptable I/O System," https://csmd.ornl.gov/adios.

[3] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *SC*, 2003, pp. 39–39.

[4] P. Braam, "The Lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[5] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *FAST*, vol. 2, no. 19, 2002.

[6] "Automatic Library Tracking Database at NERSC," https://sdm.lbl.gov/exahdf5/papers/201810-HDF5-Usage.pdf, accessed: 2019-4-28.

[7] S. Byna, M. Chaarawi, Q. Koziol, J. Mainzer, and F. Willmore, "Tuning HDF5 subfiling performance on parallel file systems," *CUG*, 2017.

[8] M. Howison, "Tuning HDF5 for Lustre File Systems," *IASDS*, 2010.

[9] 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 *SC*, 2013.

[10] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel i/o autotuning with performance modeling," in *HPDC*, 2014, p. 253–256.

[11] B. Behzad, S. Byna, and M. Snir, "Optimizing I/O performance of HPC applications with autotuning," *ACM Transactions on Parallel Computing*, vol. 5, no. 4, pp. 1–27, 2019.

[12] NERSC Documentation, "Lustre Striping on Cori Scratch," https://docs.nersc.gov/performance/io/lustre/.

[13] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *SC*, 2012.

[14] B. Xie, Y. Huang, J. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting output performance of a petascale supercomputer," in *HPDC'17*, 2017.

[15] B. Xie, J. Chase, D. Dillow, S. Klasky, J. Lofstead, S. Oral, and N. Podhorszki, "Output performance study on a production petascale filesystem," in *HPC-IODC'17*, 2017.

[16] B. Xie, S. Oral, C. Zimmer, J. Y. Choi, D. Dillow, S. Klasky, J. Lofstead, N. Podhorszki, and J. S. Chase, "Characterizing output bottlenecks of a production supercomputer: Analysis and implications," *ACM Transactions on Storage*, 2020.

[17] B. Xie, Z. Tan, P. Carns, J. Chase, K. Harms, J. Lofstead, S. Oral, S. S. Vazhkudai, and F. Wang, "Interpreting write performance of supercomputer i/o systems with regression models," in *IPDPS'21*, 2021.

[18] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in petascale kinetic plasma simulation with vpic and roadrunner," in *Journal of Physics: Conference Series*, vol. 180, no. 1, 2009, p. 012055.

[19] 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.

[20] 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," *Journal of Open Source Software*, vol. 4, no. 37, pp. 1370–1370, 2019.

[21] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Frontiers*, 1999, pp. 182–189.

[22] W.-k. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *SC*, 2008, pp. 1–12.

[23] W. Yu, J. S. Vetter, and H. S. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *IPDPS*, 2008, pp. 1–11.

[24] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol, "Tuning parallel I/O on Blue Waters for writing 10 trillion particles," *CUG*, 2015.