h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns

Tonglin Li, Suren Byna, Quincey Koziol, Houjun Tang, Jean Luca Bez, and Qiao Kang
Lawrence Berkeley National Laboratory, Berkeley, CA 94720
Email: tonglinli@lbl.gov, SByna@lbl.gov, koziol@lbl.gov, htang4@lbl.gov, jlbez@lbl.gov, qkang@lbl.gov

***Abstract*—Parallel I/O is a critical technique for moving data between compute and storage subsystems of supercomputing systems. With massive amounts of data being produced or consumed by compute nodes, high performant parallel I/O is essential. I/O benchmarks play an important role in this process, however, there is a scarcity of I/O benchmarks that are representative of current workloads on HPC systems. Towards creating representative I/O kernels from real world applications, we have created `h5bench` a set of I/O kernels that exercise HDF5 I/O on parallel file systems in numerous dimensions. Our focus on HDF5 is because of the parallel I/O library's heavy usage in a wide variety of scientific applications running on supercomputing systems. The various dimensions of `h5bench` include I/O operations (read and write), data locality (arrays of basic data types and arrays of structures), array dimensionality (1D arrays, 2D meshes, 3D cubes) and I/O modes (synchronous and asynchronous). In this paper, we present the observed performance of `h5bench` executed along several of these dimensions on a Cray system: Cori at NERSC using both the DataWarp burst buffer and a Lustre file system and Summit at Oak Ridge Leadership Computing Facility (OLCF) using a SpectrumScale file system. These performance measurements are using find performance bottlenecks, identify root causes of any poor performance, and optimize I/O performance. As the I/O patterns of `h5bench` are diverse and capture the I/O behaviors of various HPC applications, this study will be helpful not only to the CUG community but also to the broader supercomputing community.**

***Keywords*—HDF5 benchmarks,**

I. Introduction

Applications using high-performance computing (HPC) resources are highly dependent on storing data to and retrieving previously stored data from file systems. I/O libraries such as HDF5 [1], [2], netCDF [3], ROOT [4], etc. play a critical role in providing access to and from file systems. Of these, parallel I/O libraries, such as HDF5 and PnetCDF [5], and MPI-IO [6], which allow multiple processes or ranks from MPI programs access data concurrently from file systems need to be efficient.

Parallel I/O benchmarks that are representative of real world HPC applications play an important role in evaluating I/O libraries and removing any performance bottlenecks. Several parallel I/O benchmarks are available that measure performance of parallel file systems with various I/O libraries.

IOR[1] is the most popular parallel I/O benchmark used for testing the performance of parallel file systems [7]. IOR allows usage of multiple file access patterns (read and write, single shared file, and file per process) and multiple I/O library interfaces (POSIX, MPI-IO, and HDF5). IOR also provides various distributed computing interfaces, such as S3, HDFS, etc. MDtest, now integrated to IOR, is often used for evaluating the metadata performance of POSIX-compliant parallel file systems. In specific, MDtest evaluates the performance of creation, stat, and removal of files, directories, and hierarchy of directories of a given depth.

While IOR can be configured to represent a significant number of file access patterns, there is still a need for I/O kernels that are representative of real world applications that use high-level I/O libraries, such as HDF5. These I/O kernels would be helpful to represent not only various in memory data models and file access patterns, but also to test new features and to identify performance inefficiencies. For instance, HDF5 has recently developed asynchronous I/O feature that overlaps I/O overhead with computation phases, which is unexplored by existing I/O benchmarks [8], [9]. Similarly, performance evaluation of compression in I/O libraries is another unexplored area. In the metadata performance evaluation, MDtest's focus is on file system performance in operating with files and directories. Self-describing formats, such as HDF5, also have user-defined metadata to be added to describe data. Evaluation and optimization of user-level metadata access costs is another important requirement for parallel I/O kernels.

In this paper, we focus on the HDF5 API and parallel I/O library due to its heavy usage at supercomputing systems [10]. A few HDF5 benchmarks exist to represent application-specific I/O patterns that support different I/O interfaces. For instance, MACSio (Multi-purpose, Application-Centric, Scalable I/O Proxy Application) [11] provides various kernels that test and evaluate I/O performance in different data models. MACSio allows using different I/O library interface plugins in addition to HDF5, such as PDB, TyphonIO, Exodus, etc. and parallel I/O patterns. While MACSio is covering a significant number of patterns, new features such as asynchronous I/O needs further benchmarking. We have previously developed Parallel I/O Kernel (PIOK) suite that included simple HDF5 I/O operations such as reads and writes with basic array data types. There are a number I/O patterns that are not covered by PIOK, such as multi-dimensionality of the array data, asynchronous I/O, etc. There are a few HDF5 I/O benchmarks that are based on application I/O, such as FLASH-IO [12], ChomboIO [13], and AMReX [14]. These benchmarks are specific to the patterns of the applications and there is still a need for benchmarks that can exercise various configurable

[1]IOR: `https://github.com/hpc/ior`

options that are generic to cover a variety of patterns and the latest I/O optimization features.

In our effort, we focus on bringing together a set of HDF5 benchmarks, called `h5bench` and making them available for a broader audience. We focus in this paper on discussing basic I/O kernels in different dimensions including I/O operations (read, write, and HDF5 metadata), data locality (arrays of basic data types and arrays of structure representations both in memory and in file), array dimensionality (1D arrays, 2D meshes, 3D cubes), and different I/O modes (synchronous and asynchronous). We focus on HDF5 in this work based on the library's heavy usage on HPC systems. Defining HDF5 benchmarks that are representative of HDF5 applications and tuning them on HPC systems will benefit broadly.

We evaluate these different dimensions of read and write kernels on Cori (a Cray XC40 system at The National Energy Research Scientific Computing Center (NERSC)) and on Summit (an IBM system at The Oak Ridge Leadership Computing Facility (OLCF)). On Cori, we use a Lustre parallel file system that uses hard disk drives (HDDs), and a DataWarp [15] file system on solid state drives (SSDs). We evaluate the performance of `h5bench` by comparing synchronous and asynchronous read and write operations on Cori and Summit. We study the impact of data locality (contiguity of data) in memory and in file, using various dimensions for arrays. On Cori, we compare read and write performance of Lustre and the DataWarp Burst Buffer. We also show improved performance of asynchronous I/O in the case of reading data when I/O phases are fully overlapped with emulated computation time compared to partially overlapped I/O phases.

The remainder of the paper is organized as follows. In Section II, we briefly introduce HDF5 and asynchronous I/O with HDF5. In Section III, we describe different modes of the basic I/O operations introduced in `h5bench`. In the following sections (§IV and §V), we describe experimental setup and performance evaluation, respectively. In Section VI, we conclude the paper with a brief discussion of future work.

II. Background

A. A brief introduction to HDF5

HDF5 (Hierarchical Data Format Version 5) is a self-describing file format and an I/O library [1] that provides flexibility, extendibility, and portability. It supports a variety of data structures across science domains and stores data and the corresponding metadata (e.g., data type, data size) within a single HDF5 file. HDF5 relieves the user from manual file management such as file space allocation and seeking specific offsets to access data.

Due to its longevity and robustness, HDF5 is used widely in many science domains as a *de facto* standard to manage various data models and is used for efficient parallel I/O in HPC simulation and analysis applications. For instance, HDF5 is among the top five libraries loaded by applications at NERSC and OLCF [10]. Because of this popularity, ensuring HDF5's parallel I/O performance is efficient is critical for HPC facilities. Providing benchmarks and tuning I/O patterns in them paves the path for achieving the goal of efficiency and good overall performance for applications.

B. Asynchronous I/O with HDF5

Asynchronous I/O allows overlapping the I/O time with computation and communication, which can significantly reduce the overall application runtime. Scientific applications with interleaved computation and I/O phases may observe their I/O time partially or fully hidden with asynchronous I/O. HDF5 provides the Virtual Object Layer (VOL) [16] that allows intercepting the HDF5 I/O routines and applying optimizations for better data management, and is transparent to the application. An asynchronous I/O VOL connector [8] is developed to enable asynchronous I/O for HDF5 operations using background threads. This implementation can be compiled as a dynamically linked library (DLL) and can be linked to a user's application directly, remaining separate from the installed version of HDF5 and making it easy to adopt. The background threads are managed by Argobots, a lightweight low-level threading framework [17].

There are two modes of asynchronous I/O in HDF5: implicit and explicit mode. The implicit mode needs minimal code changes but has performance limitations (e.g., all read operations are synchronous), and the user can enable it by running the application with a few environment variables set. The explicit mode requires some code changes such as replacing the HDF5 APIs with the EventSet APIs; it gives more control to applications over when to execute asynchronous operations and a better mechanism for detecting errors. More details of the asynchronous I/O implementation in HDF5 are available in [8], [9].

III. h5bench I/O kernel suite

With the goal of providing a comprehensive set of HDF5 I/O kernels that are representative of applications using the HDF5 API and of tuning their I/O performance using novel features introduced in HDF5, we developed `h5bench`[2].

In the current release of `h5bench`, we provide a set of read and write kernels. We started the development of `h5bench` by taking previously available I/O kernels for writing (called VPIC-IO [18]) and for reading (called BD-CATS-IO [19]). VPIC-IO was originally derived from a plasma physics simulation that was designed to understand magnetic reconnection phenomena, which often occurs in space weather events such as solar flares interacting with the Earth's magnetosphere [20]. BD-CATS-IO was derived from a parallel DBSCAN algorithm's reading particle data, such as that generated by VPIC [21] or Nyx [22] simulations. Both these write and read patterns are simple in the data structures in memory and in file. In VPIC, we recently implemented a new particle and file data write strategies that writes data from a user-defined data structure (similar to `struct` in `C` or a "compound data structure" in HDF5) form either 1-dimensional arrays or "compound data structure" in HDF5. This data structure

[2] `h5bench`: `https://github.com/hpc-io/h5bench`

is also commonly referred to as "array of structures" in literature. In h5bench, we added these new memory buffers and in file layout to the original VPIC-IO kernel. In addition to the locality in memory and in file, we then expanded the write and read benchmarks to add multiple I/O modes and patterns – such as asynchronous I/O, multi-dimensional arrays, file system-specific configurations, and MPI-IO specific configurations. In h5bench, we provide configurable options that users can provide to exercise various I/O patterns. We describe each of these configurations in the following.

The h5bench benchmark suite assumes simulation or analysis done in many time steps with multiple subsequent computation and I/O phases. This is a typical pattern in physics simulations that performs number crunching over a large number of time steps that emulates a physical phenomenon under study [23]–[25]. The state of a simulation is written to storage frequently either to study the progression of the simulation or to checkpoint for handling any failures. For instance, the VPIC simulation studying the magnetic reconnection phenomenon [20] computes ≈20,000 time steps and dumps the data ≈2,000 time steps, i.e., a total of 10 time steps. In the benchmark, we do not use real computations, hence, use `sleep ()` to emulate the computation time. The produced data in the write benchmarks is random using the current time as the seed. The read benchmarks use the data written using the write benchmarks and emulate data analysis time using `sleep ()` function. A user can specify the emulated compute time in the configuration file to be used by the write and read benchmarks.

The h5bench benchmark reports the total emulated compute time and data size read/written, which are set by users. The performance metrics reported by h5bench include data preparation time (i.e., time to initialize memory buffers in the case of the write benchmarks), metadata read or write times, other file operations (create, flush, and close times), raw read or write times, raw read or write rates, and observed read/write times, and observed read/write rates. The raw rates are the wall clock times for performing a read or write operations. These times are obtained by measuring the elapsed times of `H5Dwrite` and `H5Dread` functions. In case of asynchronous I/O operations, these times are measured, however, because this time is overlapped with the emulated compute time, the overall benchmark runtime does not observe this time. The observed times only report the time that the overall benchmark observes. The I/O rates (raw and observed) are calculated as the ratio of the size of data and the times (raw and observed, respectively). The rates are currently reported as MB/s, but we will be changing them to GB/s or KB/s depending on the sizes of the data.

A. I/O modes

The h5bench currently supports write and read operations with various patterns. The benchmark defines HDF5 datasets and writes or reads metadata of the datasets. The metadata includes the dimensions of the datasets and their names. The basic HDF5 write and read APIs we use are `H5Dwrite` and `H5Dread`, that write to or read from HDF5 datasets, respectively. Both write and read benchmarks currently use weak scaling, where the number of particles per rank is specified by the user in a configuration file. As the number of the MPI ranks increase, the data size increases.

The ranks select non-overlapping partitions of the HDF5 datasets using hyperslab selections in HDF5. In h5bench, we support three types of hyperslab selections: contiguous, strided, and partial. In the contiguous access pattern, each MPI rank accesses a single contiguous partition of the data. In the strided pattern, hyperslabs selections have gaps with a stride that is set in the configuration file. The strides can be in multiple dimensions as well. The partial pattern only applies to read operation, where only a subset of data is read by analysis applications. Analysis applications often read a subset of data [19], [26]. For example, in big data clustering [27], the analysis application reads the top $k\%$ of the particle data to analyze, where k is variable. In [26], patterns include reading slides of 1D, 2D, or 3D datasets. In h5bench, we currently support the first $k\%$ data elements in arrays. We plan to expand this functionality in future to reading arbitrary slides in different dimensions using offsets in defining HDF5 hyperslabs.

We support three modes of reads and writes: synchronous, implicit asynchronous, and explicit asynchronous. As explained briefly in Section II-B, HDF5 supports asynchronous I/O in two ways – implicit and explicit. In the synchronous mode, at the completion of a computation phase, the I/O phase starts. The subsequent computation phase starts after the I/O phase is complete. With both asynchronous I/O modes, the subsequent computation phase starts after the I/O operations are posted to the background threads and without waiting for the data to be written to or read from a file on the storage device. The background threads can use memory or node-local storage to cache the data. In h5bench, a user can set a limit on the memory that the background threads can use as a cache.

B. Locality

The data structures applications use in memory and the data layout in files on file systems a have significant impact on performance. To evaluate the performance of these data structures, we provide contiguity of data structures in memory and in files. The contiguous in memory pattern means that there are multiple arrays that are of the same basic data types such as integer, float, double, etc. The non-contiguous pattern in memory is also known as "array of structures" or "derived data type", where the datatype is derived from basic data types. HDF5 allows storing arrays of basic data types and derived data types. In storing basic data types each array is stored as a dataset. The derived data types are stored as a "compound datatype" dataset. One can construct a compound datatype in HDF5 by using `H5Tcreate` for each basic data type in the derived data type. We show these four basic patterns in Figure 1, where we use "contiguous" to represent arrays in memory and HDF5 datasets in file that are of basic data types. We use "compound" to describe array of structures in memory and compound datatypes in HDF5 files.

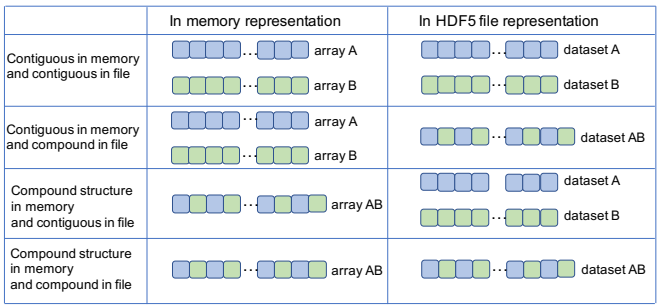

Figure 1: In memory data structure and in HDF5 file data layout mappings. For simplicity, we illustrate data structures as 1-dimensional arrays. Arrays are stored as HDF5 datasets. We do not depict the metadata (attributes of the datasets).

C. Additional options

Dimensions of arrays: In scientific datasets, multidimensional arrays are the common data structures. To support the dimensionality, `h5bench` supports reading and writing 1D, 2D, and 3D datasets. A user can specify the dimensions in the configuration file.

File locations: The HDF5 files can be written to various POSIX-based parallel file systems, such as Lustre, GPFS, and DataWarp. Users can set the location of the files in the configuration file. We have applied file system specific optimizations in `h5bench`. For instance, we set the alignment for HDF5 file requests, that align a block of data to file system block. This optimization benefits I/O performance on GPFS. For Lustre, we are currently relying on users to set the stripe size and stripe count using `lfs setstripe` command. In future work, we plan to set these from the benchmark.

MPI-IO Options: MPI-IO provides various optimizations to improve performance. In `h5bench`, we allow turning on or off the collective buffering [6]. In future work, we plan on providing support for setting the collective buffer size, etc.

IV. Experimental Setup

We evaluated `h5bench` on Cori at NERSC and Summit at Oak Ridge Leadership Computing Facility (OLCF).

A. Cori

Cori located at NERSC is a Cray XC40 system with a peak peerformance of 30PF. It contains two main computation partitions, i.e., Haswell and KNL. In all our tests in this paper, we used the Haswell partition with 2,388 compute nodes. Each Haswell node has two sockets, and each socket has a 16-core Haswell processor. Each core supports 2 hyper-threads and each node has 128 GB DDR4 memory shared by the two sockets.

Lustre scratch: Cori provides a 30 PB Lustre file system as temporary scratch space for files. The file system has a peak performance of 700 GB/s I/O bandwidth. The Lustre file system is equipped with 248 Object Storage Targets (OSTs), with a default striping settings of 1 MB stripe size and 1 OST as stripe width. Users can change the striping on Lustre using `lfs setstripe` command with options for stripe size and stripe width.

SSD-based burst buffer: NERSC provides an SSD-based burst buffer that uses Cray DataWarp. The peak bandwidth of the burst buffer is 1.7 TB/s, where each burst buffer node (server) has 6.5 GB/s. To request the burst buffer, a user has to request the required capacity. Each 20 GB capacity request provides one burst buffer server. For instance, a request for 100 GB of burst buffer allocates 10 burst buffer servers.

B. Summit

The Summit supercomputer deployed at the Oak Ridge Leadership Computing Facility (OLCF) is based upon the IBM AC922 system. It is comprised of 4,608 compute nodes, each equipped with 2 IBM POWER9 (P9) processors and 6 NVIDIA Tesla V100 (Volta) GPUs. Also, each node has 512 GB of DDR4 CPU memory, and each GPU has 16 GB of HBM2 memory. An NVLink 2.0 bus connects each P9 CPU to 3 V100 GPUs. An InfiniBand EDR network with a fat-tree topology connects the nodes. A 1.6 TB NVMe device is present on each compute node to be used as node-local storage.

Summit's compute nodes are connected to the central-wide Alpine parallel file system, a 250 PB IBM Spectrum Scale (GPFS) deployment. The file system of Summit offers a peak performance of 2.5 TB/s for sequential I/O.

C. `h5bench` configurations

In this work, we test the configurations of `h5bench` shown in Table I to demonstrate a sample of the capabilities of the benchmark suite. On Cori, we used 16 MPI ranks per node, and on Summit, 32 MPI ranks per node for these runs. On Cori Lustre, we used a stripe count of 244 and a stripe size of 16 MB. On the burst buffer of Cori, we requested 8 TB, which allocates all available 270 burst buffer servers. On Summit's GPFS, we have set HDF5 alignment of 16 MB, which is equal to the block size of GPFS. In HDF5, the `H5Pset_alignment` call sets the properties of a file access to allow any file object larger than a given threshold to be aligned on an offset address on file system that is a multiple of the set alignment size. Because the block size of GPFS in Summit is 16 MB, we set the alignment property with 16 MB for file objects that are greater than 4KB. Each benchmark run also uses five iterations of compute and I/O phases, where the compute phase is emulated with `sleep` functions. We used up to 15 seconds of emulated compute phase. We varied this emulated compute time to overlap the I/O latency efficiently. The data related to each phase are organized in a different HDF5 group in the same file. For asynchronous I/O mode, we have used "explicit" I/O mode, where the benchmark is modified to use the HDF5 event sets that give more control to users when to perform the I/O operations using background threads. Users can also use "implicit" I/O mode, which does not require code modifications, and asynchronous I/O mode is enabled by setting environment variables. One can try many more combinations of configurations for the benchmark suite.

IO operations	IO Modes	Scale (powers of 2)	Platforms
Write	Sync I/O and Async I/O	16 to 2048	Cori and Summit
Read	Sync I/O and Async I/O	16 to 2048	Cori and Summit
Read	Full and Partial	16 to 2048	Cori
Write	Locality	256 to 2048	Cori
Write	Array dimensions	16 to 2048	Cori and Summit
Read	Array dimensions	16 to 2048	Cori and Summit
Write and read	Burst buffer vs. Lustre	16 to 2048	Cori

Table I: Experimental setup used in this paper.

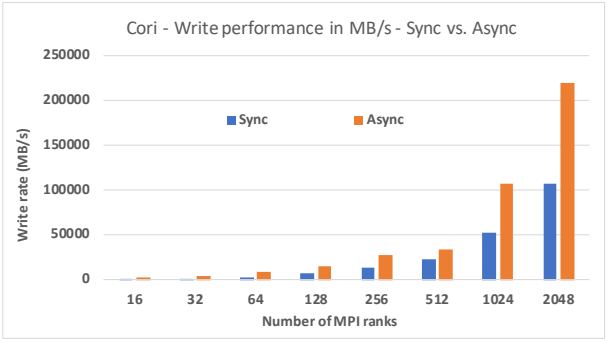

Figure 2: A comparison of synchronous and asynchronous write performance on Cori with varying number of MPI ranks. We used 16 cores per compute node.

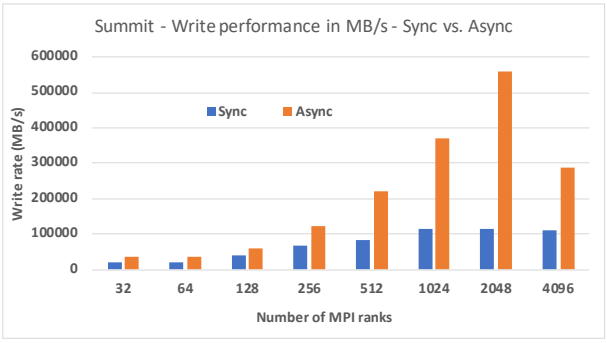

Figure 3: A comparison of synchronous and asynchronous write performance on Summit with varying number of MPI ranks. We used 32 cores per compute node.

V. Performance Evaluation

In evaluating `h5bench`, we ran each configuration at least three times and report the best performance. Since the three runs of a configuration were executed consecutively, the performance reported may have some variability due to the jobs running concurrently and sharing the file system. We report the *observed I/O rate*, which is the ratio of the amount of data written or read and the sum of the wall clock time taken to perform all the write or read operations, respectively. In the plots, we show the observed I/O rate in MB/s. In a few plots, we also show the *elapsed time* in performing an I/O operation.

A. Write performance, scalability, and asynchronous I/O

In Figure 2, we compare the observed I/O rate for writing data in asynchronous and synchronous I/O modes on Cori, and in Figure 3, the rate for the same operations on Summit. We used a contiguous pattern in memory and in file, where the 8 1D arrays are written as 8 HDF5 datasets. As can be seen, the observed write rate of asynchronous I/O mode is significantly higher than that of synchronous I/O mode. This is achieved by overlapping write operations while the benchmark's computation phase (using an emulated compute time by applying the `sleep()` functions). As the number of MPI ranks increase, the benchmark achieved higher write rates. At the scale of 2K MPI ranks, the observed asynchronous I/O rate is ≈220 GB/s on Cori and ≈560 GB/s on Summit. One abnormality we observed is on Summit at the scale of 4K cores, where the I/O rated dropped down to ≈290 GB/s. This may be due to interference in the file system during the time these jobs ran. However, we need to investigate further if there are any other potential reasons for this performance drop.

B. Read performance, scalability, and asynchronous I/O

In Figures 4 and 5, we compare the performance of the read benchmark that was run on Cori and on Summit, respectively, with synchronous and asynchronous I/O modes. We used the same data we wrote by the write benchmark (from Section V-A), which are contiguous 1D arrays from HDF5 datasets. To avoid caching effects of reading, the read benchmark was also run on different days after the write benchmark's generation of the files. We also used 15 seconds of emulated compute time between consecutive time steps in these runs to overlap the read time completely with the (emulated) compute phase. With asynchronous I/O, at 2K cores, the observed I/O rate is ≈1TB/s on Cori and ≈700 GB/s on Summit. On Cori, although the peak I/O bandwidth is 700 GB/s, the observed I/O rate is much higher because the elapsed time by the benchmark is overlapped by the computation phase. In ideal scenarios, if the I/O phase is completely overlapped with the computation phase, the observed I/O rate can be *infinite*. However, in the write tests, data for the last time step to be written to file does not have a computation phase to follow. Hence, the file write phase is not overlapped with computation. Similarly, for the first read phase in the read tests does not precede a computation phase and there was no overlapping. Because of these non-overlapped I/O phases, the observed I/O rates are not infinite, but high. Another observation is that in the read case, we do not see a performance drop from 2K to 4K MPI ranks on Summit.

When data analysis applications read data, it is commonplace to read a partial amount of data or slices of data instead of the entire data [19], [26]. As mentioned earlier, a big data clustering application [27] reads the top $k\%$ of the particle data to analyze, where k is variable. We mimic this pattern by

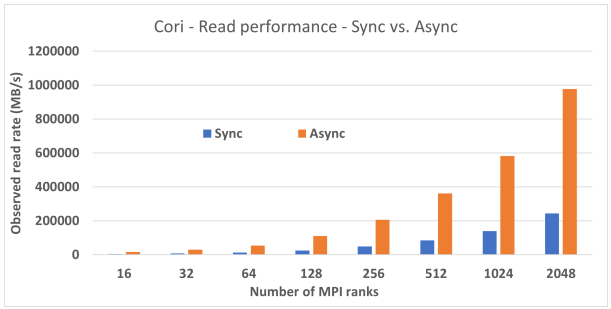

Figure 4: Cori - read - sync vs. async

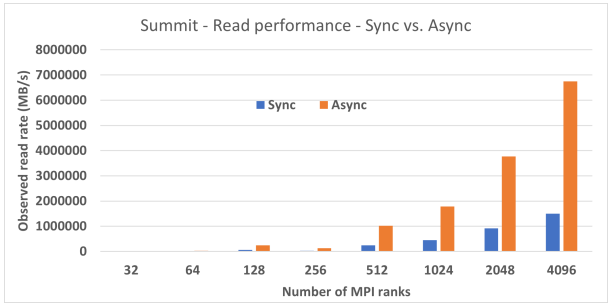

Figure 5: Summit - read - sync vs. async

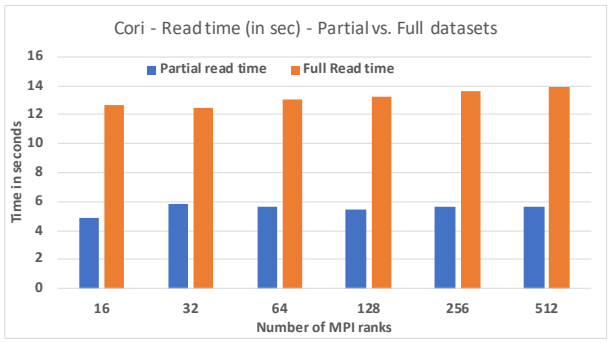

Figure 6: A comparison of observed time to read the entire dataset and read 10% of the dataset by each MPI rank on Cori.

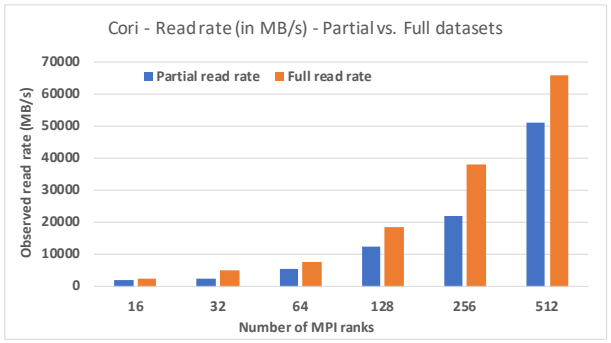

Figure 7: A comparison of observed I/O rate for reading the entire dataset and reading 10% of the dataset by each MPI rank on Cori.

reading the top 10% of the 1D array data written by `h5bench` write benchmark and comparing it with reading the entire data. In both, "full read" and "partial read", cases, we partition the entire 1D array across all MPI ranks. In the "full read" case, each MPI rank reads its entire partition, and in the "partial read" case, each rank reads 10% of its partition.

In Figure 6, we compare the observed I/O time for reading a partial amount of data and reading the entire dataset by a various number of MPI ranks. Note that this is a weak scaling test, where for each set of bars in the plot (from 16 to 512 ranks on x-axis), the size of the 1D array increases. The number of particles in each rank's partition is equal to 16 x 2$\hat{1}$024 (16M). As a result, the observed time is equal for both the partial and full read cases, showing efficient scaling. Although each MPI rank is reading only 10% of the data, the "partial read" time is nearly 45% of the 'full read" time. The observed time is a combination of reading the metadata of all the datasets and then reading the actual data, the metadata reading overhead is common in both cases. In the "partial read" case, the time is only reduced in reading the data, but not the metadata. Further analysis of this time is needed to profile the metadata and the data read times separately to identify any other inefficiencies. We also compare the observed read rate for these two cases, which understandably shows higher rates for reading the full data as the amount of data transferred is 90% more than that in the "partial read" case.

C. Impact of locality

In Figure 8, we compare the write rates on Cori for various numbers of MPI ranks when the memory layout and file layout are contiguous or compound data types. This plot shows that when the memory layout and file layout are matching, i.e., individual memory 1D arrays written as separate HDF5 datasets in the file (CC) and compound datatype in memory written as an HDF5 compound datatype (II), the observed write rates are superior. Since there is no difference in mapping and no overhead of converting memory buffer to file layout, these result in contiguous transfers of data and hence a good write rate of 400 GB/s at the scale of 1024 MPI ranks and more than 550 GB/s at the scale of 2048 MPI ranks. In the other two cases, i.e., individual 1D arrays have to be converted to an HDF5 compound datatype (CI) and an array of structures in memory to individual HDF5 datasets (IC), the conversion overhead is impacting performance and resulting in lower than writing contiguous data. An interesting observation is that forming an HDF5 compound datatype layout in a file from individual 1D arrays (CI) causes significant overhead and performs the worst of all these four patterns. Writing individual HDF5 1D datasets by extracting data from an array of structures achieves respectable write rates of more than 200 GB/s at 2048 MPI ranks.

D. Impact of the shape of arrays

In Figures 9 and 10, we show the performance when writing data of different array dimensions on Cori and on Summit, respectively. In Figures 11, we depict the observed read rates for different array dimensional data on Cori. In these tests,

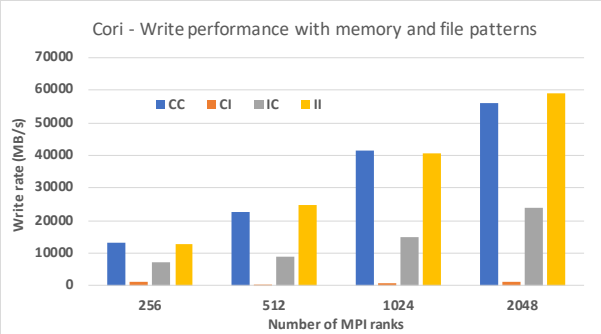

Figure 8: Cori - write - patterns

we use configurations that write or read individual arrays in memory to or from separate HDF5 datasets. In other words, we did not use any array of structures or compound datatypes. Our main observation is that writing or reading the same amount of data in different dimensions achieved relatively similar performance. Earlier studies demonstrated that 3D decomposition achieves poor I/O performance [26]. However, other than the usual variance in performance, we observed similar write and read performance with varying dimensions. We have further studied the code used in [26][3]. We confirmed that the overhead in initializing the fill value in NetCDF4 was the actual cause of poor performance than the actual I/O latency. On Summit, we were observing the write rates dropping at the scale of 4K cores, which needs further investigation.

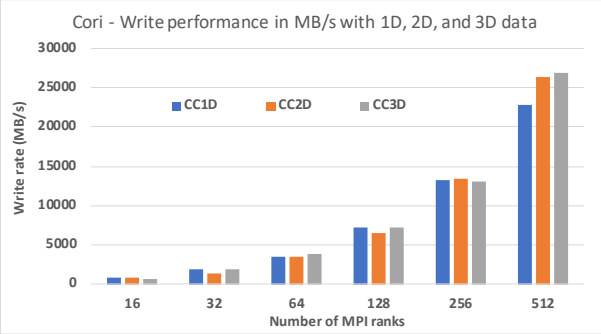

Figure 9: Observed write rate on Cori with varying dimensions of arrays (1D, 2D, and 3D) being written to Lustre with different numbers of MPI ranks.

E. Burst Buffer vs. Lustre on Cori

In Figures 12 and 13, we compare the write and read performance of writing data to the Lustre file system and the DataWarp Burst Buffer on Cori. In these tests, we configured the benchmarks to write or read 1D arrays into individual HDF5 datasets in a file. As expected, we observed higher I/O rates with using the Burst Buffer compared to those with Lustre. However, when we consider performance beyond 64

[3]`https://github.com/gflofst/e2e-hpdc2011`

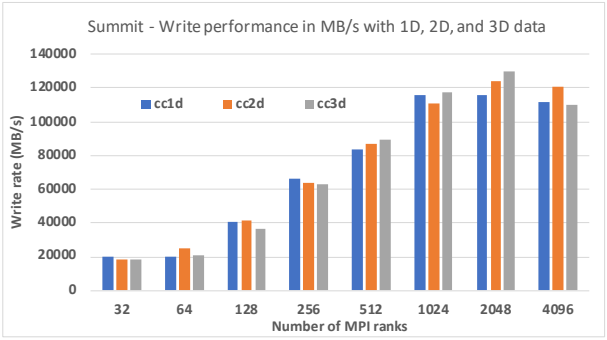

Figure 10: Observed write rate on Summit with varying dimensions of arrays (1D, 2D, and 3D) being written to GPFS with different numbers of MPI ranks.

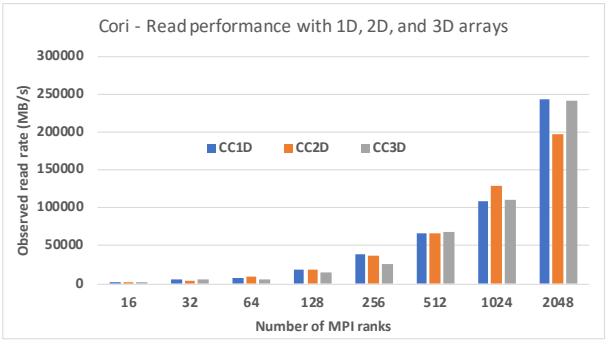

Figure 11: Observed read rate on Cori with varying dimensions of arrays (1D, 2D, and 3D) being read from Lustre with different numbers of MPI ranks.

MPI ranks, the Burst Buffer performance 2.45× faster than Lustre for writes on average. The read performance speedup of the Burst Buffer over Lustre is 1.7×.

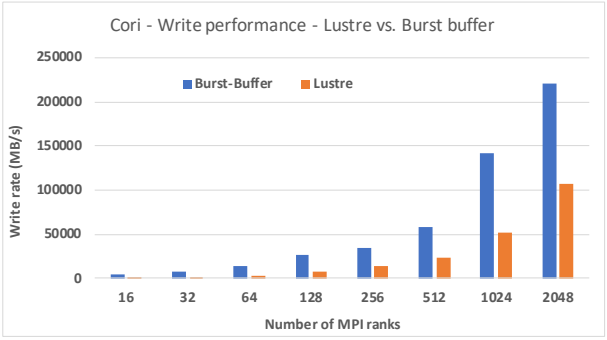

Figure 12: Observed write rates of the DataWarp Burst Buffer and Lustre on Cori.

VI. Conclusion and Future Work

In this paper, we introduce `h5bench`, a suite of benchmarks for performing writing and reading data in various I/O patterns using the HDF5 API. These patterns represent I/O operations considering data structures in memory and file layout, and multi-dimensional arrays (1D, 2D, and 3D). We have also

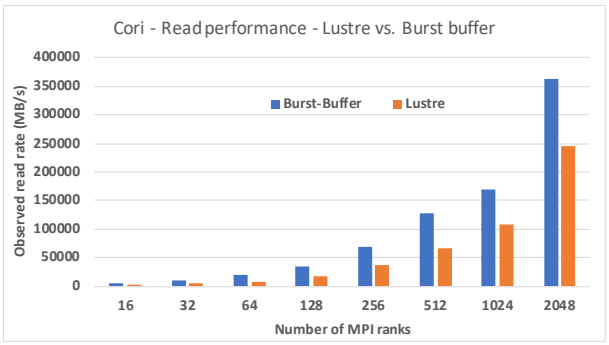

Figure 13: Observed read rates of the DataWarp Burst Buffer and Lustre on Cori.

added new HDF5 functionality such as asynchronous I/O that overlaps I/O phases with computation phases and can hide a significant portion of I/O latency when there are multiple I/O phases. Users can use the `h5bench` benchmark by setting various configuration parameters. Our performance evaluation of the benchmark on Cori (a Cray XC40 platform) and on Summit at OLCF shows superior performance improvements using asynchronous I/O. We also demonstrate the impact of in memory and file layout data structures – i.e., I/O that doesn't require any transformation of data structure between memory and file layouts achieve good performance. The benchmark is available at `https://github.com/hpc-io/h5bench` with a BSD-like license that allows external contributions as well as using the code publicly. In the future, we plan to integrate various benchmarks to the suite by adding new features in HDF5, such as caching using memory and node-local storage, compression, etc. We also packaged metadata stress testing benchmarks in the `h5bench` suite, that were developed by external developers. We will evaluate these benchmarks as well after further optimizations.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used 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. This research 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.

References

[1] The HDF Group. (1997-) Hierarchical Data Format, version 5. Http://www.hdfgroup.org/HDF5.

[2] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *EDBT/ICDT*, 2011, pp. 36–47.

[3] R. Rew and G. Davis, "netcdf: An interface for scientific data access," in *IEEE Computer Graphics and Applications*, vol. 10, no. 4. IEEE, Jul. 990, pp. 76–82.

[4] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. G. Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. M. Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel, "Root — a c++ framework for petabyte data storage, statistical analysis and visualization," *Computer Physics Communications*, vol. 180, no. 12, pp. 2499–2512, 2009, 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465509002550

[5] Rob Latham, Rob Ross, Rajeev Thakur, Kui Gao, Alok Choudhary, Wei-keng Liao, Jianwei Li and Bill Gropp , "Parallel netcdf," 2003-2010.

[6] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *ICPADS*, 1999, pp. 23–32.

[7] J. M. Kunkel, J. Bent, J. Kunkel, and G. S. Kunkel, "Establishing the IO-500 Benchmark," The IO500 Foundation, Tech. Rep., 2016. [Online]. Available: https://www.vi4io.org/_media/io500/about/io500-establishing.pdf

[8] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling Transparent Asynchronous I/O using Background Threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 11–19.

[9] H. Tang, Q. Koziol, S. Byna, and J. Ravi, "Transparent Asynchronous Parallel I/O using Background Threads," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021.

[10] S. Byna, M. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems," *Journal of Computer Science and Technology*, vol. 35, pp. 145–160, 2020.

[11] M. C. Miller, "Multi-Purpose, Application-Centric, Scalable I/O Proxy Application, Version 00," 6 2015. [Online]. Available: https://www.osti.gov/biblio/1232293

[12] F. C. for Computational Science, "The FLASH code," http://flash.uchicago.edu/site/flashcode/.

[13] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. Mccorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. V. Straalen, "Chombo software package for amr applications design document. technical report lbnl-6616e," Lawrence Berkeley National Laboratory, Tech. Rep., 2003.

[14] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. P. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "Amrex: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, 2019. [Online]. Available: https://doi.org/10.21105/joss.01370

[15] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and Design of Cray DataWarp," in *Proceedings of Cray User Group Meeting, CUG 2016*, 2016.

[16] "HDF5 Virtual Object Layer (VOL) User Guide," Feb. 2020.

[17] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A Lightweight Low-Level Threading and Tasking Framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[18] K. Wu, S. Byna, B. Dong, and USDOE, "VPIC IO Utilities," 12 2018. [Online]. Available: https://www.osti.gov/biblio/1487266

[19] S. Byna, "BD-CATS-IO, Version 00," 4 2017. [Online]. Available: https://www.osti.gov/biblio/1459439

[20] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Supercomputing*, 2012, pp. 59:1–59:12.

[21] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu, ""parallel i/o, analysis, and visualization of a trillion particle simulation"," in *Supercomputing*, 2012, pp. 59:1–59:12.

[22] 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, p. 39, Mar. 2013.

[23] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer, "Workload characterization of a leadership class storage cluster," in *2010 5th Petascale Data Storage Workshop (PDSW) 2010*. IEEE, nov 2010. [Online]. Available: https://doi.org/10.1109%2Fpdsw.2010.5668066

[24] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Euro-Par*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 302–311.

[25] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," ser. ICPP '12. USA: IEEE Computer Society, 2012, p. 148–157. [Online]. Available: https://doi.org/10.1109/ICPP.2012.45

[26] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 49–60. [Online]. Available: https://doi.org/10.1145/1996130.1996139

[27] M. M. A. Patwary *et al.*, "BD-CATS: Big Data Clustering at Trillion Particle Scale," in *Supercomputing*, 2015.