Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

RESEARCH PAPER – SPECIAL ISSUE

h5bench: A Unified Benchmark Suite for Evaluating HDF5 I/O Performance on Pre-Exascale Platforms

Jean Luca Bez*[1] | Houjun Tang[1] | Scot Breitenfeld[2] | Huihuo Zheng[3] | Wei-Keng Liao[4] | Kaiyuan Hou[4] | Zanhua Huang[5] | Suren Byna[1,6]

[1]Scientific Data Division, Lawrence Berkeley National Laboratory, California, United States
[2]HDF Group, Illinois, United States
[3]Data Science Group, Argonne National Laboratory, Illinois, United States
[4]Electrical and Computer Engineering Department, Northwestern University, Illinois, United States
[5]Computer Science Department, Northwestern University, Illinois, United States
[6]Department of Computer Science and Engineering, The Ohio State University, Ohio, United States

Correspondence
*Jean Luca Bez Email: jlbez@lbl.gov

Summary

Parallel I/O is a critical technique for moving data between compute and storage subsystems of supercomputers. With massive amounts of data 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 representative of current workloads on HPC systems. Toward 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 due to the parallel I/O library's heavy usage in various scientific applications running on supercomputing systems. The various tests benchmarked in the **h5bench** suite 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), I/O modes (synchronous and asynchronous). In this paper, we present the observed performance of **h5bench** executed along several of these dimensions on existing supercomputers (Cori and Summit) and pre-exascale platforms (Perlmutter, Theta, and Polaris). **h5bench** measurements can be used to identify performance bottlenecks and their root causes and evaluate I/O optimizations. As the I/O patterns of **h5bench** are diverse and capture the I/O behaviors of various HPC applications, this study will be helpful to the broader supercomputing and I/O community.

KEYWORDS:
HDF5, I/O benchmarks, I/O access patterns, I/O performance

1 | INTRODUCTION

Applications using high-performance computing (HPC) resources depend highly on storing data and retrieving previously stored data from file systems. Therefore, I/O libraries such as HDF5[1,2], netCDF[3], and ROOT[4] 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 to access data concurrently from file systems need to be efficient.

Parallel I/O benchmarks representing I/O operations in real-world HPC applications play an important role in evaluating I/O libraries and fixing any performance bottlenecks. Several parallel I/O benchmarks are available that measure the 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 the 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). It also provides various distributed computing interfaces, such as S3 and HDFS. MDtest, now integrated into IOR, is often used for evaluating the

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

metadata performance of POSIX-compliant parallel file systems. Specifically, 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 optimization options and new features in I/O libraries and to identify performance inefficiencies. For instance, HDF5 has recently developed an 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 focuses on file system performance when 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.

This paper focuses on the HDF5 API and parallel I/O library due to its heavy usage in 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. Furthermore, MACSio allows using different I/O library interface plugins in addition to HDF5, such as PDB, TyphonIO, Exodus, and parallel I/O patterns. While MACSio covers many patterns, new features such as asynchronous I/O need further benchmarking. We have previously developed Parallel I/O Kernel (PIOK) suite[12] that included simple HDF5 I/O operations such as reads and writes with basic array data types. However, there are several I/O patterns that PIOK does not cover, such as the multi-dimensionality of the array data and asynchronous I/O. There are a few HDF5 I/O benchmarks that are based on application I/O, such as FLASH-IO[13], ChomboIO[14], and AMReX[15]. While these benchmarks test specific I/O patterns, there is no single framework for testing them under different tuning options, and new HDF5 features, such as asynchronous I/O[8], caching[16], or HDF5 subfiling.

In our effort, we focus on bringing together a set of HDF5 benchmarks called `h5bench` and making them available to 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, and 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)). We also investigate the performance of distinct HDF5 features on three new pre-exascale platforms: Perlmutter (a Cray EX supercomputer at NERSC), Theta (a Cray XC40 system at Argonne Leadership Computing Facility (ALCF)), and Polaris (an HPE Apollo 6500 Gen 10+ based system also at ALCF). We evaluate the performance of `h5bench` by exercising different HDF5 features in each system. We focus on the observed I/O rate, which relies on the total wall clock time, thus implicitly considering the impact of both data and metadata operations on the perceived performance. In addition, 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[17]. 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 2, we briefly introduce HDF5 and the VOL connectors. In Section 3, we describe `h5bench`, with its different modes of the basic I/O operations, access patterns, and kernels. In Section 4 we describe the platforms, experimental setup, and demonstrate how `h5bench` can be used to exercise various HDF5 features and I/O access patterns. In Section 5, we conclude the paper with a brief discussion of future work.

2 | HDF5 AND VOL CONNECTORS

2.1 | HDF5 I/O Library

HDF5 (Hierarchical Data Format Version 5) is a self-describing file format and I/O library[1] that provides flexibility, extendibility, and portability for scientific applications. It supports a variety of data structures and stores data and its corresponding metadata (e.g., data type, data size) within a single file. HDF5 relieves the manual file management from the users, 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. It 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 the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF)[10]. Because of this widespread use, ensuring efficient HDF5's parallel I/O performance is a critical task for HPC facilities. Providing benchmarks and tuning I/O patterns in them paves the path for achieving the goal of efficiency and overall performance for applications.

2.2 | Virtual Object Layer (VOL) Connectors

HDF5 provides the Virtual Object Layer (VOL)[18] that allows intercepting the HDF5 I/O routines and applying optimizations for better data management and is transparent to the application. This layer stands between the public HDF5 API calls and the storage-oriented code in HDF5, as illustrated by Figure 1. It allows the creation of a VOL connector to perform arbitrary operations when storage-oriented calls are issued. Non-storage HDF5 API calls (e.g., dataspace and property list calls) do not go through the VOL layer. VOL connectors can be divided into *passthrough* and *final* connectors.

Passthrough connectors perform operations and then invoke another VOL connector layer underneath it. That means they can also be stacked on top of other VOL connectors. For instance, the Cache VOL[2] can be stacked on top of the Asynchronous VOL connector (referred to as Async VOL)[3] before reaching the native final VOL. On the other hand, terminal VOL connectors do not pass operations to other VOLs and typically map HDF5 file objects and metadata to storage. For instance, the DAOS VOL connector[4] is a terminal VOL that allows for direct interfacing with the Distributed Asynchronous Object Storage (DAOS) system, bypassing both MPI I/O and POSIX I/O for efficient and scalable I/O, removing the limitations of the native HDF5 file format and enabling new features such as independent creation of objects in parallel, key-value store objects, data recovery, asynchronous I/O.

In the following subsections, we briefly introduce the VOL connectors we used in the context of our experiments with `h5bench` in different platforms. We highlight their goals and advantages.

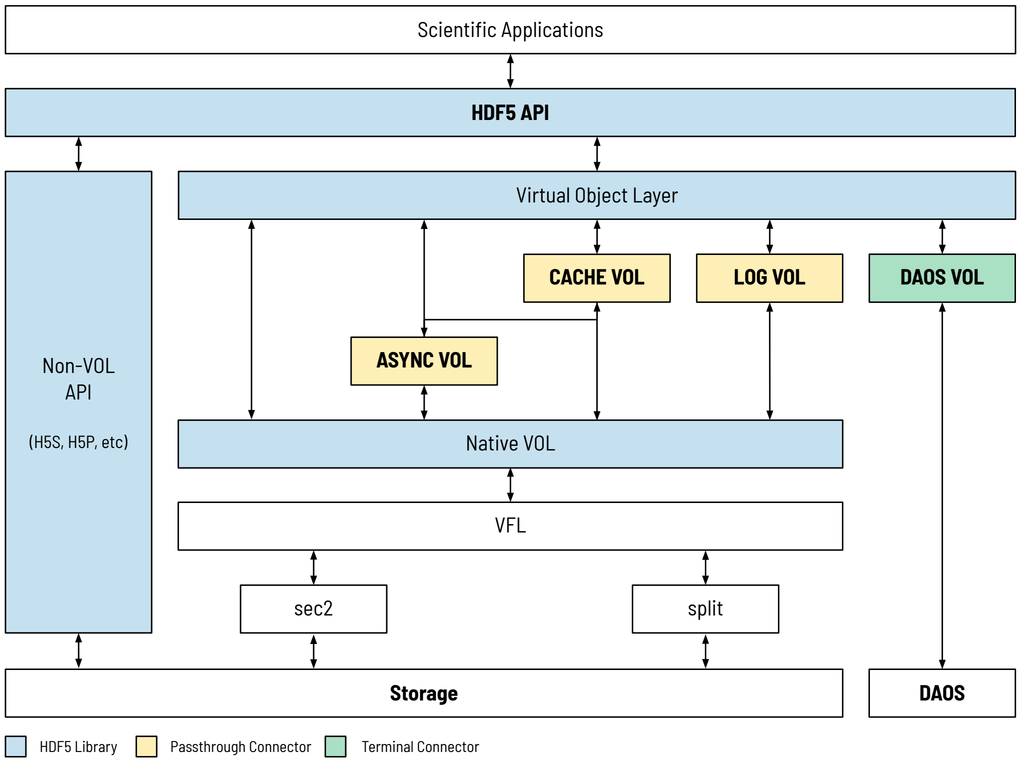


Figure 1 The HDF5 Virtual Object Layer (VOL) connector architecture. Passthrough connectors perform operations and then invoke another VOL connector layer underneath it. Terminal connectors map HDF5 file objects and metadata to storage.

2.2.1 | Asynchronous I/O VOL Connector

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. The asynchronous I/O VOL connector (Async VOL)[8] is developed to enable asynchronous I/O for HDF5 operations using background threads.

[2]https://github.com/hpc-io/vol-cache
[3]https://github.com/hpc-io/vol-async
[4]https://github.com/HDFGroup/vol-daos

This implementation can be compiled as a dynamically linked library (DLL) and dynamically loaded by the HDF5 library through setting environment variables by the user. The background threads are managed by Argobots, a lightweight low-level threading framework[19].

There are two modes of asynchronous I/O in HDF5: implicit and explicit. The implicit mode needs minimal code changes but has performance limitations (e.g., all read operations are synchronous). 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].

2.2.2 | Cache VOL Connector

The Cache VOL connector[16] allows caching or staging data on node-local storage and then moving data asynchronously between the node-local storage and parallel file system. This will enable applications to hide I/O overhead behind the computation. The overall observed I/O overhead is from moving data between the compute nodes and node-local storage, assuming that there is enough computation to hide the overhead for data migration. This improves parallel I/O efficiency and scalability. It benefits applications with heavy check-pointing writes and applications with repeated intensive reads.

Cache VOL uses the Async VOL connector for asynchronous data migration. It is always recommended to stack both VOLs together. Like the asynchronous VOL, cache VOL can be compiled as a dynamically linked library (DLL) and dynamically loaded by the HDF5 library through setting environment variables by the user. Existing HDF5 applications can use Cache VOL with minimal code modifications.

2.2.3 | Log VOL Connector

The Log VOL connector stores write requests contiguously in files, using a data layout similar to a time log. The write requests from a single process are appended one after another into a contiguous file space. Such a strategy avoids the potentially expensive inter-process communication required if data is stored in a canonical order in the file. Such a layout has been proven[20] to be less susceptible to complex I/O patterns and able to provide a higher parallel write performance. However, this strategy requires additional metadata to keep track of the locations of individual requests relative to their canonical layouts so that, later on, the read operations can work as expected. In a way, the overhead of reorganizing the data is deferred to once the data needs to be read.

For parallel I/O, all write requests from an MPI process are stored in a single contiguous file space. The contiguous file spaces of all processes are organized in the file by following the increasing order of MPI ranks. The files created by the Log VOL connector are valid HDF5 files, conforming to the HDF5 file format specification. Because the Log VOL connector is implemented through the HDF5 VOL plugin, existing HDF5 programs can benefit from this data layout to achieve a better parallel write performance without changing source code.

2.3 | HDF5 Subfiling

The *subfiling* feature, introduced in HDF5 version 1.14.0, is a compromise between a single shared file and one file per process. It helps to avoid the two extremes associated with (1) lock contention problems on parallel file systems for a single shared file and (2) generating a massive and unmanageable number of files for one file per process. As a virtual file driver (VFD) in HDF5, subfiling strips the logical HDF5 file and redirects I/O requests to the corresponding subfile(s) per node. The subfiling VFD uses an I/O concentrator (IOC) VFD, which utilizes nodal IOC threads (default is four threads per node) to aggregate data from nodal ranks to perform I/O from/to the subfile(s). Since the subfiles are an HDF5 file striped across the nodes, the subfiles can be combined (using the HDF5 provided script *h5fuse.sh*) as a post-processing step to create a single HDF5 file that the native HDF5 VFD can manage. The subfiling and IOC VFD parameters are controlled by environment variables or a configuration structure associated with the file access property list.

For `h5bench`, the subfiling configuration is controlled by environment variables, and the subfiling VFD can be enabled or disabled by using the `SUBFILING` keyword in the configuration file. Additionally, the subfiling VFD can use node-local storage by setting the environment variable `H5FD_SUBFILING_SUBFILE_PREFIX` to the node-local storage path. The performance using `h5bench` with subfiling on the Summit supercomputer is discussed in Section 4.3.

3 | THE `h5bench` BENCHMARKING SUITE

Intending to provide 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`[5]. The following section describes the benchmark design, supported patterns, and available features.

3.1 | `h5bench` Architectural Overview

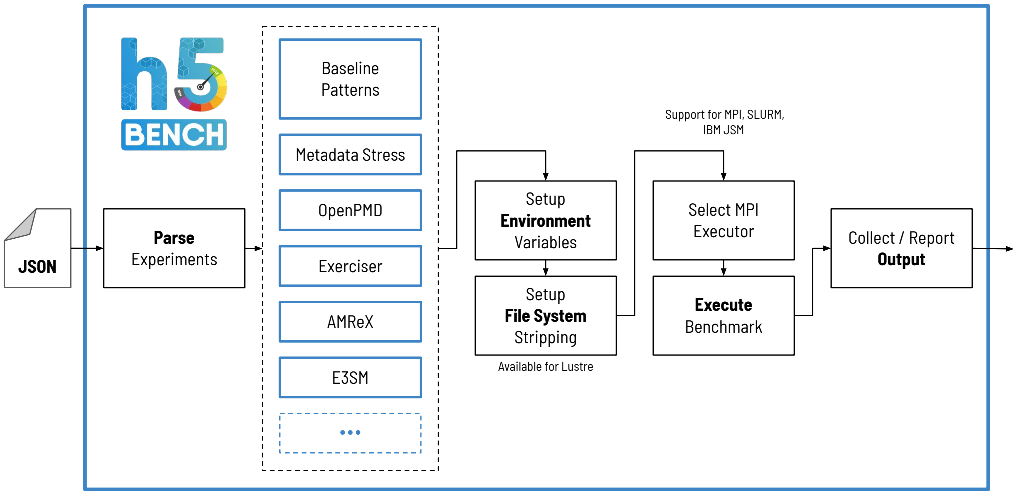


Figure 2 An overview of the `h5bench` suite architecture.

`h5bench` provides a single entry point to seamlessly configure and run a single kernel or multiple patterns in a workflow fashion, adding flexibility to combine multiple benchmarks and runs with distinct parameters in a single JSON configuration file. `h5bench` will handle environment variables, launching the experiments with the provided configuration (including the parallel job launcher and the parallel file system striping) and HDF5 VOL connectors (§2.2). Figure 2 illustrates the benchmarking suite architecture.

`h5bench` takes as input a JSON configuration file that has five main properties: `mpi`, `vol`, `file-system`, `directory`, and `benchmarks`. You can set the MPI launcher (`mpi`), and provide the number of processes you want to use. `h5bench` currently has out-of-the-box support for direct MPI via `mpirun` or `mpiexec`, SLURM (`srun`), Component Based Lightweight Toolkit (`qsub`), and IBM Job Step Manager (`jsrun`). For other methods or a fine-grain control on the job configuration, `h5bench` exposes additional configuration options to launch the experiments using the command property you provided.

`h5bench` allows the usage of HDF5 VOL connectors (e.g., Async VOL, Cache VOL, Log VOL, DAOS VOL) for supported benchmarks and kernels, as detailed in Table 1. The `vol` property allows the user to provide the necessary information (additional libraries, connector path, VOL configuration) in the configuration file to handle the VOL setup during runtime. For systems with Lustre deployment `h5bench` can setup the striping through the **file-system** (`stripe-size` and `stripe-count`) property that will be applied to the `directory` created for the given experimental setup. Finally, the `benchmarks` property details the patterns, their execution order, and input parameters, as illustrated by Figure 3.

To facilitate its usage on diverse systems and software stacks, `h5bench` is also available as a Spack (i.e., `spack install h5bench`) recipe with multiple variants to allow a fine grain control of which parts of the benchmark suite are compiled. `h5bench` is also part of the Extreme-scale Scientific Software Stack (E4S)[21] community effort to provide open-source software packages for developing, deploying, and running scientific applications on high-performance computing platforms.

Not all I/O kernels in `h5bench` support all VOL connectors, as detailed in Table 1, as some require code changes or do not provide support for all the HDF5 API calls used by a kernel. `h5bench` is an ongoing community effort seeking to congregate access patterns from diverse science domains. Furthermore, because each benchmark and kernel measures and reports performance metrics in diverse formats, `h5bench` currently does not present a final detailed aggregated report. Future work will address this issue by providing a unified API report to extract metrics of interest.

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

Figure 3 Sample snippet of a workflow configuration file in JSON for the `h5bench` Benchmarking Suite.

```
{
   "benchmark": "write",
   "file": "test.h5",
   "configuration": {
      "MEM_PATTERN": "CONTIG",
      "FILE_PATTERN": "CONTIG",
      "NUM_PARTICLES": "16 M",
      "TIMESTEPS": "5",
      "DELAYED_CLOSE_TIMESTEPS": "2",
      "COLLECTIVE_DATA": "NO",
      "COLLECTIVE_METADATA": "NO",
      "EMULATED_COMPUTE_TIME_PER_TIMESTEP": "1 s",
      "NUM_DIMS": "1",
      "DIM_1": "16777216",
      "DIM_2": "1",
      "DIM_3": "1",
      "MODE": "SYNC",
      "CSV_FILE": "output.csv"
   }
}
```

Table 1 I/O benchmarks and I/O kernels available in `h5bench`. Tested support for existing HDF5 VOL connectors is detailed for each kernel.

I/O Pattern / Kernel	SYNC	ASYNC VOL	CACHE VOL	LOG VOL
h5bench write	✓	✓	✓	✓
h5bench read	✓	✓	✓	✓
AMReX	✓	✓	✗	✗
OpenPMD (write)	✓	✓	✗	✗
OpenPMD (read)	✓	✓	✗	✗
E3SM-IO	✓	✓	✗	✓

3.1.1 | Baseline I/O Access Patterns

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[22]) and for reading (called BD-CATS-IO[23]). 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[24]. BD-CATS-IO was derived from a parallel DBSCAN algorithm's reading particle data, such as that generated by VPIC[25] or Nyx[26] simulations. Both these write and read patterns are simple in the data structures in memory and in file. In VPIC, we recently implemented new particle and file data write strategies that write 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 is also commonly referred to as an "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 for users to exercise various I/O patterns. We briefly cover some of these configurations in this paper. For detailed information, please refer to `h5bench` documentation[6].

The baseline `h5bench` patterns in the suite assume 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[27,28,29]. 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[24] computes $\approx 20,000$

[6] `h5bench` Documentation – `https://h5bench.readthedocs.io`

time steps and dumps the data $\approx 2,000$ time steps, i.e., a total of 10 time steps. In the benchmark, we do not use real computations but instead, rely on `sleep()` to emulate the computation time. The produced data in the write benchmarks are 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 read or write operations. These times are obtained by measuring the elapsed times of `H5Dwrite` and `H5Dread` functions. In the case of asynchronous I/O operations, these are also measured, but 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 data size and the times (raw and observed, respectively).

A contiguous pattern in the memory means that multiple arrays are of the same basic data types, such as integer, float, double, etc. The non-contiguous pattern in memory also referred to as "array of structures" (AoS) or "derived data type", represents compound datatypes derived from basic data types. The first helps access adjacent work items in contiguous memory locations, while the latter is often more intuitive from the developer's perspective as each structure is kept together. Once that data needs to be persisted in a file, it can use the same strategy or the opposite one used to represent the data in memory. Table 2 depicts this by comparing in-memory and in-file representations when using HDF5, for instance, to store data in a contiguous or compound fashion.

Table 2 In-memory data structure and in-file data layout mappings. For illustration purposes, we restrict ourselves to 1D arrays.

In-memory representation	**In-file representation**
Array A: A A A A ⋯ A A Array B: B B B B ⋯ B B *Contiguous*	Dataset A: A A A A ⋯ A A Dataset B: B B B B ⋯ B B *Contiguous*
Array A: A A A A ⋯ A A Array B: B B B B ⋯ B B *Continuous*	Dataset A, B: A B A B ⋯ A B *Compound*
Array A, B: A B A B ⋯ A B *Compound*	Dataset A: A A A A ⋯ A A Dataset B: B B B B ⋯ B B *Continuous*
Array A, B: A B A B ⋯ A B *Compound*	Dataset A, B: A B A B ⋯ A B *Compound*

3.1.2 | OpenPMD Access Pattern

OpenPMD[30] is an open meta-data schema that provides meaning and self-description for data sets in science and engineering. The openPMD-API library[31] provides a reference API to handle data in OpenPMD. In the `h5bench` Suite, we provide support for the write and read parallel benchmarks using the HDF5 backend.

For write operations, the benchmark takes as input the number of dimensions, whether or not it should emulate a balanced load, the ratio of particle to mesh, the number of iterations, the number of mini-blocks (block meshes are represented as a grid of mini blocks), and the size of the grid. For reads, the benchmark takes as input the number of dimensions, the number of mini blocks, the size of the grid, and the read access pattern. The read pattern can be the metadata only, a slice of the ρ mesh in the x, y, or z-axis, or a slice of the 3D magnetic field in the x, y, or z-axis.

3.1.3 | AMReX Access Pattern

AMReX[32] is a framework for massively parallel, block-structured adaptive mesh refinement (AMR)[33] applications used in combustion, accelerator physics, carbon capture and storage, cosmology, and astrophysics. Block-structured AMR enables varying resolutions of spatiotemporal regions of interest during computation.

AMReX takes as input the domain size, the maximum size of each subdomain (used for parallel decomposition), the number of levels, components in the *multifab* class, particles per cell, plot and particle files to write, the sleep time before each write, and whether to check the correctness of checkpoint/restart. Furthermore, it allows AMReX to read grids from a file and specify the refinement ratios and the HDF5 compression algorithm. Both the plot file and particle output organize the data by AMR levels and store them in separate HDF5 groups. Within each level's group, the data from individual AMR blocks are flattened and concatenated into a single 1D array for each datatype (e.g., integers and double-precision floating point values). The access pattern can be categorized as "contiguous-contiguous." Additional metadata describing the simulation parameters and the storage layout is stored in each group. HDF5 chunking is used when compression is used with a default chunk size of $16MB$.

3.1.4 | E3SM Access Pattern

Energy Exascale Earth System Model (E3SM) is a fully coupled model of the Earth's climate, including important biogeochemical and cryospheric processes[34,35]. E3SM uses the Hilbert space curve algorithm to partition the linearized 2D grid that covers the surface of the problem domain under study. The data assigned to each MPI process consists of a long list of noncontiguous subarrays. In E3SM production runs for standard resolution, this number per process can reach several hundreds of thousands, which makes I/O tasks very challenging to obtain good performance on parallel computers. E3SM-IO [7] is a benchmark program that captures the I/O kernel of E3SM. It has been used to evaluate the I/O costs when using various I/O libraries and help improve the E3SM's I/O implementation.

`h5bench` suite currently supports the I/O pattern of three E3SM modules – the atmospheric component (F case), the oceanic component (G case), and the land component (I case). The F case contains three unique data decomposition patterns shared by 387 2D and 3D variables (1 sharing Decomposition 1, 323 sharing Decomposition 2, and 63 sharing Decomposition 3). In the F case, each process writes to many (up to 174953 per process) small and non-contiguous regions in the file. The G case contains 6 data decompositions shared by 52 variables (6 sharing Decomposition 1, 2 sharing Decomposition 2, 25 sharing Decomposition 3, 2 sharing Decomposition 4, 2 sharing Decomposition 5, and 4 sharing Decomposition 6). As a result, processes write to fewer (up to 20888 per process) but larger regions in the G case compared to the F case. The I case contains 5 data decompositions shared by 538 variables (465 sharing Decomposition 1, 69 sharing Decomposition 2, 2 sharing Decomposition 3, 1 sharing Decomposition 4, 1 sharing Decomposition 5, and 4 sharing Decomposition 6). The I/O pattern of the I case is similar to the F case, except that there are holes (unwritten region) in the file, and the data is written in multiple (240) time steps. With a large number of fragment requests and a virtually random I/O pattern, the E3SM-IO is among the most challenging applications for I/O libraries.

4 | RESULTS

The results presented in this section were repeated multiple times, spanning different days to account for the variability of concurrently running jobs sharing the file system. In this paper, we focus on performance, particularly the *observed I/O rate*, which represents the ratio of the amount of data written or read and the total wall clock time taken to perform all the write or read operations. Since we consider the total wall clock time, this metric will also implicitly consider the time spent in metadata-related operations. Section 4.1 summarizes our initial observations on production machines while the following sections explore the initial performance in Perlmutter (§4.5), Theta (§4.6), and Polaris (§4.7) pre-exascale machines.

4.1 | HDF5 Performance on Supercomputers

We evaluated `h5bench` on Cori at the National Energy Research Scientific Computing Center (NERSC) and Summit at the Oak Ridge Leadership Computing Facility (OLCF), in 2021. We present these results as a baseline and investigate the performance of distinct HDF5 features on three new pre-exascale platforms: Perlmutter (Section 4.5), Theta (Section 4.6), and Polaris (Section 4.7).

[7] https://github.com/Parallel-NetCDF/E3SM-IO

4.1.1 | Cori

Cori is a Cray XC40 system with a peak performance of 30 PFlops. It contains two main computation partitions: 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 two hyper-threads, and each node has 128 GB DDR4 memory shared by the two sockets.

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 setting 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. NERSC provides an SSD-based burst buffer that uses Cray DataWarp[36]. 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.

4.1.2 | Summit

The Summit supercomputer is based on the IBM AC922 system. It comprises 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 for 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.

4.1.3 | Experimental Setup

We test the configurations of `h5bench` shown in Table 3 to demonstrate a sample of the capabilities of the benchmark suite. We ran each of these configurations at least three times and reported the best performance for each run. On Cori, we used 16 MPI ranks per node; 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 file access to allow any file object larger than a given threshold to be aligned on an offset address on the 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 4 KB.

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 relating 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 uses the HDF5 event sets that give more control to users when to perform the I/O operations using background threads. One can try many more combinations of configurations for the benchmark suite.

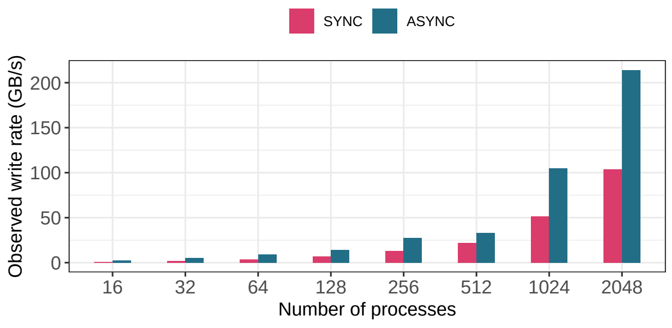
I/O Operations	I/O Modes	Scale*	Platforms
Write	Sync I/O and Async I/O	16 to 2048	Cori / Summit
Read	Sync I/O and Async I/O	16 to 2048	Cori / Summit
Read	Full and Partial	16 to 2048	Cori
Write	Locality	256 to 2048	Cori
Write	Array Dimensions	16 to 2048	Cori / Summit
Read	Array Dimensions	16 to 2048	Cori / Summit
Write / Read	Burst Buffer vs. Lustre	16 to 2048	Cori

* Scale details the range of processes in the experiments using powers of two.

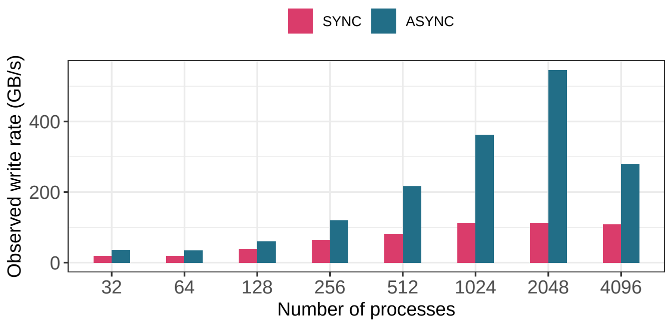
Table 3 Experimental setup used to exercise petascale platforms.

4.1.4 | HDF5 Performance, Scalability, and Asynchronous I/O

In Figure 4(a), we compare the observed I/O rate for writing data in asynchronous and synchronous I/O modes on Cori, and in Figure 4(b), 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 increased, 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. We observed one abnormality on Summit at the scale of 4K cores, where the I/O rating dropped to ≈ 290 GB/s. This may be due to interference in the file system when these jobs ran.



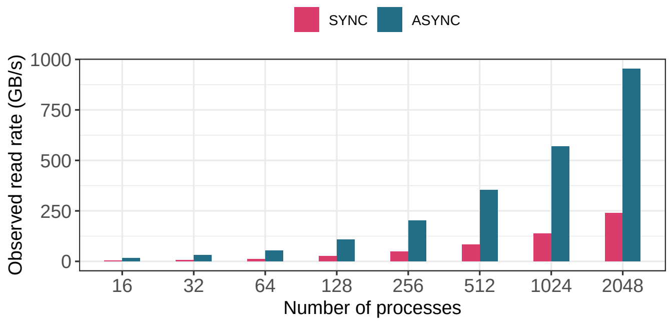
(a) Cori (16 ranks per node)



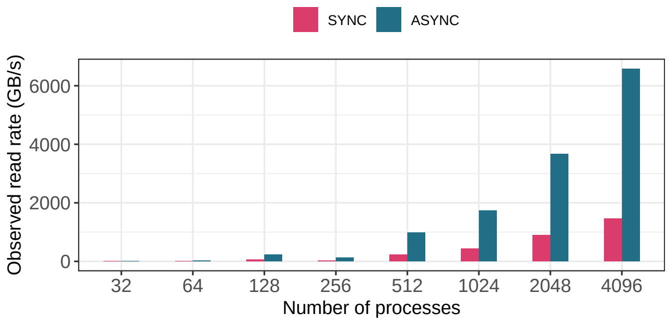
(b) Summit (32 ranks per node)

Figure 4 Write performance of `h5bench` in synchronous and asynchronous modes with varying numbers of MPI ranks.

In Figures 5(a) and 5(b), 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, 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 computation phase overlaps the elapsed time by the benchmark. 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 the file does not have a computation phase to follow. Hence, the file write phase is not overlapped with computation. Similarly, 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 high. Another observation is that in the read case, we do not see a performance drop from 2K to 4K MPI ranks on Summit.



(a) Cori (16 ranks per node)

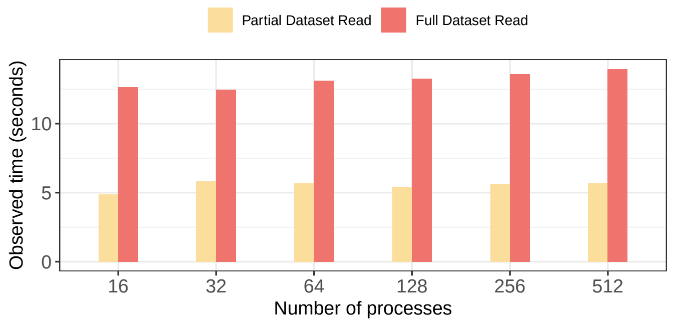


(b) Summit (32 ranks per node)

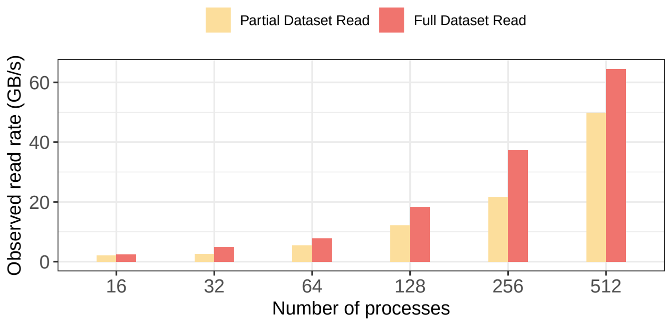
Figure 5 Read performance of `h5bench` in synchronous and asynchronous modes with varying number of MPI ranks.

When data analysis applications read data, it is a commonplace to read a partial amount of data or slices of data instead of the entire data[23,37]. As mentioned earlier, a big data clustering application[38] reads the top $k\%$ of the particle data to analyze, where k is variable. We mimic this pattern by 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(a), we compare the observed I/O time for reading a partial amount of data and reading the entire dataset by various numbers 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 the x-axis), the size of the 1D array increases. The number of particles in each rank’s partition equals 16×2^{1024} (16 M). 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 in Figure 6(b), 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.



(a) Observed time (seconds)



(b) Observed rate (GB/s)

Figure 6 `h5bench` partial and full dataset read performance with a varying number of MPI ranks.

4.1.5 | HDF5 Performance with Log VOL Connector

Figure 7 shows the write performance of E3SM-IO when using the Log VOL connector, ADIOS, and PnetCDF. Similar to the Log VOL connector, the ADIOS method also stores data in a log layout in files. On the contrary, the PnetCDF method stores data in the canonical order, which requires inter-process communication to exchange write requests in the MPI collective write operations. A direct comparison of this PnetCDF method against the other two of using the log layout is not considered fair, as the PnetCDF library can also be used to store data in a log layout. Interested readers are referred to the E3SM-IO’s GitHub repository[8]for additional performance results. However, we add the PnetCDF results in the same chart, so they can be used to help understand the impact of communication cost in the MPI-IO on the overall performance.

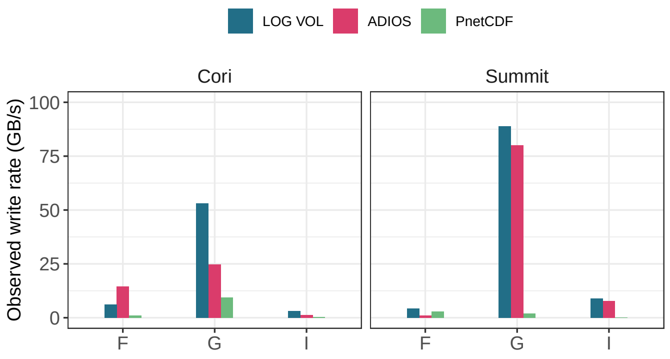


Figure 7 Observed write bandwidths of E3SM I/O using the Log VOL connector, ADIOS, and PnetCDF. Both Log VOL and ADIOS methods write data in the log layout. The PnetCDF method writes data in the canonical layout.

[8]https://github.com/Parallel-NetCDF/E3SM-IO

The E3SM-IO benchmark is configured to match the setup of the three production runs of high-resolution E3SM simulations. The F, G, and I cases run on 21600, 9600, and 1344 processes, respectively. In our experiments, the subfiling option in both Log VOL connector and ADIOS was enabled to produce one output file per compute node allocated. This feature avoids file system lock conflicts and thus can improve write performance significantly. Our experiments were carried out on Cori at NERSC and Summit at OLCF. On Cori, we ran 64 processes per KNL compute node, using The Lustre parallel file system with file striping configuration of 1 MB stripe size and 8 stripe count. On Summit, we ran 84 processes per compute node, using the IBM Spectrum Scale GPFS parallel file system.

Among the three cases, the G case's I/O pattern contains a smaller number of noncontiguous requests, and the request lengths are relatively larger than the other two cases. The number of time steps (data checkpoints) in the three cases is 25 in the F case, 1 in the G case, and 240 in the I case. In both the F and I cases, the file space of all variables is fully written, while in the I case, only partial space of variables is written. Figure 7 clearly shows that using the log layout in the Log VOL connector and ADIOS to store data outperforms the canonical layout used in the PnetCDF method. In addition, we observe the Log VOL connector outperforms the ADIOS method.

4.1.6 | HDF5 Performance with Caching on Node-local Storage in Summit

In Figure 8, we show the performance of HDF5 with Cache VOL on Summit for 1D contiguous (in-memory) contiguous (in-file) pattern with 8 million particles. The experiments were run with 16 processes per node from 1 to 256 nodes. NVMe SSDs were used for caching data in Cache VOL. The data were written to NVMe SSDs synchronously and moved to the GPFS parallel file system asynchronously. The emulated compute time is 100 seconds.

The observed write rate for Cache VOL scales linearly with the number of nodes, eventually outperforming the baseline HDF5 (SYNC) because more and more NVMe SSDs were used and the aggregate write bandwidth of them is higher than the bandwidth of GPFS at the same scale.

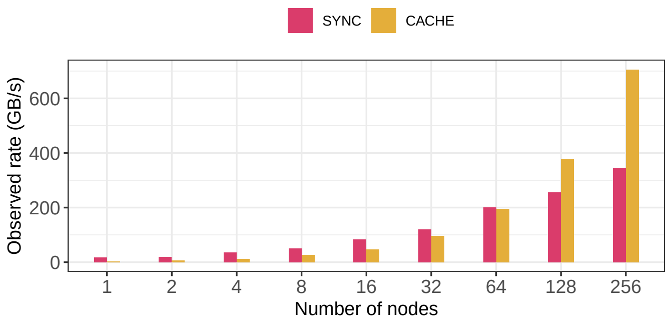


Figure 8 Observed rate for 1D contiguous (in-memory) contiguous (in-file) pattern using SYNC and CACHE VOL in Summit.

4.1.7 | Impact of Locality in Cori

In Figure 9, 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.

4.2 | Impact of Array Dimensions in Cori and Summit

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

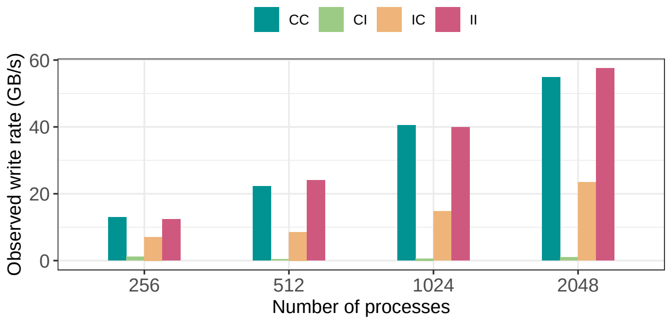
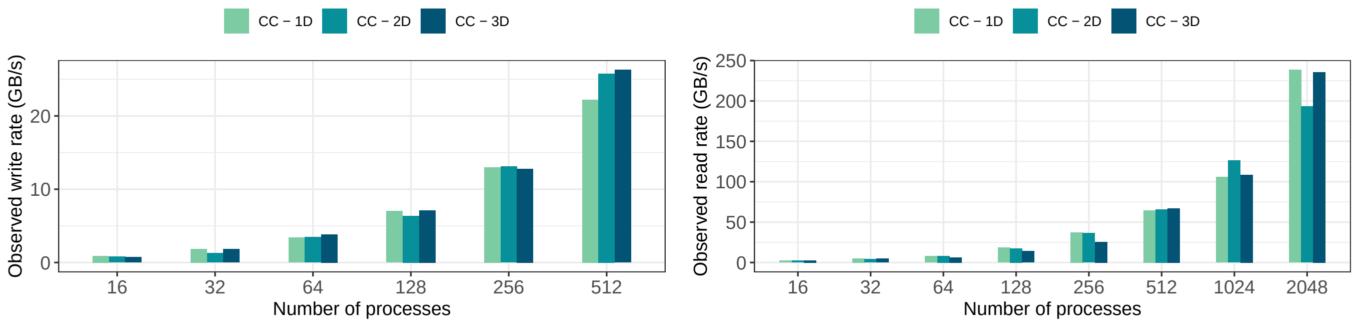


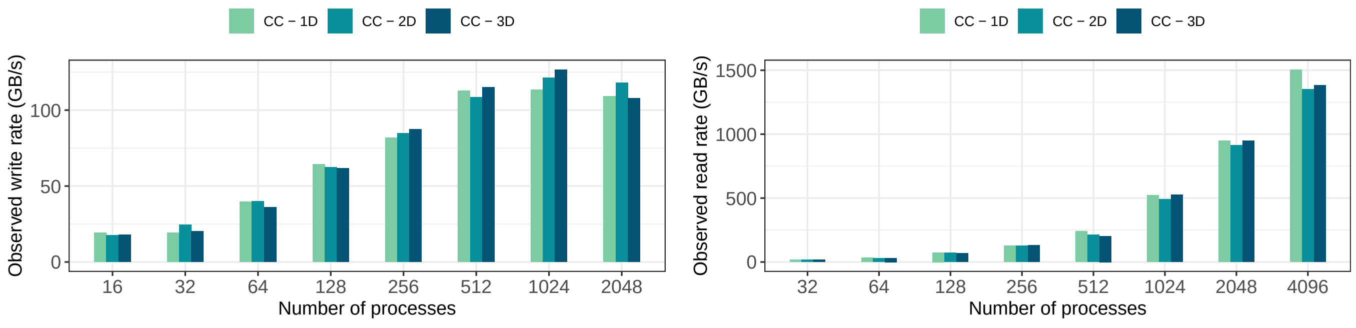
Figure 9 Observed rate of write patterns in Cori at different scales when the memory and file layout are contiguous or compound data types.

demonstrated that 3D decomposition achieves poor I/O performance[37]. 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[37][9]. 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[39]. On Summit, as depicted by Figures 11(a) and 11(b), we observed the write rates dropping at the scale of 4K cores, which needs further investigation.



(a) Write (b) Read

Figure 10 `h5bench` observed write and read rates with varying dimensions of arrays (1D, 2D, and 3D) being written to the shared Lustre file system on Cori with different numbers of MPI ranks.



(a) Write (b) Read

Figure 11 `h5bench` observed write and read rates with varying dimensions of arrays (1D, 2D, and 3D) being written to the shared GPFS file system on Summit with different numbers of MPI ranks.

4.3 | Impact of Subfiling on Summit

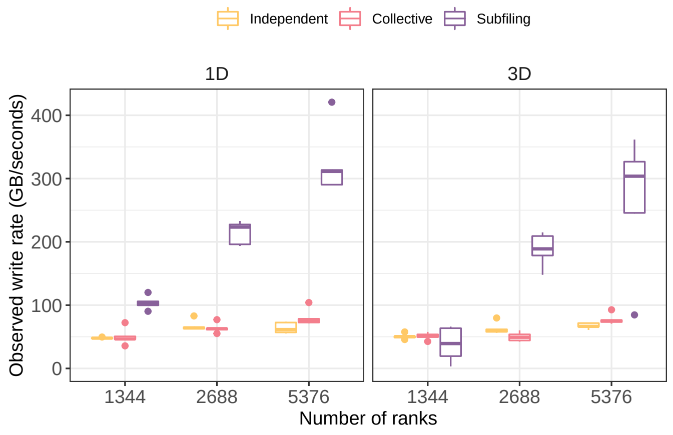
The first subfiling study (§2.3) was performed on Summit using `h5bench` the 1D and 3D dimensions of arrays I/O patterns. The total file sizes between the array rank cases are the same and scale with the number ranks (Table 4). The dataset is contiguous for both in-memory and in-file.

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

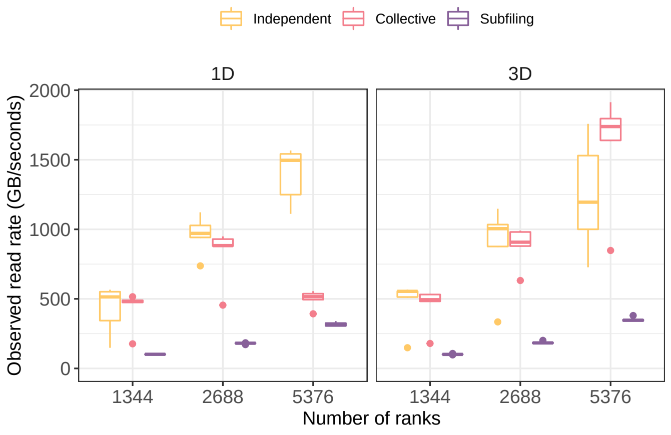
Number of Ranks	File Size (TB)
1344	0.625
2688	1.25
5376	2.50

Table 4 The `h5bench` file sizes for both 1D and 3D dataset I/O patterns.

The default subfiling VFD parameter of one subfile per node was used and compared to both collective and independent I/O to a single shared file (SSF) over five runs, Figure 12. For writing, subfiling improves the rate for both array dimension patterns as the number of ranks increases, whereas the rate for SSF remains relatively constant. The subfiling read does not perform as well as the SSF case and remains an open investigation on ways to improve its performance. The second experiment studied the effects of using more than one subfile per node, Figure 13. There is an apparent performance gain for both array dimension patterns by using more subfiles per node. The 3D dimension pattern shows slightly inconsistent performance compared to the 1D dimension case at higher rank counts.

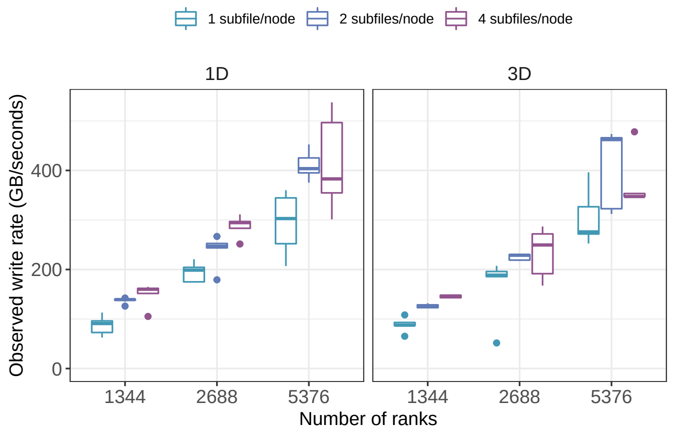


(a) Write

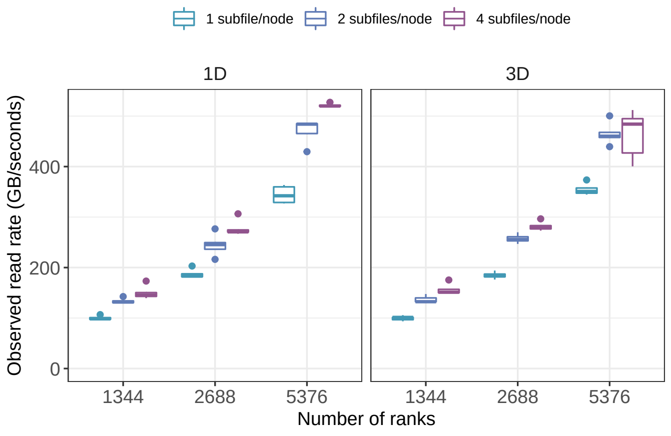


(b) Read

Figure 12 The `h5bench` observed write and read rates for a single shared HDF5, using collective and independent I/O, and one subfile per node to the GPFS file system on Summit with different MPI ranks.



(a) Write

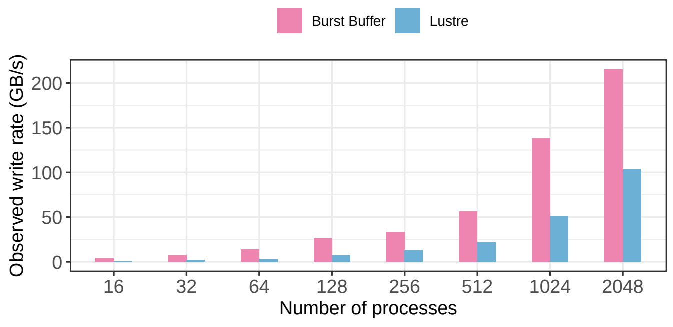


(b) Read

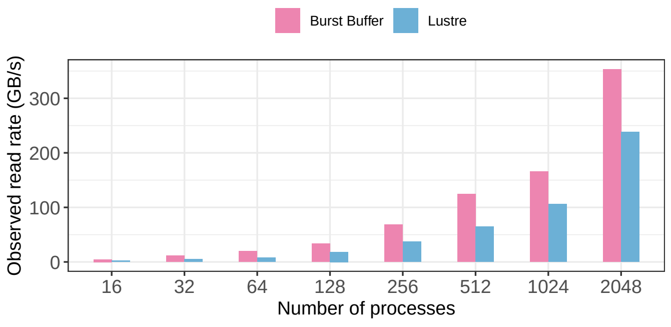
Figure 13 The `h5bench` observed write and read rates for different subfiles per node to GPFS on Summit with varying the number of MPI ranks.

4.4 | Burst Buffer vs. Lustre on Cori

In Figures 14(a) and 14(b), 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 using the Burst Buffer than those with Lustre. However, when considering performance beyond 64 MPI ranks, the Burst Buffer performance is $2.45\times$ faster than Lustre for writes on average. The read performance speedup of the Burst Buffer over Lustre is $1.7\times$.



(a) Write



(b) Read

Figure 14 Observed write and read rates with DataWarp Burst Buffer and Lustre on Cori as reported by `h5bench`.

4.5 | HDF5 Performance in Permutter (NERSC)

Perlmutter[10] is a heterogeneous system deployed at National Energy Research Scientific Computing Center (NERSC) based on the HPE Cray Shasta platform, comprising both CPU-only and GPU-accelerated nodes. The system is being built in two phases: Phase 1, with 12 GPU-accelerated cabinets housing over $1,500$ nodes and 35PB of all-flash storage delivered by early 2021, and Phase 2, with 12 CPU cabinets currently being assembled. Perlmutter's CPU-only nodes have two sockets of AMD EPYC 7763 (Milan) processors per node, 64 cores per socket, and two threads per code. Each node has 512 GiB of available memory. One Cassini NIC per node, connected via PCIe 4.0. and Slingshot 11 interconnect fabric.

Perlmutter has an all-flash Lustre file system devised for the high-performance storage of large files. It has 35 PB of disk space, an aggregate bandwidth of over 5 TB/sec, and 4 million IOPS (4 KiB random). Its Lustre deployment has 16 MDS (metadata servers), 274 I/O servers called OSSs, and $3,792$ dual-ported NVMe SSDs. All files on Perlmutter are, by default, striped across a single OST with a stripe size of 1 MB.

In Perlmutter, `h5bench` was compiled using PrgEnv-gnu/8.3.3, cray-mpich/8.1.22, gcc/11.2.0 system modules, and HDF5 1.13.2. We ran the baseline `h5bench` access patterns in Perlmutter using collective data and metadata call with 10 seconds emulated compute time per timestep and a total of 5 timesteps. Table 5 summarizes the baseline patterns we have used in our experiments. We set Lustre to stripe the shared file into 1 MB stripes over 256 OSTs. Experiments were conducted with a weak-scaling approach using 1024, 2048, and 4096 ranks, with 64 ranks per node. We compared the baseline results for Perlmutter with those obtained using the VOL-ASYNC and VOL-LOG connectors.

Operation	Dimensions	In-memory	In-file
write	1D, 2D, 3D	contiguous	contiguous
write	1D, 2D	interleaved	contiguous
write	1D, 2D	interleaved	interleaved
read	1D, 2D, 3D	contiguous	contiguous

Table 5 `h5bench` baseline patterns used in our experimental evaluation in Perlmutter.

Figure 15 summarizes the results. For the 1D write scenarios, we observed a speedup of $2.93\times$, $1.38\times$, and $2.20\times$ using 1024, 2048, and 4096 ranks, respectively, using VOL-ASYNC if compared to the synchronous calls when the memory layout is interleaved (i.e., represents an array of

[10]https://docs.nersc.gov/systems/perlmutter/architecture

structure where each array element is a C struct) and the file layout is contiguous. Comparing the synchronous executions with the Log VOL connector, we observed a speedup of $9.58\times$, $7.39\times$, and $11.47\times$ using 1024, 2048, and 4096 ranks, respectively, for this I/O pattern. We also observed a $1.55\times$ speedup while reading contiguously in-memory and in-file with 1024 ranks.

The 2D interleaved in memory and contiguous in file write scenario also benefit from the VOL-ASYNC connector. We observed speedups of $2.75\times$, $2.27\times$, and $2.56\times$ using 1024, 2048, and 4096 ranks, respectively. We also observed similar behavior using Log VOL, an $8.21\times$, $8.64\times$, and $12.36\times$ speedup using 1024, 2048, and 4096 ranks, compared to not using any connector. We also observed a $1.47\times$ speedup while reading contiguously in-memory and in-file with 1024 ranks. For the 3D scenarios, we only observed a significant speedup of 1.23 and 1.18 while reading contiguous in-memory and in-file patterns with 1024 and 2048 processes.

It is important to notice that the observed rate also relies on the amount of transferred data and the parallel file system striping configuration. It is beyond the scope of this paper to identify the best setup and configuration for each pattern or demonstrate all the scenarios where each VOL shines. We instead provide a benchmarking suite capable of aiding system administrators and end-users to asses the expected I/O performance of HDF5-based scientific applications by exercising multiple access patterns and features (e.g., VOL connectors, sub-filling) in a given system. Furthermore, at the time of the experiments, Perlmutter and its Lustre parallel system are not final production-quality resources. As a result, we have observed significant performance variance and encountered different run failures, which are common in systems in construction. Our focus is to show `h5bench` can exercise different access patterns rather than comparing the performances of different VOL connectors.

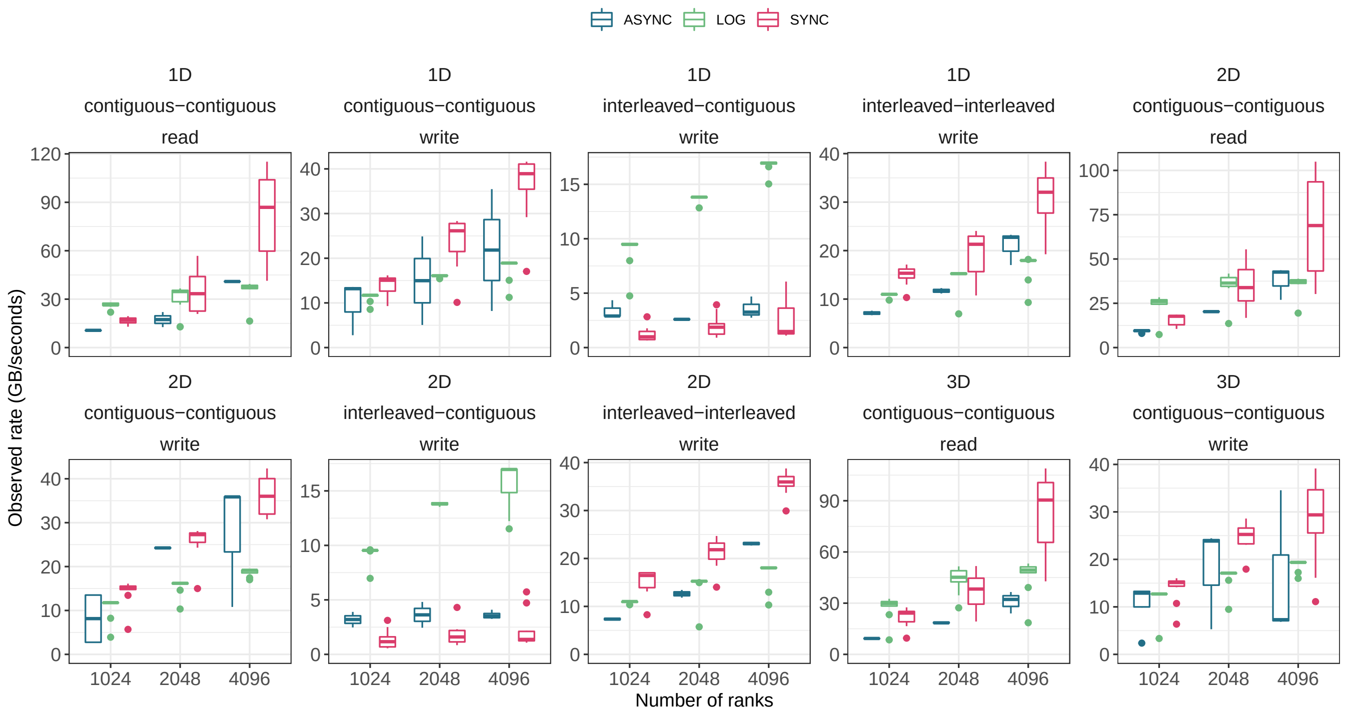


Figure 15 Observed write and read rates for 1D write patterns using SYNC, ASYNC, and LOG VOL connectors on Perlmutter.

The baseline benchmarks also report the time taken with HDF5 metadata operations, more specifically, the time spent in dataset and group creation calls. Figure 16 illustrates the same results now focused on metadata time. Between all the experiments, the LOG VOL connector is the one that reports the lowest (non-zero) time in metadata for all the tested scenarios, while the ASYNC VOL reports more time than the SYNC experiments. However, as demonstrated in Figure 15, ASYNC VOL still manages to improve the overall observed performance. This increase in time compared to others is due to wait times of pending dataset write calls before metadata create operations occur. Because HDF5 uses a global mutex for asynchronous operations, the wait time for metadata calls might include additional unrelated operations' time, as observed in this case.

4.6 | HDF5 Performance in Theta (ANL)

Theta is a Cray XC40 early-access pre-exascale testbed deployed at Argonne Leadership Computing Facility (ALCF). Theta has $4,392$ compute nodes equipped with Intel Knights Landing 7230 processors with 16 GiB of MCDRAM and 192 GiB of DDR4. These nodes have 64 cores each, and each core has 4 SMT hardware threads available.

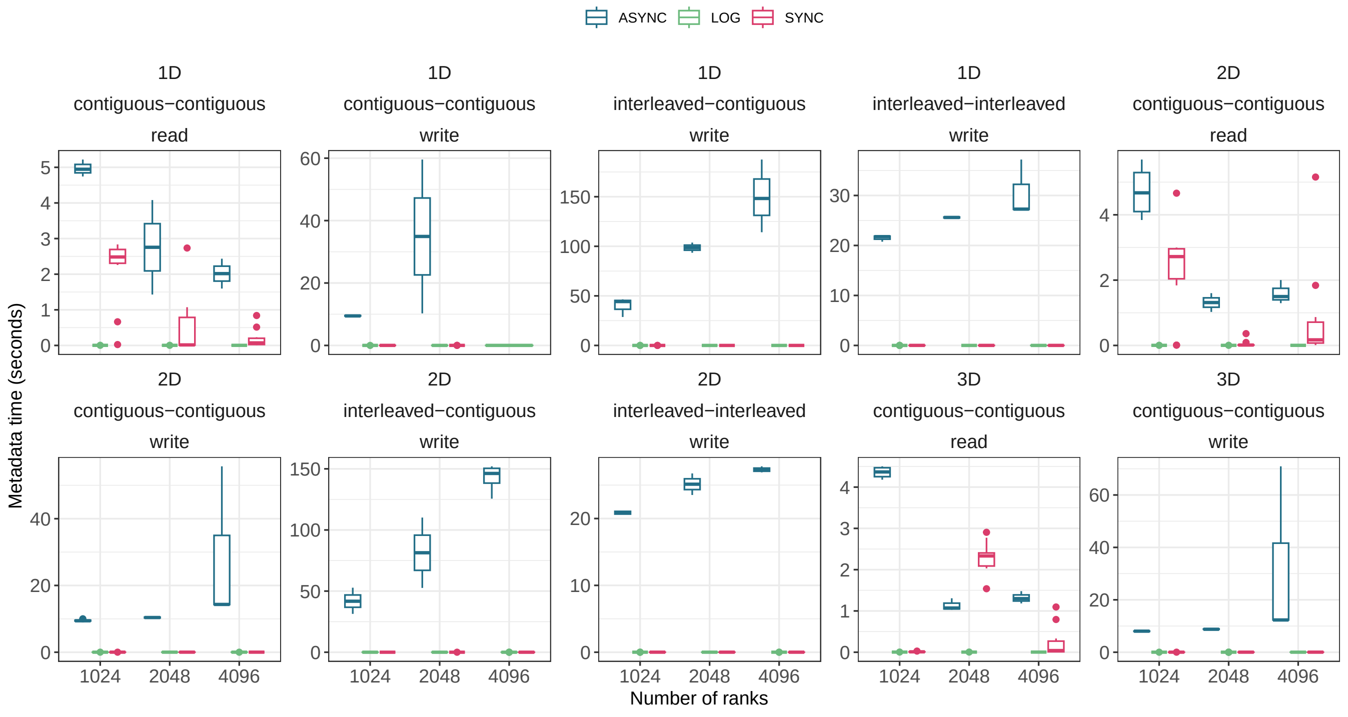


Figure 16 Metadata time for 1D write patterns using SYNC, ASYNC, and LOG VOL connectors on Perlmutter.

Theta is connected to a 100 PB Lustre file system (Grand) residing on an HPE ClusterStor E1000 platform across 8480 disk drives. It provides 160 Object Storage Targets (OSTs) and 40 Metadata Targets (MDT) with an aggregate data transfer rate of 650 GB/s. By default, all files on Grand are striped across a single OST with a stripe size of 1 MB.

`h5bench` was compiled using PrgEnv-intel/6.0.7 and cray-mpich/7.7.14 system modules, and HDF5 1.13.1. We ran the 1D baseline `h5bench` access patterns in Theta using collective data and metadata call with 10 seconds emulated compute time per timestep, and a total of 5 timesteps. Table 6 summarizes the baseline patterns we have used in our experiments. We set Lustre to stripe the shared file into 8 MB stripes over 48 OSTs. Due to queue policy limitations, experiments were conducted using a weak-scaling approach using 64, 128, 256, 512 ranks, and 64 ranks per node. For Theta, we compared the baseline results with the ones obtained using the Async VOL connector.

Operation	Dimensions	In-memory	In-file
write	1D, 3D	contiguous	contiguous
write	1D	contiguous	interleaved
write	1D	interleaved	contiguous
write	1D	interleaved	interleaved
read	1D, 3D	contiguous	contiguous

Table 6 `h5bench` baseline patterns used in our experimental evaluation.

Figure 17 summarizes the results for the 1D data representations. While reading data with the Async VOL connector, the benchmark can attain a $4.45\times$, $3.56\times$, and $2.44\times$ speedup using 64, 128, and 256 processes, respectively. On the other hand, writing data synchronously is the best choice for half of the setups (i.e., ranks + operation + patterns. For the contiguous-interleaved (in-memory and in-file), and interleaved-contiguous patterns, Async VOL achieves speedups of up to $1.31\times$ and $2.06\times$. For the contiguous-contiguous and interleaved-interleaved patterns, we observed a slowdown of up to $0.68\times$ and $0.51\times$.

When considering the 3D scenarios depicted by Figure 18, we observed an improvement of $1.89\times$ when using the Async VOL connector to read data using 512 ranks over eight nodes. To take advantage of this connector, applications must have distinguishable phases comprised of computation/communication and I/O. The first has to be long enough to hide the latency in an I/O phase. When there is insufficient non-I/O time

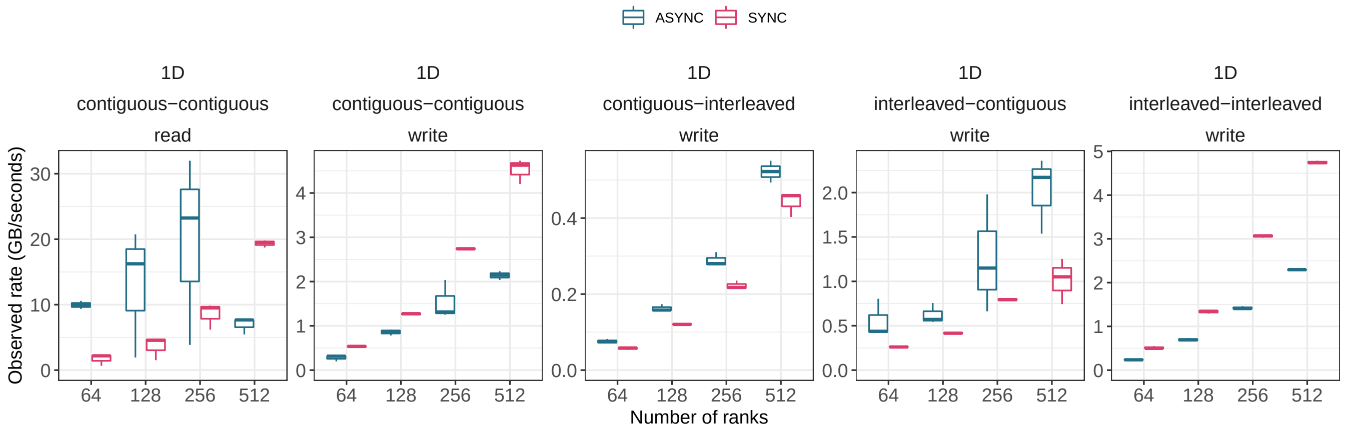


Figure 17 Observed rate for 1D patterns using SYNC and ASYNC VOL connector in Theta.

to overlap, the asynchronous connector may only partially hide the I/O latency. Furthermore, applications should also transfer a significant amount of data to account for the internal overhead of the connector (i.e., copying data to a buffer). For the evaluated contiguous in-memory and in-file 3D pattern, each process transfers ≈ 135MB of data, which results in a small amount of I/O time that is comparable to the Async VOL's write overhead, showing a slower write performance compared to the SYNC version.

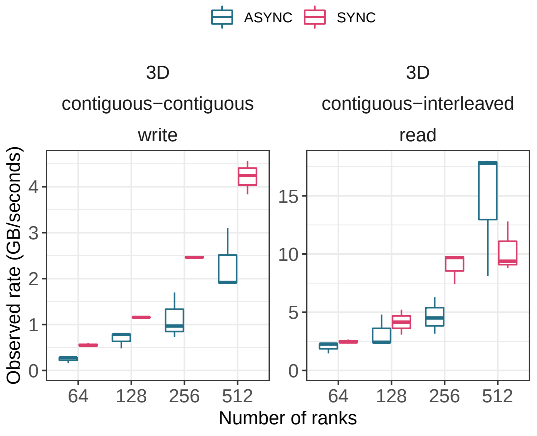


Figure 18 Observed rate for 3D patterns using SYNC and Async VOL connector in Theta.

In Figure 19, we show the performance of HDF5 with CACHE VOL on Theta for 1D contiguous (in-memory) contiguous (in-file) pattern with 8 million particles. The experiments were run with 32 processes per node from 2 to 2048 nodes. The Lustre stripe count is 128, and the stripe size is 16 MB. SSDs were used in Cache VOL. The data were written to SSDs and moved to the Lustre parallel file system asynchronously. The emulated compute time is 100 seconds.

The observed write rate for Cache VOL scales linearly with the number of nodes because more SSDs are used. The observed write rate is basically the aggregate write rate of all the SSDs used because the data migration to the Lustre parallel file system is hidden behind the emulated compute. HDF5 with Cache VOL eventually outperforms the baseline HDF5 (SYNC), reaching 1TB/s at 2048 nodes, whereas the peak write rate for baseline HDF5 is only about 100GB/s at 128 nodes and does not scale well beyond that.

4.7 | HDF5 Performance in Polaris (ANL)

Polaris is a 560-node HPE Apollo 6500 Gen 10+ based system deployed at Argonne Leadership Computing Facility (ALCF). Each node has a single 2.8 GHz AMD EPYC Milan 7543P 32-core CPU with 512 GB of DDR4 RAM and four Nvidia A100 GPUs connected via NVLink, a pair of local 1.6TB of SSDs in RAID0, and a pair of slingshot network adapters. Polaris is also connected to the same 100 PB Lustre file system (Grand) as Theta 4.6. By default, all files on Grand are striped across a single OST with a stripe size of 1MB.

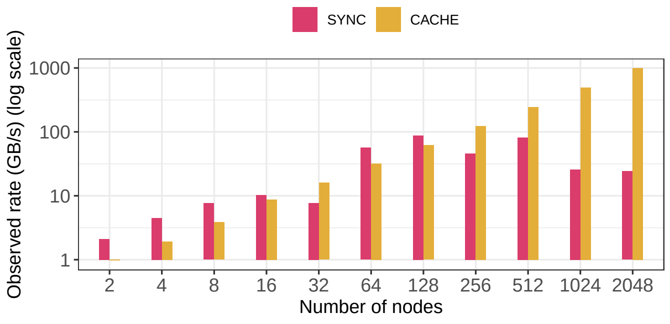
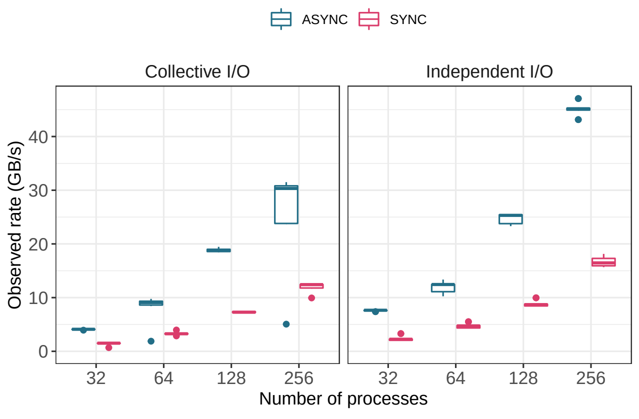


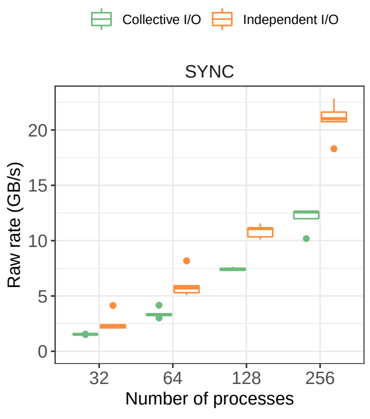
Figure 19 Observed rate for 1D contiguous (in-memory) contiguous (in-file) pattern using SYNC and Cache VOL in Theta.

h5bench was compiled using PrgEnv-nvhpc/8.3.3 and cray-mpich/8.1.16 system modules, and HDF5 1.13.1. We ran the 1D baseline h5bench access patterns in Polaris using independent I/O operations, and collective data and metadata call with 10 seconds emulated compute time per timestep and a total of 5 timesteps. We set Lustre to stripe the shared file into 8MB stripes over 48 OSTs. Experiments were conducted using a weak-scaling approach using 64, 128, 256, 512 ranks, and 32 ranks per node. For Polaris, we compared the baseline results with the ones obtained using the Async VOL connector 2.2.1.

Figure 20(a) depicts the results comprising at least five repetitions using boxplots. The smallest and largest values are found at the end of the whiskers, while the middle line represents the median. Dots depict outliers, i.e., data points located outside the box plot's whiskers and are numerically distant from the rest of the data. Overall, at the evaluated scale, Async VOL can fully overlap h5bench's I/O operations with the emulated computation, achieving a higher observed rate. For example, when using 8 compute nodes (256 ranks, 32 per node) and collective I/O operations enabled by HDF5, Async VOL reaches a median rate of 30.32 GB/s compared to 12.41 GB/s of the synchronous approach, yielding a $2.44\times$ higher transfer. If independent data and metadata operations are used instead, Async VOL reaches a median rate of 45.16 GB/s compared to 16.46 GB/s of the synchronous approach, yielding a $2.74\times$ higher transfer. In Polaris, independent I/O operations result in higher performance than collective I/O under the evaluated setup. For instance, the independent synchronous calls are $1.66\times$ as illustrated by 20(b).



(a)



(b)

Figure 20 (a) Observed rate (i.e., bandwidth) for 1D write patterns using synchronous calls and the Async VOL connector in Polaris. (b) Raw rate (i.e., the bandwidth of the H5Dwrite calls). Due to the scale of the experiments, H5Dwrite_async can transfer all data to a memory buffer using the Async VOL; thus we do not depict those results.

We have selected the best execution (i.e., the one with the fastest runtime) to break down its runtime into compute, data, and metadata times for each experiment as illustrated by Figure 21. The white box represents the makespan. The total emulated compute time of 40 seconds (10 seconds between each of the 5 iterations) is in yellow. The raw time measures the H5Dwrite and H5Dwrite_async I/O calls, whereas the metadata time encompasses HDF5 metadata calls. When using synchronous calls, the raw time takes most of the I/O time of the application (the metadata is barely visible). On the other hand, due to the scale of the experiments, H5Dwrite_async is able to transfer all data to a memory buffer within the ASYNC VOL.

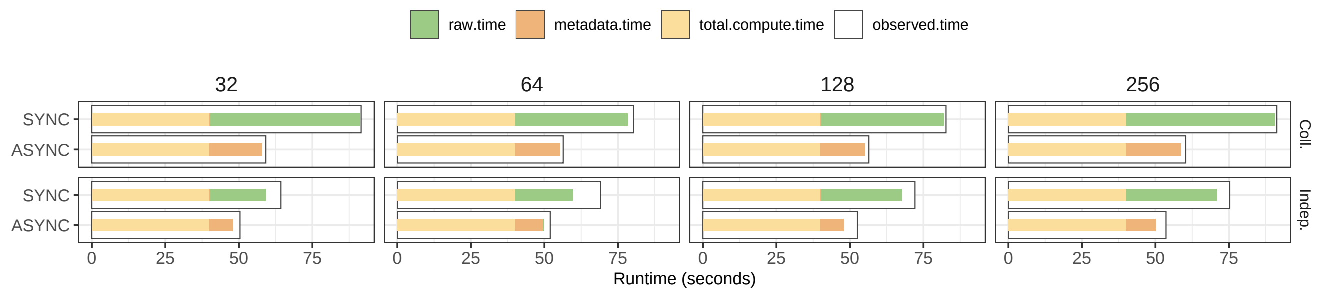


Figure 21 Runtime breakdown of h5bench on Polaris using collective and individual I/O calls and the synchronous and Async VOL connector.

h5bench also records the file creation (H5Fcreate), file flush (H5Fflush) and file close (H5Fclose) times. Figure 22 reports these metrics for each tested scenario. In all the experiments, file creation and closing have a minimal contribution to the total runtime. H5Fcreate takes on average 0.37 seconds, an a maximum of $1.4s$ in Polaris. H5Fclose was observed to take at most 17 milliseconds. On the other hand, H5Fflush, which is used for flushing all unwritten data to an HDF5 file, takes longer than the others. When using independent I/O calls, flush takes longer, especially when not using the Async VOL connector.

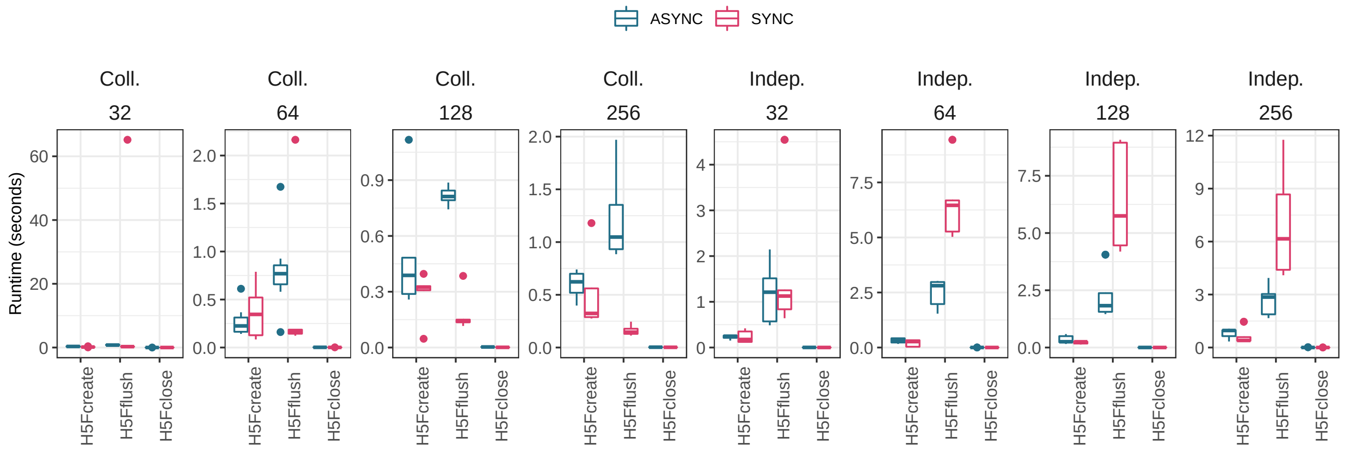


Figure 22 Runtime breakdown of file creation, flush, and close times as reported by h5bench.

In Figure 23, we show the performance of HDF5 with Cache VOL on Polaris for 1D contiguous (in-memory) contiguous (in-file) pattern with 8 million particles. The experiments were run with 32 processes per node from 2 to 2048 nodes. The Lustre stripe count is 128, and the stripe size is 16 MB. NVMe SSDs were used in Cache VOL. The data were written to NVMe SSDs and moved to the Lustre parallel file system asynchronously. The emulated compute time is 100 seconds.

Similar to the case on Theta, the observed write rate for Cache VOL scales linearly with the number of nodes. It outperforms the baseline HDF5 (SYNC) even at a small scale because the NVMe SSDs have high write bandwidth ($4-5$GB/s per node), which is higher than the bandwidth of SSDs on Theta ($500-600$ MiB/s per node).

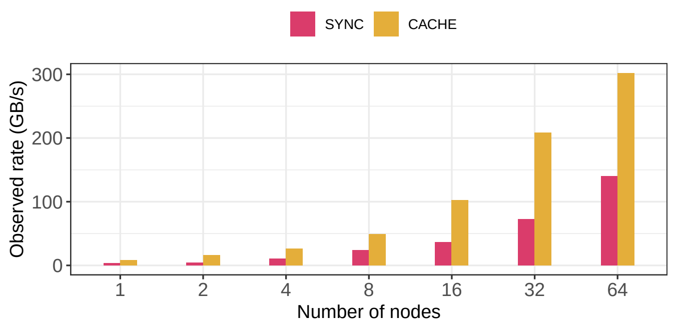


Figure 23 Observed rate for 1D contiguous (in-memory) contiguous (in-file) pattern using synchronous I/O calls and Cache VOL in Polaris.

5 | CONCLUSION

In this paper, we introduce h5bench, a unified suite of benchmarks for performing writing and reading data following various I/O patterns using the HDF5 API. The suite comprises baseline patterns representing I/O operations considering data structures in memory and file layout, multi-dimensional arrays (1D, 2D, and 3D), and scientific I/O kernels from multiple science domains. In addition, h5bench can evaluate novel HDF5 functionality such as VOL connectors (e.g., Async, Cache, Log). For example, Async VOL hides a significant portion of the I/O latency by overlapping I/O phases with computation phases. Cache VOL allows caching or staging data on node-local storage and then moving data asynchronously between the node-local storage and parallel file system. Log VOL stores write requests in contiguous file spaces in a log layout, avoiding the potentially expensive inter-process communication required if data is stored in a canonical order. In summary, our focus in this paper is to show that h5bench can exercise different access patterns rather than comparing the performances of different VOL connectors. Users can use the h5bench suite by setting various configuration parameters based on which patterns, HDF5 features, and large-scale systems they want to exercise.

Our performance evaluation of h5bench covered different dimensions of read and write kernels on Cori and on Summit. We also investigated the performance of h5bench by exercising different access patterns and HDF5 features in three new pre-exascale platforms: Perlmutter, Theta, and Polaris. h5bench measurements can be used to identify performance bottlenecks and their root causes and evaluate I/O optimizations. Further optimizations can be applied as these pre-production systems become available. As the I/O patterns of h5bench are diverse and capture the I/O behaviors of HPC applications from various science domains, these features will be helpful to the broader supercomputing and I/O community.

The h5bench suite 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. It can also be easily installed with Spack (`spack install h5bench`), and it is available as part of the Extreme-scale Scientific Software Stack (E4S)[21] deployed at multiple supercomputing facilities (e.g., NERSC, JLSE, OLCF). In future, we plan to support additional features (e.g., compression, unified performance report, breakdown of data vs. metadata) and provide a unified reporting API and score to follow results and track changes in exascale system's performance.

ACKNOWLEDGEMENTS

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 work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0021411. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. 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 . Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5; 1997-.
2. Folk M, Heber G, Koziol Q, Pourmal E, Robinson D. An overview of the HDF5 technology suite and its applications. In: ACM. Association for Computing Machinery; 2011: 36–47.
3. Rew R, Davis G. netCDF: An interface for Scientific Data Access. In: . 10. IEEE. ; 990: 76–82.
4. Antcheva I, Ballintijn M, Bellenot B, et al. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications* 2009; 180(12): 2499-2512. 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architecturesdoi: https://doi.org/10.1016/j.cpc.2009.08.005
5. Li J, Liao kW, Choudhary A, et al. Parallel netCDF: A High-Performance Scientific I/O Interface. In: IEEE. ; 2003: 39-39
6. Thakur R, Gropp W, Lusk E. On Implementing MPI-IO Portably and with High Performance. In: ACM. ; 1999: 23–32.
7. Kunkel JM, Bent J, Kunkel J, Kunkel GS. Establishing the IO-500 Benchmark. https://www.vi4io.org/_media/io500/about/io500-establishing.pdf; 2016.

8. Tang H, Koziol Q, Byna S, Mainzer J, Li T. Enabling Transparent Asynchronous I/O using Background Threads. In: IEEE. ; 2019: 11-19

9. Tang H, Koziol Q, Byna S, Ravi J. Transparent Asynchronous Parallel I/O using Background Threads. *IEEE Transactions on Parallel and Distributed Systems* 2021: 1-1. doi: 10.1109/TPDS.2021.3090322

10. Byna S, Breitenfeld M, Dong B, et al. ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems. *JCST* 2020; 35: 145-160.

11. Miller MC. Multi-Purpose, Application-Centric, Scalable I/O Proxy Application, Version 00. https://www.osti.gov/biblio/1232293; 2015.

12. Byna S, Howison M. Parallel I/O Kernel (PIOK) Suite. https://sdm.lbl.gov/exahdf5/software.html; 2015.

13. Computational Science fFC. The FLASH code. http://flash.uchicago.edu/site/flashcode/; .

14. Colella P, Graves DT, Johnson JN, et al. Chombo software package for AMR applications design document. Technical Report LBNL-6616E. tech. rep., Lawrence Berkeley National Laboratory; United States: 2003.

15. Zhang W, Almgren A, Beckner V, et al. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 2019; 4(37): 1370. doi: 10.21105/joss.01370

16. Zheng H, Vishwanath V, Koziol Q, et al. HDF5 Cache VOL: Efficient and Scalable Parallel I/O through Caching Data on Node-local Storage. In: IEEE. ; 2022: 61-70

17. Henseler D, Landsteiner B, Petesch D, Wright C, Wright NJ. Architecture and Design of Cray DataWarp. In: Cray User Group. ; 2016.

18. HDF5 Virtual Object Layer (VOL) User Guide. https://github.com/HDFGroup/hdf5doc/blob/master/RFCs/HDF5/VOL/connector_author_guide/vol_conne
2020.

19. Seo S, Amer A, Balaji P, et al. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 2018; 29(3): 512-526. doi: 10.1109/TPDS.2017.2766062

20. Kimpe D, Ross R, Vandewalle S, Poedts S. Transparent Log-Based Data Storage in MPI-IO Applications. In: Cappello F, Herault T, Dongarra J., eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*Springer Berlin Heidelberg. ; 2007; Berlin, Heidelberg: 233–241.

21. Heroux MA, McInnes LC, Thakur R, et al. ECP Software Technology Capability Assessment Report. tech. rep., USDOE Office of Science (SC); United States: 2020

22. Wu K, Byna S, Dong B, USDOE . VPIC IO Utilities. https://www.osti.gov/biblio/1487266; 2018

23. Byna S. BD-CATS-IO, Version 00. https://www.osti.gov/biblio/1459439; 2017.

24. Byna S, Chou J, Rübel O, Prabhat , Karimabadi H, others . Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation. In: IEEE. ; 2012: 59:1–59:12.

25. Byna S, Chou J, Rübel O, et al. "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation". In: IEEE. ; 2012: 59:1–59:12.

26. Almgren AS, Bell JB, Lijewski MJ, Lukić Z, Van Andel E. Nyx: A Massively Parallel AMR Code for Computational Cosmology. *The Astrophysical Journal* 2013; 765: 39. doi: 10.1088/0004-637X/765/1/39

27. Kim Y, Gunasekaran R, Shipman GM, Dillow DA, Zhang Z, Settlemyer BW. Workload characterization of a leadership class storage cluster. In: IEEE. ; 2010

28. Ibtesham D, Arnold D, Ferreira KB, Bridges PG. On the Viability of Checkpoint Compression for Extreme Scale Fault Tolerance. In: Springer-Verlag. ; 2012; Berlin, Heidelberg: 302–311.

29. Ibtesham D, Arnold D, Bridges PG, Ferreira KB, Brightwell R. On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-Based Fault Tolerance. In: ICPP '12. IEEE Computer Society. ; 2012; USA: 148–157

30. Huebl, Axel and Lehe, Remi and Vay, Jean-Luc and Grote, David P. and Sbalzarini, Ivo F. and Kuschel, Stephan and Sagan, David and Mayes, Christopher and Perez, Frederic and Koller, Fabian and Bussmann, Michael . openPMD: A meta data standard for particle and mesh based data. https://doi.org/10.5281/zenodo.1167843; 2015.

31. Koller, Fabian and Poeschel, Franz and Gu, Junmin and Huebl, Axel . openPMD-api 0.10.3: C++ & Python API for Scientific I/O with openPMD. https://doi.org/10.14278/rodare.209; 2019. DOI: 10.14278/rodare.209.

32. Zhang W, Almgren A, Beckner V, et al. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 2019; 4(37): 1370. doi: 10.21105/joss.01370

33. Plewa T, Linde T, Weirs V. *Adaptive Mesh Refinement – Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3-5, 2003*. Lecture Notes in Computational Science and EngineeringSpringer Berlin Heidelberg . 2004.

34. E3SM Project D. Energy Exascale Earth System Model v1.3. [Computer Software] https://doi.org/10.11578/E3SM/dc.20210924.5; 2019

35. Leung LR, Bader DC, Taylor MA, McCoy RB. An Introduction to the E3SM Special Collection: Goals, Science Drivers, Development, and Analysis. *Journal of Advances in Modeling Earth Systems* 2020; 12(11): e2019MS001821. e2019MS001821 2019MS001821doi: https://doi.org/10.1029/2019MS001821

36. Cray . Cray Data Warp Web Page. http://www.cray.com/products/storage/datawarp; 2012.

37. Lofstead J, Polte M, Gibson G, et al. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In: HPDC '11. ACM. Association for Computing Machinery; 2011; New York, NY, USA: 49–60

38. Patwary MMA, Byna S, Satish NR, et al. BD-CATS: big data clustering at trillion particle scale. In: IEEE. ; 2015: 1-12

39. Bez JL, Tang H, Xie B, et al. I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis. In: IEEE. ; 2021: 15-22

How to cite this article: J. L. Bez, H. Tang, and M.S. Breitenfeld, and H. Zheng, and W. Liao, and K. Hou, and S. Byna (2022), `h5bench`: Exploring HDF5 Access Patterns Performance in Pre-Exascale Platforms, *Concurrency and Computation: Practice and Experience*, *2022;00:1–6*.