ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems

Suren Byna[1,*], M. Scot Breitenfeld[2], Bin Dong[1], Quincey Koziol[1], Elena Pourmal[2], Dana Robinson[2]
Jerome Soumagne[2], Houjun Tang[1], Venkatram Vishwanath[3], and Richard Warren[2]

[1] *Lawrence Berkeley National Laboratory, Berkeley, CA 94597, U.S.A.*
[2] *The HDF Group, Champaign, IL 61820, U.S.A.*
[3] *Argonne National Laboratory, Lemont, IL 60439, U.S.A.*

E-mail: sbyna@lbl.gov; brtnfld@hdfgroup.org; {dbin, koziol}@lbl.gov
{epourmal, derobins, jsoumagne}@hdfgroup.org; HTang4@lbl.gov; venkat@anl.gov
Richard.Warren@hdfgroup.org

Abstract Scientific applications at exascale generate and analyze massive amounts of data. A critical requirement of these applications is the capability to access and manage this data efficiently on exascale systems. Parallel I/O, the key technology enables moving data between compute nodes and storage, faces monumental challenges from new applications, memory, and storage architectures considered in the designs of exascale systems. As the storage hierarchy is expanding to include node-local persistent memory, burst buffers, etc., as well as disk-based storage, data movement among these layers must be efficient. Parallel I/O libraries of the future should be capable of handling file sizes of many terabytes and beyond. In this paper, we describe new capabilities we have developed in Hierarchical Data Format version 5 (HDF5), the most popular parallel I/O library for scientific applications. HDF5 is one of the most used libraries at the leadership computing facilities for performing parallel I/O on existing HPC systems. The state-of-the-art features we describe include: Virtual Object Layer (VOL), Data Elevator, asynchronous I/O, full-featured single-writer and multiple-reader (Full SWMR), and parallel querying. In this paper, we introduce these features, their implementations, and the performance and feature benefits to applications and other libraries.

1 Introduction

In pursuit of more accurate modeling of real-world systems, scientific applications at exascale will gene-rate and analyze massive amounts of data. A critical requirement of these applications is the capability to access and manage this data efficiently on exascale systems. Parallel I/O, the key technology behind moving data between compute nodes and storage, faces monumental challenges from the new application workflows as well as the memory, interconnect, and storage architectures considered in the designs of exascale systems. The storage hierarchy in existing pre-exascale computing systems includes node-local persistent mem-

This research was supported by the Exascale Computing Project under Grant No. 17-SC-20-SC, a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. This work is also supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract Nos. DE-AC02-05CH11231 and DE-AC02-06CH11357. This research was funded in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract No. DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, which is DOE Office of Science User Facilities supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

ory, burst buffers, etc., as well as disk and tape-based storage①②. Data movement among these layers must be efficient, with parallel I/O libraries of the future capable of handling file sizes of many terabytes and beyond. Easy-to-use interfaces to access and move data are required for alleviating the burden on scientific application developers and in improving productivity. Exascale I/O systems must also be fault-tolerant to handle the failure of compute, network, memory, and storage, given the number of hardware components at these scales.

Intending to address efficiency, fault-tolerance, and other challenges posed by data management and parallel I/O on exascale architectures, we are developing new capabilities in HDF5 (Hierarchical Data Format version 5)[1], the most popular parallel I/O library for scientific applications. As shown in Fig.1, HDF5 is the most used library for performing parallel I/O on existing HPC systems at the National Energy Research Scientific Computing Center (NERSC). (HDF5 and hdf5 are used interchangeably) HDF5 is among the top libraries used at several US Department of Energy (DOE) supercomputing facilities, including the Leadership Computing Facilities (LCFs). Many of the exascale applications and co-design centers require HDF5 for their I/O, and enhancing the HDF5 software to handle the unique challenges of exascale architectures will play an instrumental role in the success of the Exascale Computing Project (ECP)④.

ExaHDF5 is a project funded by ECP in enhancing the HDF5 library. In this paper, we describe new capabilities we have developed in the ExaHDF5 project, including Virtual Object Layer (VOL), Data Elevator, asynchronous I/O, full single-writer and multiple-reader (SWMR), and parallel querying. We describe each of these features and present an evaluation of the benefits of the features.

There are multiple I/O libraries, such as PnetCDF[2] and ADIOS[3], which provide parallel I/O functionality. However, this paper is focusing on implementations in the HDF5 library. The remainder of the paper is organized as follows. We provide a brief background to HDF5 in Section 2 and describe the features enhancing HDF5 in Section 3. In Section 4, we describe our experimental setup used for evaluating various developed features using different benchmarks (presented in Section 5) and conclude our discussion in Section 6.

We have presented the benefits of the developed features individually as these have been developed for different use cases. For instance, Data Elevator is for using burst buffers on systems that have the hardware. Full SWMR is for applications that require multiple readers analyzing data while a writer is adding data to an HDF5 file. Integrating all these features into the single HDF5 product requires a greater engineering effort, which will occur in the next phase of the project.

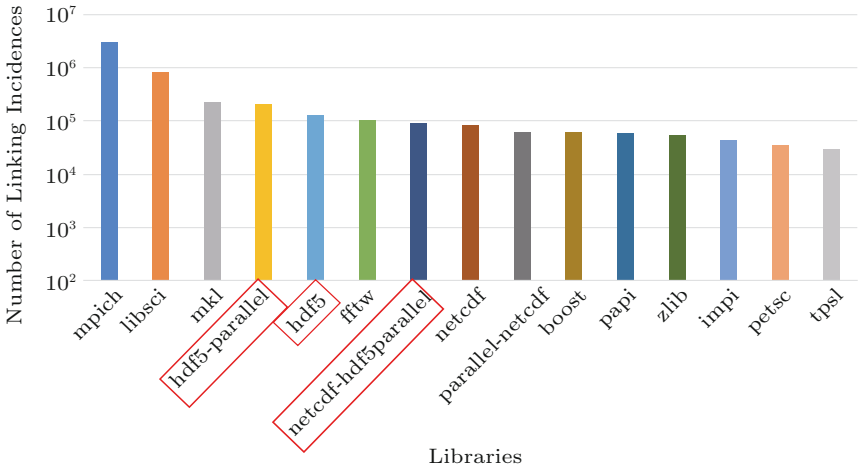

Fig.1. Library usage at NERSC on Cori and Edison in 2017 using the statistics collected by Automatic Library Tracking Database (ALTD)③.

①OLCF Summit: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, Nov. 2019.

②Cori at NERSC: https://www.nersc.gov/users/computational-systems/cori/, Nov. 2019.

③https://www.nersc.gov/assets/altdatNERSC.pdf, Dec. 2019.

④ECP homepage: https://www.exascaleproject.org/, Nov. 2019.

2 Background to HDF5

HDF5 provides a data model, library, and file format for storing and managing data. Due to the versatility of its data model, and its portability, longevity, and efficiency, numerous scientific applications use HDF5 as part of their data management solution. The HDF Group (THG)[5] has been designing, developing, and supporting the HDF5 software for more than 20 years.

HDF5 is an open-source project with a BSD-style license and comes with C, Fortran, C++, and Java APIs, and third-party developers offer API wrappers in Python, Perl, and many other languages. The HDF5 library has been ported to virtually all existing operating systems and compilers, and the HDF5 file format is portable and machine-independent. The HDF5 community contributes patches for bugs and new features, and actively participates in testing new public releases and feature prototypes.

HDF5 is designed to store and manage high-volume and complex scientific data, including experimental and observational data (EOD). HDF5 allows storing generic data objects within files in a self-describing way, and much of the power of HDF5 stems from the flexibility of the objects that make up an HDF5 file: *datasets* for storing multi-dimensional arrays of homogeneous elements and *groups* for organizing related objects. HDF5 *attributes* are used to store user-defined metadata on objects in files.

The HDF5 library offers several powerful features for managing data. The features include random access to individual objects, partial access to selected dataset values, and internal data compression. HDF5 allows the modification of datasets values, including compressed data, without rewriting the whole dataset. Users can use custom compression methods and other data filtering techniques that are most appropriate for their data, for example, ZFP compression developed at LLNL[6] along with the compression methods available in HDF5. Data extensibility is another powerful feature of the HDF library. Data can be added to an array stored in an HDF5 dataset without rewriting the whole dataset. Modifications of data and metadata stored in HDF5 can be done in both sequential mode and parallel mode using MPI I/O.

HDF5 supports a rich set of pre-defined datatypes as well as the creation of an unlimited variety of complex user-defined datatypes. User-defined datatypes provide a powerful and efficient mechanism for describing users' data. Datatype definition includes information about byte order, size, and floating point representation; it fully describes how the data is stored in the file, insuring portability between the different operating system and compilers.

Due to the simplicity of the HDF5 data model, and flexibility and efficient I/O of the HDF5 library, HDF5 supports all types of digitally stored data, regardless of origin or size. Petabytes of remote sensing data collected by satellites, terabytes of computational results from nuclear testing models, and megabytes of high-resolution MRI brain scans are stored in HDF5 files, together with metadata necessary for efficient data sharing, processing, visualization, and archiving.

3 Features

Towards handling challenges posted by massive concurrency of exascale computing systems, deeper storage hierarchies of these systems, and large amounts of scientific data produced and analyzed, several new features are needed in enhancing HDF5. In the ExaHDF5 project, we have been working on opening the HDF5 API to allow various data storage options, on providing transparent utilization of deep storage hierarchy, on optimizing data movement, on providing coherent and concurrent access to HDF5 data by multiple processes, and on accessing desired data efficiently. In the remainder of this section, we will describe the features addressing these aspects of improving HDF5.

3.1 Virtual Object Layer (VOL)

The HDF5 data model is composed of two fundamental objects: groups and datasets, composed as a directed graph within a file (sometimes also called a "container"). This data model is powerful and widely applicable to many applications, including HPC applications. However, the classic HDF5 mechanism of storing data model objects in a single, shared file can be a limitation as HPC systems evolve toward a massively concurrent future. Completely re-designing the HDF5 file format would be a possible way to address this limitation, but the software engineering challenges of supporting two (or more, in the future) file formats directly in the HDF5 library are daunting. Ideally, applications would continue to use the HDF5 programming API and

[5] The HDF Group homepage: https://www.hdfgroup.org/, Nov. 2019.

[6] LZF Website at LLNL: https://computation.llnl.gov/projects/floating-point-compression, Nov. 2019.

data model, but have those objects stored with the storage mechanism of their choice.

As shown in Fig.2, the Virtual Object Layer (VOL) adds a new abstraction layer internally to the HDF5 library and is implemented just below the public API. The VOL intercepts all HDF5 API calls that access objects in a file and forwards those calls to an "object driver" connector. A VOL connector can store the HDF5 data model objects in a variety of ways. For example, a connector could distribute objects remotely over different platforms, provide a direct mapping of the model to the file system, or even store the data in another file format (like the native netCDF or HDF4 format). The user still interacts with the same HDF5 data model and API, where access is done to a single HDF5 "container"; however, the VOL connector translates those operations appropriately to access the data, regardless of how it is actually stored.

While HDF5 function calls can be intercepted by binary instrumentation libraries such as *gotcha*[7], the VOL feature in HDF5 is designed to work with multiple VOL connectors based on the user requirement. This VOL connector stackability allows users to register and unregister multiple connectors. For example, Data Elevator can be stacked with the asynchronous I/O connector and a provenance collection VOL connector. Since each VOL connector serves a different purpose, they can be developed as different modules by different developers and can be stacked to work together. This type of stackability is not feasible with other binary instrumentation libraries. Having the VOL implementation within HDF5 reduces the dependency on external libraries not maintained by the HDF5 developers.

In our recent work, we have integrated the VOL feature implementation into the mainstream HDF5 library and released it in HDF5 version 1.12.0. The VOL feature implementation included new HDF5 API to register (`H5VLregister*`), to unregister (`H5VLunregister*`), and to obtain the VOL information (`H5VLget*`). We have also developed a "pass through" VOL connector as an example implementation that forwards each VOL callback to an underlying connector.

Our implementation of VOL allows VOL connectors widely accessible to the HDF5 community and will encourage more developers to use or create new connectors for the HDF5 library. The VOL implementation is currently available in the "develop" branch as we write this paper[8], which is available for public access. The HDF Group is working on releasing the feature as the next major public release. Several VOL connectors have been or are under development but

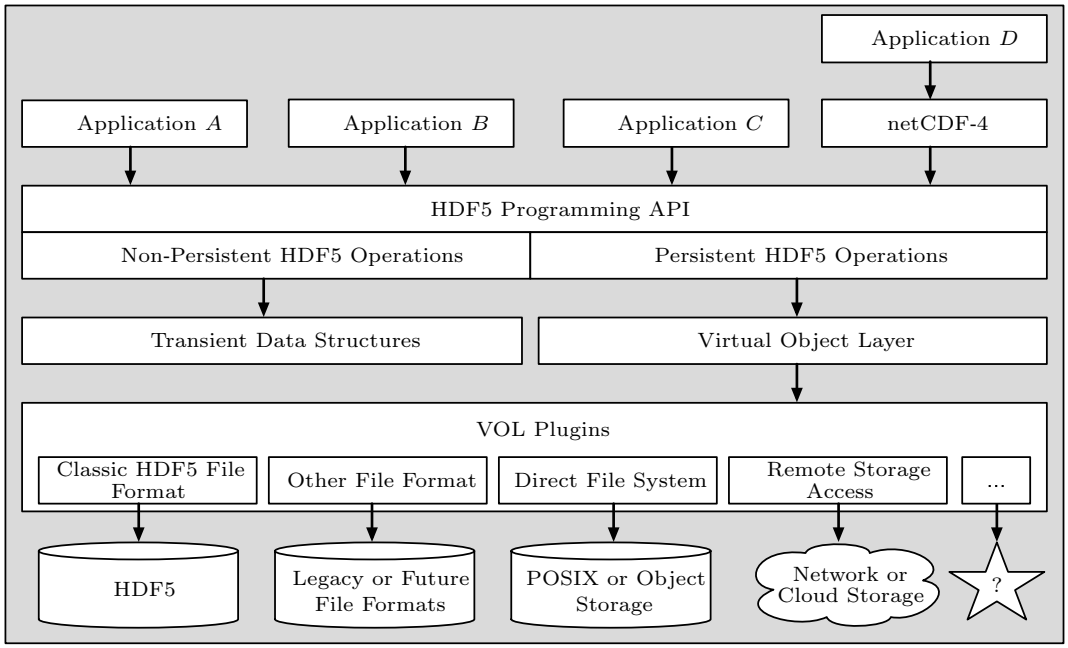

Fig.2. Overview of the Virtual Object Layer (VOL) architecture within HDF5.

[7] https://github.com/llnl/gotcha, Nov. 2019.

[8] HDF5 develop branch: https://bitbucket.hdfgroup.org/projects/HDFFV/repos/hdf5, Nov. 2019.

cannot be officially released until the VOL architecture is finalized and integrated with the main HDF5 library. These VOL connectors in development include those aiming to map HDF5 natively on top of object storage, to support remote HDF5 object storage, and to track statistics of HDF5 usage, to name a few. We present two VOL connectors in this paper, i.e., the Data Elevator connector (Subsection 3.2) for transparently moving data between multiple layers of storage and the asynchronous I/O VOL connector (Subsection 3.3) for moving data asynchronously by letting the computation process continue while a thread moves the data. The VOL interface will also enable developers to create connectors for representing HDF5 objects in multiple types of non-persistent and persistent memories, to support workflows that take advantage of in-memory data movement between different codes.

3.2 Data Elevator

Key synergistic criterion to achieve scalable data movement in scientific workflows is the effective placement of data. The data generation (e.g. by simulations) and consumption (such as for analysis) can span various storage and memory tiers, including near-memory NVRAM, SSD-based burst buffers, fast disk, campaign storage, and archival storage. Effective support for caching and prefetching data based on the needs of the application is critical for scalable performance.

Solutions such as Cray DataWarp[9], DDN Integrated Memory Engine (IME), and parallel file systems such as Lustre and GPFS provide solutions for a specific storage layer, but it is currently left to the users to move data between different layers. It is imperative that we design parallel I/O and data movement systems that can hide the complexity of caching and data movement between tiers from users, to improve their productivity without penalizing performance.

Towards achieving the goal of an integrated parallel I/O system for multi-level storage, as part of the ExaHDF5 research project, we have recently developed the Data Elevator library [4]. Data Elevator intercepts HDF5 write calls, caches data in the fastest persistent storage layer, and then moves it to the final destination specified by an application in the background. We have used the HDF5 Virtual Object Layer (VOL) (described in Subsection 3.1) to intercept HDF5 file open, dataset open and close, dataset write and read, and file close calls. In addition to HDF5, we have extended the Data Elevator library to intercept MPI-IO data write calls. This extension will cover a large number of applications to use the library. For supporting data reads, we developed prefetching and caching data in a faster storage layer. To support data reads, data is prefetched and cached in a faster storage layer. We describe the write and read caching methods in this subsection.

3.2.1 *Write Caching in Data Elevator*

In Fig.3, we show a high-level overview of the write and read caching implementations of Data Elevator. The main issues we target to address with DE are 1) to provide a transparent mechanism for using a burst buffer (BB) as a part of a hierarchical storage system, and 2) to move data between different layers of a hierarchical storage system efficiently with low resource contention on BB nodes. The data stored temporarily in the burst buffer can be reused for any analysis. Hence, we call this feature caching instead of simply buffering the data.

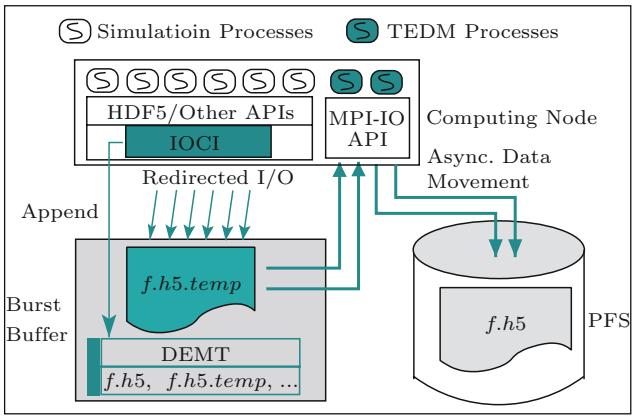

Fig. 3. Overview of Data Elevator write caching functionality. The arrows from IOCI to DEMT are control requests and the the remaining arrows are requests for data. IOCI: I/O Call Interceptor. DEMT: Data Elevator Metadata Table.

The first issue arises because burst buffers are introduced in HPC systems as independent storage spaces. Burst buffers that are being considered in exascale systems are of two types: those provide a single namespace and are shared by all the compute nodes, and those installed on each node and do not provide a single namespace. In the Data Elevator, our implementations are on burst buffers that are shared by all compute nodes. Due to limited storage capacity, i.e., 2x to 4x the main memory size, burst buffers are typically available to users only during the execution of their programs. Consequently, if users choose to write data to BB, they

[9] Cray DataWarp: http://docs.cray.com/books/S-2558-5204/S-2558-5204.pdf, Nov. 2019.

are also responsible for moving the data to PFS for retaining the data. The second issue is caused by Cray DataWarp, the state-of-the-art middleware for managing burst buffers on Cray systems. DataWarp uses a fixed and small number of BB nodes to serve both regular I/O and data movement requests. This typically results in performance degradation caused by interference between the two types of requests.

To address above issues, Data Elevator performs asynchronous data movement to enable transparent use of hierarchical data storage, and also to use a low-contention data flow path by using compute nodes for transferring data between different storage layers. Data Elevator allows one to use as many data transfer nodes as necessary, which reduces the chance of contention. As shown in Fig.3, Data Elevator has three main components: I/O Call Interceptor (IOCI), Data Elevator Metadata Table (DEMT), and Transparent and Efficient Data Mover (TEDM or Data Mover in short).

The IOCI component intercepts I/O calls from applications and redirects I/O to fast storage, such as burst buffer. We implement IOCI mechanism using the HDF5 VOL feature, by developing an HDF5 VOL plug-in. While the current implementation supports HDF5-based I/O, the implementation can be extended to other I/O interfaces, such as MPI-IO. DEMT contains a list of metadata records, e.g., file name, for the files to be redirected to BB. The Data Mover (TEDM) component is responsible for moving the data from a burst buffer to a PFS. TEDM component can share the nodes with the application job or run using a separate set of compute nodes. In Fig.3, TEDM shares two of the eight CPU cores with a simulation job (that uses the remaining six cores) on a computing node.

To reduce resource contention on BB during data movement, Data Elevator is instantiated either on separate compute nodes or on the same compute nodes as an application. While this design of Data Elevator increases the number of CPU cores needed for running an application, extensive test results show that using a small portion of computing power to optimize I/O performance reduces the end-to-end time of the whole I/O intensive simulation. When the data is in BB, Data Elevator also allows to perform transit analysis tasks on the data, before it is moved to the final destination of the file specified by the application.

Data Elevator applies various optimizations to improve I/O performance in writing data to BB. After the application finishes writing a data file, it can continue computations without waiting for the data to be moved to a PFS. Meanwhile, the Data Mover monitors the metadata periodically and once it finds that the writing process is complete, it starts to move the data from the BB to the PFS. Data Elevator reads the data from the BB to the memory on computing nodes, where Data Elevator is running, and writes the data to PFS without interfering with other I/O requests on the BB. Data Elevator provides optimizations, such as overlapping of reading data from BB to memory and writing to the PFS, and aligning Lustre PFS stripe size with the data request size (assuming PFS is Lustre) to reduce the overhead of data movement. While the data is in the burst buffer, Data Elevator allows data analysis codes to access the data, which is called in situ or in transit analysis, by redirecting data read accesses to the data stored in the burst buffer.

3.2.2 Read Caching in Data Elevator

In read caching, we have implemented caching and prefetching chunks of data based on the history of HDF5 chunk accesses [5]. The cached chunks are stored as binary files in a faster persistent storage device, such as an SSD-based burst buffer. When the requests to the prefetched data come, DE redirects the read requests to the cached data and thus improving performance significantly.

The data flow path for read caching in Data Elevator is shown in Fig.4. We highlight the prefetching function of Data Elevator that improves the performance of array-based analysis programs by reading ahead data from the disk-based file system to the SSD-based burst buffer. An array is prefetched as data chunks. In the figure, we show an array with 12 (i.e., a 3×4 array) chunks. Let us assume that an analysis task is accessing the first two chunks (marked in green) that are read from the disk-based file system.

Data Elevator's Array Caching and Prefetching method (ARCHIE) prefetches the following four chunks (shown in blue) into the SSDs for future reads. Data Elevator manages the metadata table to contain data access information extracted from applications. Entries of the metadata table contain the name of the file being accessed by a data analysis application. This filename information is inserted into the table by the read function linked with analysis applications. The metadata table also contains "chunk size", "ghost zone size", and "status of the cached file". A Chunk access predictor component uses the information of the cached chunks to predict the future read accesses of analysis applications and to prefetch the predicted data chunks into

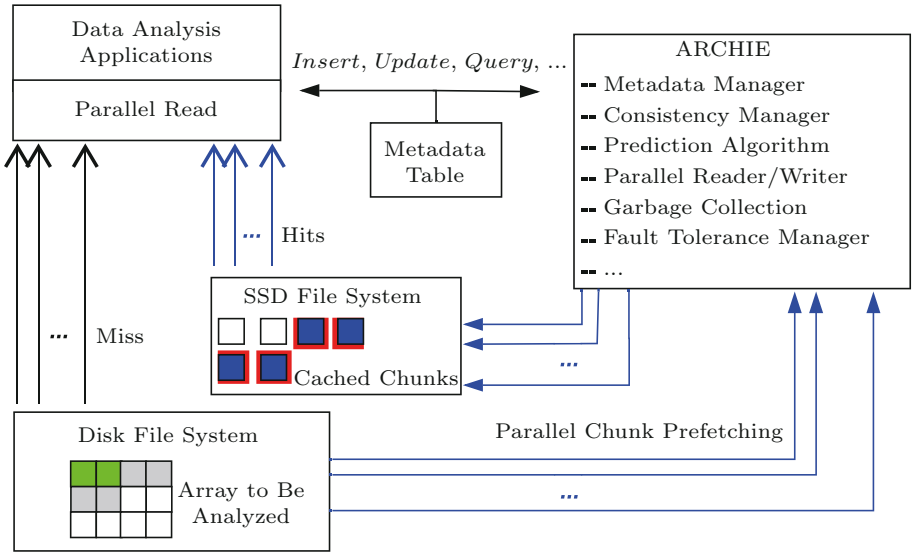

Fig. 4. High level description of Data Elevator read caching and prefetching functions. All the arrows to and from the ARCHIE component are control signals, and those between the storage (i.e., Disk File System and SSD file system) are the data movement.

the faster SSD storage. We used a simple predictor based on recent history, but any prediction algorithm can be used. A parallel reader/writer brings data from the disk file system into the SSD file system or vice versa. Because the data to be analyzed may be large, Data Elevator uses parallel reader/writer to prefetch data. Specifically, the reader/writer uses multiple MPI processes that span across all computing nodes that are running the analysis application. These processes concurrently prefetch different chunks from the disks into the SSD to increase the prefetch efficiency. When an application modifies a cached chunk, Data Elevator synchronizes updated cached chunks in the storage layers and writes the chunks back to the disk-based file systems.

Meanwhile, Data Elevator augments each prefetched chunk with a ghost zone layer (shown with a red halo around a blue chunk) to match the access pattern on the array. A user may specify the width of the ghost zone, which can be zero. The first two chunks (colored in white) read into the SSDs are actually empty chunks only containing metadata (e.g., starting and ending offsets). These metadata entries are written by the "read" function and they provide information for the prefetching algorithm in Data Elevator to predict future chunks. We have implemented parallel prefetching to accelerate parallel I/O of analysis applications.

We have also added fault tolerance features to Data Elevator, where either the application or the Data Elevator programs can fail gracefully. If an application running with the Data Elevator fails, and some data files are in the temporary staging area, the Data Elevator moves the files to the destination and then exits. In the case of Data Elevator failure while a co-locating application is running, the Data Elevator task restarts and resumes moving the data. A limitation of the Data Elevator restarting is the dependency on the SLURM scheduler that is popular in HPC center.

3.3 Asynchronous I/O

Asynchronous I/O allows an application to overlap I/O with other operations. When an application properly combines asynchronous I/O with non-blocking communication to overlap those operations with its calculation, it can fully utilize an entire HPC system, leaving few or no system components idle. Adding asynchronous I/O to an application's existing ability to perform non-blocking communication is a necessary aspect of maximizing the utilization of valuable exascale computing resources.

We have designed asynchronous operation support in HDF5 to extend to *all* aspects of interacting with HDF5 containers, not just the typical "bulk data" operations (i.e., raw data read/write). Asynchronous versions of "metadata" operations like file open, close, stat, etc. are not typically supported in I/O interfaces, but based on our previous experience, synchronization around these operations can be a significant barrier to application concurrency and performance. Supporting asynchronous file open and close operations as well as

other metadata operations such as object creation and attribute updates in the HDF5 container will allow an application to be decoupled entirely from I/O dependencies.

We have implemented the asynchronous I/O feature as a VOL connector for HDF5. The VOL interface in HDF5 allows developers to create a VOL connector that intercepts all HDF5 API calls that interact with a file. As a result, it requires no change to the current HDF5 API or the HDF5 library source code. We implemented asynchronous task execution by running those tasks in separate background threads. We are currently using Argobots [6], a lightweight runtime system that supports integrated computation and data movement with massive concurrency to spawn and execute asynchronous tasks in threads. The thread execution interface has been abstracted to allow us to replace Argobots with any other threading model, as desired.

3.4 Towards Full SWMR

The Single-writer and Multiple readers (SWMR) functionality allow multiple processes to read an HDF5 file while a single process is writing concurrently. A capability that enables a single writing process to update an HDF5 file, while multiple reading processes access the file in a concurrent, lock-free manner was introduced in the 1.10 release of HDF5. However, the SWMR feature is currently limited to the narrow use-case. The current implementation of "partial" SWMR (single-writer/multiple-reader) in the HDF5 library only allows appending to datasets with unlimited dimensions, but "full" SWMR functionality allows any modification to the file. Applications use a much broader set of operations on HDF5 files during workflow processing, making this limitation an impediment to their efficient operation.

We have designed and implemented to extend the SWMR feature in HDF5 to support all metadata operations for HDF5 files (e.g., object creation and deletion, attribute updates). In addition to removing the limitation in the types of operations that the SWMR feature supports, the limitation of only operating on files with serial applications must also be lifted, to fully support exascale application workflows. Therefore, we designed the extension of the SWMR capability to support parallel MPI applications for both the writing and reading aspects of the SWMR feature. This enables a single MPI application to function as the writing "process" and any number of MPI applications to be the reading "processes". The method of updating the HDF5 file will be identical for serial and parallel applications, so they can interoperate with each other, enabling serial readers to concurrently access a file being modified by a parallel writer, etc.

The implementation of full SWMR requires managing file space in the file, handle "meta" flush dependencies, and handle state for a closed object.

Managing File Space. A SWMR writer should not free the space when there is a possibility that readers continue accessing the space for a period of time and must not be overwritten with new information until all of the reader cached metadata entries that reference them are evicted or updated. To achieve this, the SWMR writer and readers have to agree when there are no references to agree on a "recycle time" ($\triangle t$) for space freed by the writer. Allowing for some margin of errors, freed space could be recycled on the writer after $2\triangle t$ and a reader would be required to refresh any metadata that remained in its metadata cache for longer than $\triangle t$.

Handling "Meta" Flush Dependencies. Objects that an SWMR writer is modifying in an HDF5 file that has metadata data structures with flush dependencies between those data structures must determine and maintain the correct set of "meta" flush dependencies between those data structures, over all possible operations to the object. The core capability used to implement updates to an HDF5 file by an SWMR writer is the "flush dependency" feature in the HDF5 library metadata cache. A flush dependency between a "parent" cache entry and its "child" entries indicates that all dirty child entries must be marked clean (either by being written to the file or through some other metadata cache mechanism) before a dirty parent entry can be flushed to the file. For example, when the dimension size of a dataset is extended, both the extensible array that references chunks of dataset elements and the dataspace message in the dataset object header must be updated with the new size, and the dataspace message must not be seen by an SWMR reader until the extensible array is available for the reader to access. For this situation, and others like it, proxy entries in the metadata cache are used to guarantee that all the dirty extensible array entries are flushed to the file before the object header entry with the dataspace message is written to the file.

In Fig.5, we show a typical situation in the metadata cache for a chunked dataset. All the entries for the chunk index (an extensible array, in this case) have

a "top" proxy entry that reflects the dirty state of all the entries in the index (i.e., if any index entry is dirty, the index's top proxy will be dirty also). Similarly, all the entries in the object header depend on a "bottom" proxy, which will keep any of them from being flushed (i.e., if the bottom proxy is dirty, no dirty object header entries can be flushed). Therefore, making the bottom proxy of the object header the flush dependency parent of the top proxy for the extensible array will force all extensible array entries to be written to the file and marked clean in the metadata cache before any dirty object header entries can be written to the file.

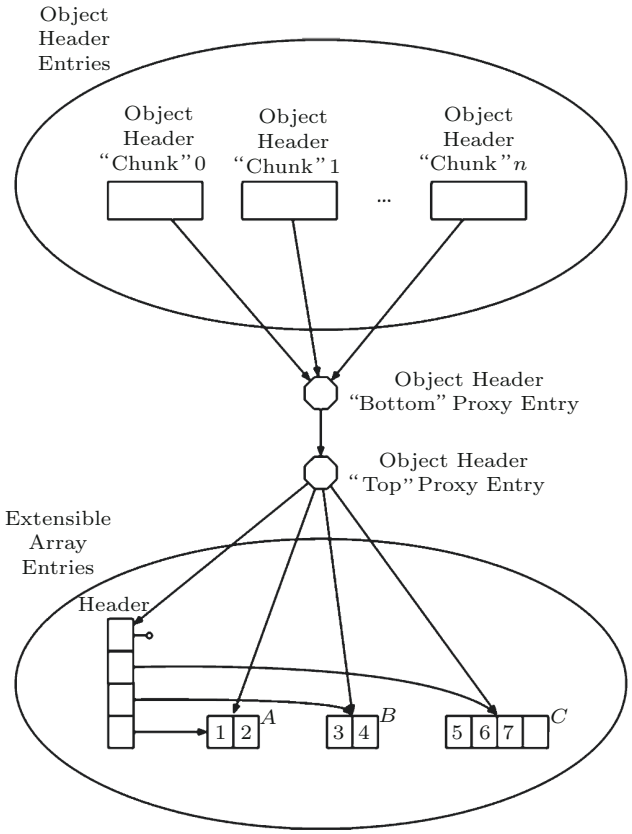

Fig.5. Typical situation of the HDF5 metadata cache for a chunked dataset.

Handle Metadata Cache State for a Closed Object. Objects that are opened by an SWMR writer must continue to maintain the necessary state required to update the object's metadata in the file while any of the metadata entries for the object are cached, even after the object is closed.

In future, we will be enhancing this feature with a mechanism for SWMR writer to notify the SWMR readers when a piece of metadata has changed, instead of current polling method where readers poll for such metadata updates. This will further improve the performance of SWMR readers.

3.5 Parallel Querying

Methods to store, move, and access data across complex Exascale architectures are essential to improve the productivity of scientists. To assist with accessing data efficiently, the HDF Group has been developing functionality for querying HDF5 raw data and releases this feature in HDF5. The interface relies on three main components that allow application developers to create complex and high-performance queries on both metadata and data elements within a file: queries, views, and indices.

Query objects are the foundation of the data analysis operations and can be built up from simple components in a programmatic way to create complex operations using Boolean operations. View objects allow the user to retrieve an organized set of query results.

The core query API is composed of two routines: `H5Qcreate` and `H5Qcombine`. `H5Qcreate` creates new queries, by specifying an aspect of an HDF5 container, in particular:

- `H5Q_TYPE_DATA_ELEM` (data element)

as well as a match operator, such as:

- `H5Q_MATCH_EQUAL` ($=$)
- `H5Q_MATCH_NOT_EQUAL` ($\neq$)
- `H5Q_MATCH_LESS_THAN` ($\leqslant$)
- `H5Q_MATCH_GREATER_THAN` ($\geqslant$)

and a value for the match operator. Created query objects can be serialized and deserialized using the `H5Qencode` and `H5Qdecode` routines[10], and their content can be retrieved using the corresponding accessor routines. `H5Qcombine` combines two query objects into a new query object, using Boolean operators such as:

- `H5Q_COMBINE_AND` ($\wedge$)
- `H5Q_COMBINE_OR` ($\vee$).

Queries created with `H5Qcombine` can be used as input to further calls to `H5Qcombine`, creating more complex queries. For example, a single call to `H5Qcreate` could create a query object that would match data elements in any dataset within the container that is equal to the value "17". Another call to `H5Qcreate` could create a query object that would match link names equal to "Pressure". Calling `H5Qcombine` with the $\wedge$ operator and those two query objects would create a new query object that matched elements equal to "17" in HDF5 datasets with link names equal to "Pressure". Creating

[10] Serialization/deserialization of queries was introduced so that queries can be sent through the network.

the data analysis extensions to HDF5 using a programmatic interface for defining queries avoids defining a text-based query language as a core component of the data analysis interface, and is more in keeping with the design and level of abstraction of the HDF5 API. Table 1 describes the result types for atomic queries and combining queries of different types.

Table 1. Query Combinations

Query	Result Type
`H5Q_TYPE_DATA_ELEM`	Dataset region
Dataset element ∧ dataset element	Dataset region
Dataset element ∨ dataset element	Dataset region

Index objects are the core means of accelerating the HDF5 query interface and enable queries to be performed efficiently. The indexing interface uses a plugin mechanism that enables flexible and transparent index selection and allows new indexing packages to be quickly incorporated into HDF5 as they appear, without waiting for a new HDF5 release.

The indexing API includes registration of an indexing plugin (`H5Xregister`), unregistering it (`H5Xunregister`), creating and removing indexes (`H5Xcreate` and `H5Xremove`, respectively), and gets information about the indexes, such as `H5Xget_info` and `H5Xget_size`. The current implementation of indexing in HDF5 supports FastBit bitmap indexing [7], which uses Word-Aligned Hybrid (WAH) compression for bitmaps generated. The bitmap indexes are generated offline using a utility, and they are stored within the same HDF5 file. These index files are used for accelerating the execution of queries. Without the indexes, query execution scans the entire datasets defined in a query condition.

4 Experimental Setup

We have conducted our evaluation on Cori, a Cray XC40 supercomputer at NERSC. Cori data partition (phase 1) consists of 1 630 compute nodes, and each node has 32 Intel® Haswell CPU cores and 128 GB memory. The Lustre file system of Cori has 248 object storage targets (OSTs) providing 30 PB of disk space. Cori is also equipped with an SSD based "Burst Buffer". The Burst Buffer is managed by DataWarp from Cray and has 144 DataWarp server nodes.

We describe the benchmarks and the results in the following section, as each feature used a different set of benchmarks and their evaluation metrics varied. For the VOL feature, evaluation of performance is unavailable as the feature as it is an implementation that enables other features, such as Data Elevator and asynchronous I/O. All the other features have been evaluated using various benchmarks and I/O kernels from real applications.

5 Evaluation

5.1 Data Elevator

We have implemented Data Elevator in C and compiled with Intel compilers. Our tests for disk-based performance used all 248 OSTs of the Lustre. In tests for the BB, we used all 144 DataWarp server nodes. The striping size for multiple DataWarp servers is fixed at 8 MB, which cannot be modified by normal users. As such, we set the striping size of Lustre file system also to be 8 MB.

We evaluated various configurations of Data Elevator using two parallel I/O benchmarks: VPIC-IO and Chombo-IO. Both benchmarks have a single time step and generate a single HDF5 file. VPIC-IO is a parallel I/O kernel of a plasma physics simulation code, called VPIC. In our tests, VPIC-IO writes 2 million particles and 8 properties per particle, resulting in a file of 64 GB in size. The I/O pattern of VPIC-IO is similar to particle simulation codes in the ECP, where each particle has a set of properties, and a large number of particles are studied. As shown in Fig.6, using Data Elevator for moving the data related to a single time step achieves 5.6x speedup on average over Cray DataWarp and 5x speedup on average over writing data directly to Lustre.

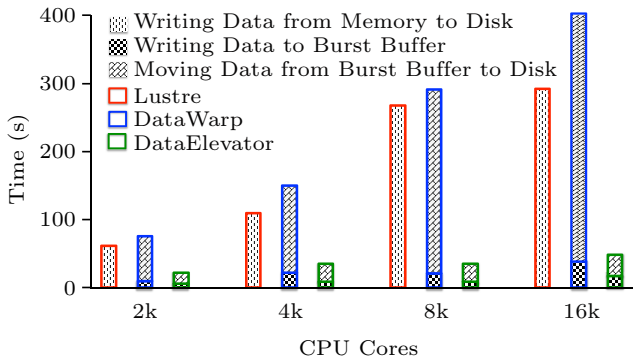

Fig.6. Evaluation of Data Elevator write caching functionality — I/O time of a plasma physics simulation's (VPIC-IO) data write pattern.

Chombo-IO is derived from Chombo, a popular block-structured adaptive mesh refinement (AMR) library. The generated file has a problem domain of

$256 \times 256 \times 256$ and is of 146 GB in size. The I/O pattern of this benchmark is representative of the subsurface flow simulation application in the ECP. In Fig.7, we show the performance benefit of the Data Elevator library. Chombo-IO benchmark achieves 2x benefit (on average) over Lustre and 5x benefit (on average) over Cray DataWarp.

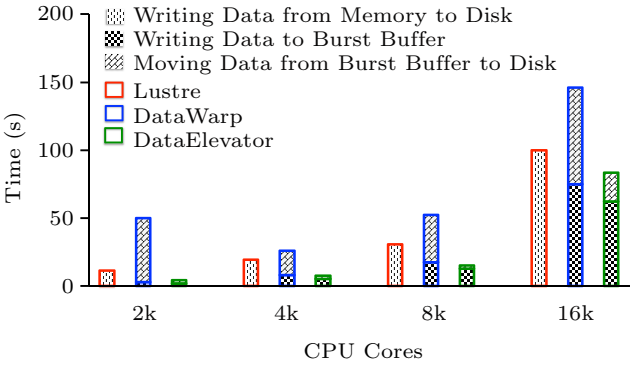

Fig.7. Evaluation of Data Elevator write caching functionality — I/O time of a Chombo adaptive mesh refinement (AMR) code's I/O kernel.

In both benchmarks, the benefit over Lustre is coming from writing data to faster SSD-based burst buffer compared with the disk-based Lustre. With regards to Cray DataWarp, the advantage is coming from two factors: the selection of MPI-IO mode for writing data to the SSD-based burst buffer and dynamic configuration of Lustre striping parameters based on the data size. The Data Elevator changes the MPI-IO mode from collective (two-phase) I/O to independent I/O because we observed that the BB on Cori is performing much better with the independent mode. For the selection of Lustre striping, while we chose all the Lustre OSTs for both Cray DataWarp and Data Elevator, the stripe size is dynamically set based on the file size in Data Elevator.

We evaluate DE read caching by comparing it with Lustre to show the benefits of the prefetching function on accelerating I/O operations by converting non-contiguous I/O to contiguous ones. Cray's DataWarp provides tools, e.g., "stage-in", to manually move a data file from disk space into SSD space. In contrast, DE automatically prefetches the array data from Lustre to the burst buffer in chunks. Our method avoids application's wait time for the entire file data movement at the start of job execution enforced by DataWarp.

We compare DE reading (labeled ARCHIE in the plots representing Array Caching in hierarchical storage) using three scientific analysis kernels: convolutional neural network (CNN)-based analysis on CAM5 data used to detect extreme weather events [8], gradient computation of plasma physics dataset using a 3D magnetic field data generated by a VPIC simulation [9], and vorticity computation on combustion data produced by an S3D simulation that captures key turbulence-chemistry interactions in a combustion engine [10].

In our evaluation of DE reading with CNN of a climate dataset, we focus on the most time-consuming step, i.e., convolution computing, in CNNs. The CAM5 dataset used in this test is a 3D array with size $[31, 768, 1\,152]$, where 768 and 1 182 are the latitude and the longitude dimensions, respectively, and 31 is the number of height levels from the earth into the atmosphere. The filter size for the convolution is $[4, 4]$. The chunk size is $[31, 768, 32]$, resulting in a total of 36 chunks. The analysis application runs with 9 MPI processes. We run DE with another 9 processes. In this configuration, there are 4 batches of reads, and each batch accesses 9 chunks. We present the read time of the analysis application in Fig.8. Reading all the data from the disks, i.e., Lustre, gives the worst performance, as expected. Using DataWarp to move the data from Lustre to the burst buffer reduces the time by 1.7x for reading data, but it contains initial overhead in staging the data in the burst buffer. Using the DE cache on both the disks and on the SSDs reduces the time for reading data. The advantage of DE comes from converting non-contiguous reads into contiguous reads. Data Elevator can also prefetch data to be accessed in the future into the burst buffer as chunks and achieves the best performance. Overall, for the CNN use case, DE is 3.1x faster than Lustre and 1.8x faster than DataWarp.

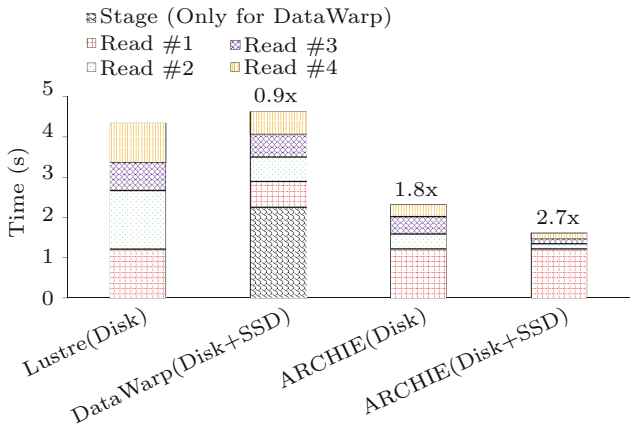

Fig. 8. Time profile of Data Elevator read caching for CNN analysis of CAM5 data. The plot compares read performance of Lustre and DataWarp with Data Elevator caching using the Lustre file system and using the SSD-based DataWarp (labeled ARCHIE).

The VPIC dataset is a 3D array with size as $[2\,000, 2\,000, 800]$. On a meshed 3D field, the gradient can be computed with a Laplacian, as shown in (1).

$$\text{grad} f(x, y, z) = \nabla f(x, y, z), \tag{1}$$

where f represents magnetic value and ∇ denotes the Laplace operator.

The chunk size for parallel array processing is $[250, 250, 100]$, giving a total of 512 chunks. Our tests use 128 processes to run the analysis program and use only 64 processes to run DE. In this test, we have a ghost zone with a size of $[1, 1, 1]$. We present the read time of this analysis in Fig.9. The performance comparison has the same pattern as that of convolution on CAM5 data. Overall, DE is 2.7x faster than Lustre and 2.4x faster than DataWarp.

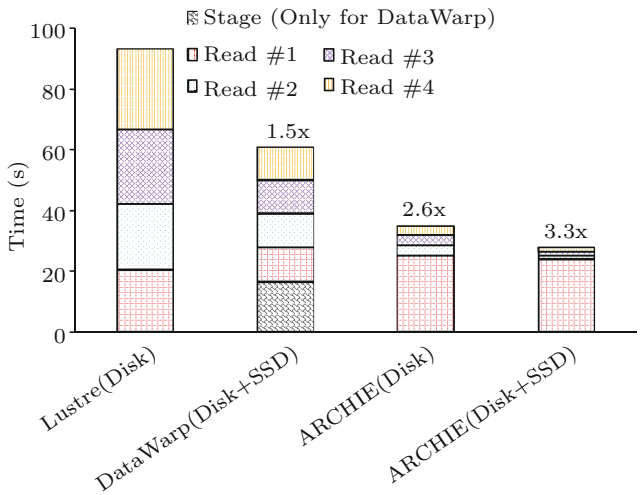

Fig.9. Data read time profiles for gradient computation of VPIC data. ARCHIE represents the Data Elevator's array caching optimization.

S3D simulation code captures key turbulence-chemistry interactions in a combustion engine. An attribute to study the turbulent motion is vorticity, which defines the local spinning motion for a specific location. Given the z component of the vorticity at a point (x, y), the vorticity analysis accesses four neighbors for each point at $(x+1, y)$, $(x-1, y)$, $(x, y-1)$ and $(x, y+1)$, as defined in [11]. Our tests use a 3D array with size $[1\,100, 1\,080, 1\,408]$. The chunk size is $[110, 108, 176]$ and the ghost zone size is $[1, 1, 1]$, giving 800 chunks. We use 100 processes to run analysis programs and 50 processes to run DE. The read performance comparison is shown in Fig.10. The analysis program reads all 800 chunks in 8 batches. DE outperforms Lustre by 5.8x and is 7x better than DataWarp. In this case, DataWarp performs 1.2x slower than Lustre.

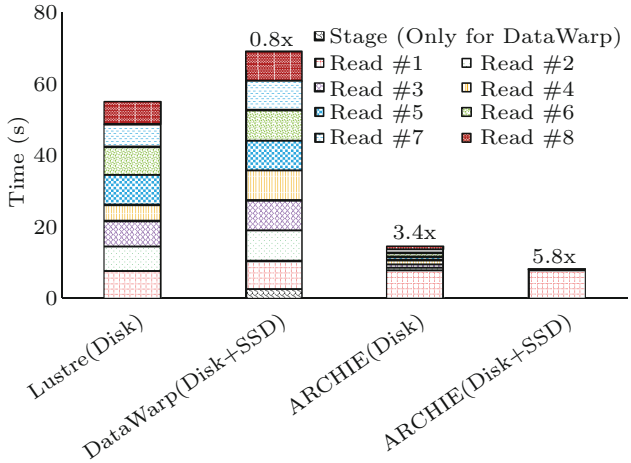

Fig.10. Data read time profiles for vorticity computation of a combustion simulation (S3D) data. ARCHIE represents the Data Elevator's array caching optimization.

5.2 Asynchronous I/O

We ran experiments with asynchronous I/O enabled in HDF5 on the Cori system. The tests used an I/O kernel from a plasma physics simulation (VPIC) that writes 8 variables per particle at 5 timesteps. The number of MPI processes is varied from 2 to 4k cores, in multiples of 2. During the computation time between subsequent time steps, which is emulated with a 20-second sleep time, Argobot tasks started on threads perform I/O by overlapping I/O time fully with the computation time.

In Fig.11 and Fig.12, we show a comparison of write time and read time, respectively, with and without the asynchronous I/O. In Fig.11, we show the I/O time for writing data produced by a simulation kernel and for reading data to be analyzed, where using asynchronous I/O obtains 4x improvement. In these cases, I/O related to 4 of the 5 timesteps is overlapped with computation. The last timestep's I/O has to be completed before the program exits, which is not overlapped. The overall performance benefit is hence dependent on the number of time steps used.

5.3 Full SWMR

To evaluate the performance of full SWMR, we have implemented a version of SWMR-like functionality with locks using MPI (labeled non-SWMR in Fig.13 and Fig.14). We compare the performance of this non-SWMR with the full SWMR implementation in HDF5 for a single writer to write 1D data to an HDF5 dataset, where the write size is varied between 1 KB and 512 MB for 100 times. While the single writer process is writ-

ing data, three readers open the dataset and read the newly written data concurrently.

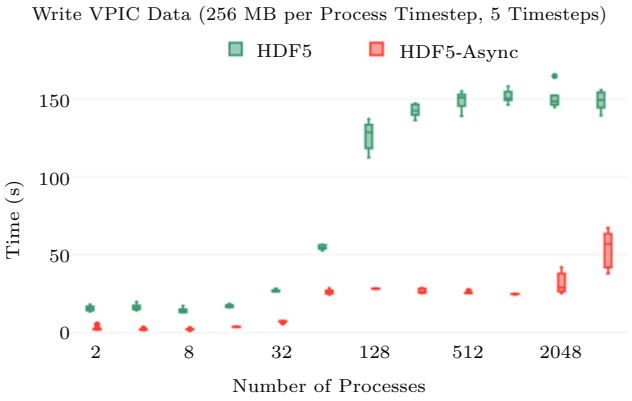

Fig.11. Parallel write performance of HDF5 with asynchronous I/O enabled (HDF5-async) compared with the current HDF5 implementation.

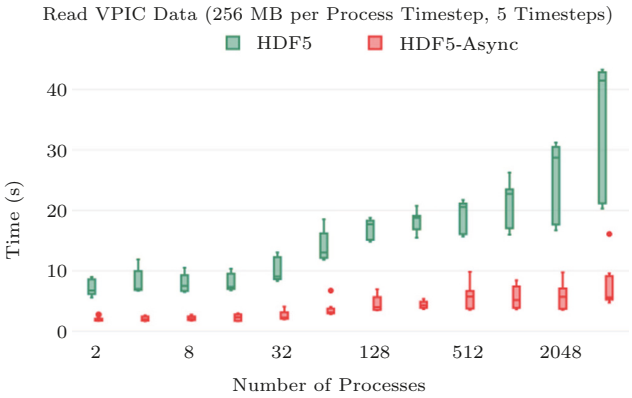

Fig.12. Parallel read performance of HDF5 with asynchronous I/O enabled (HDF5-async) compared with the current HDF5 implementation.

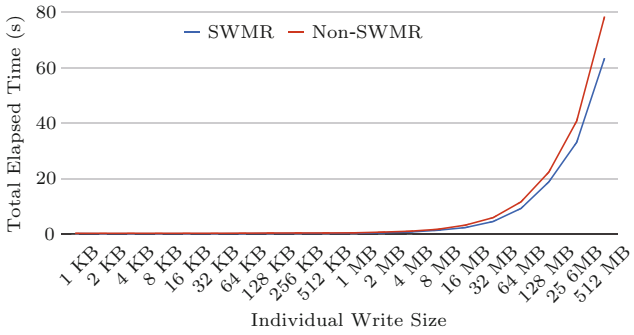

Fig. 13. Performance comparison — SWMR vs non-SWMR. 1 writer writes 1D data to a HDF5 dataset, with individual write size ranging from 1 KB to 512 MB for 100 times. At the same time, 3 readers discover and read the newly written data concurrently.

In Fig.13, we show the raw comparison, and in Fig.14, we show the speedup numbers for performing this SWMR functionality. As can be seen, our improved SWMR implementation HDF5 outperforms the non-SWMR approach by up to 8x at smaller write sizes between 2 KB and 32 KB. Beyond 32 KB, our implementation still performs well, by 20% at 512 MB writes.

5.4 Parallel Querying

Initial benchmarking of the parallel querying implementation prototype on Cori resulted in the following scalability plots (Fig.15 and Fig.16) which show the time taken to 1) construct the FastBit index; and 2) evaluate a query of the form: finding all elements in the "Energy" dataset whose value is greater than 1.9 (i.e., "Energy > 1.9"). The data used for this query

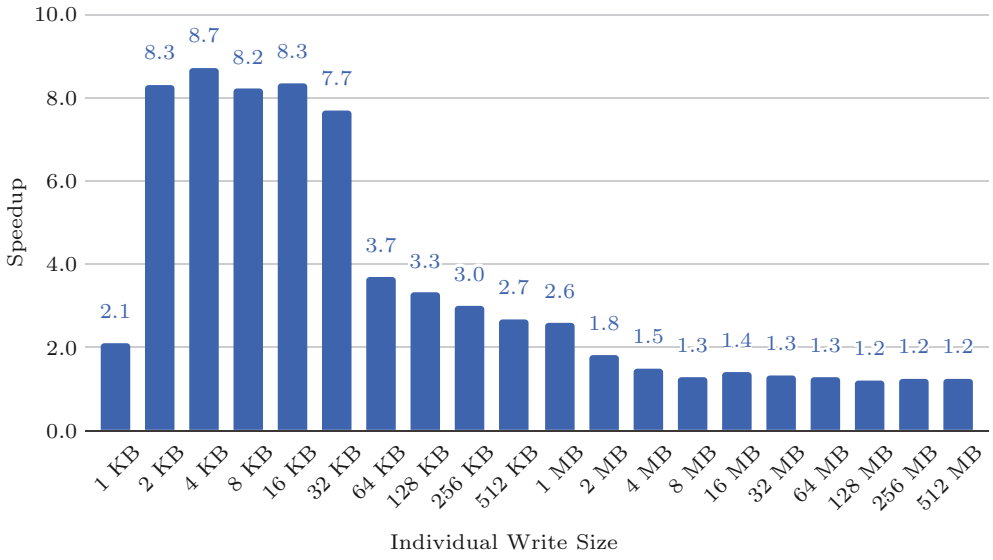

Fig.14. Speedup of SWMR over non-SWMR in writing dataset with different write sizes.

was obtained from a plasma physics simulation (VPIC) ran for understanding the magnetic field interactions in a magnetic reconnection phenomenon of space weather scenario. The HDF5 file contained particle properties, such as the spatial locations in 3D, corresponding velocities, and energy of particles. Each property was stored as an HDF5 dataset that contains 623 520 550 elements represented as 32-bit floating point values.

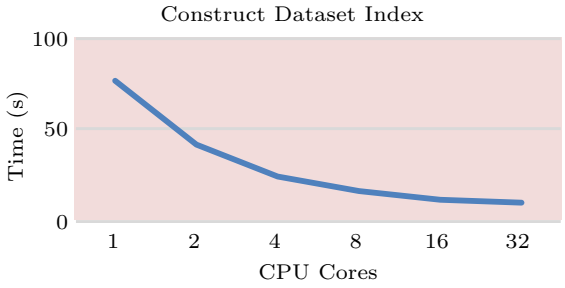

Fig. 15. Parallel index generation time, where the indexes are based bitmap indexes generated by the FastBit indexing library.

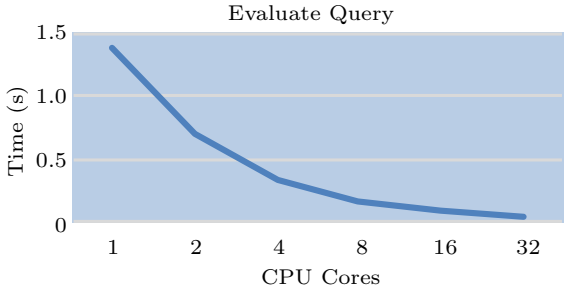

Fig.16. Parallel query evaluation time.

As shown in Fig.15 and Fig.16, both index generation and query evaluation scale well for this dataset up to 8 cores. The advantage of adding more cores diminishes because the amount of data processed by each MPI process becomes smaller. With larger datasets, both these functions will scale further.

6 Conclusions

HDF5 is a critical parallel I/O library that is used heavily by a large number of applications on supercomputing systems. Due to the increasing data sizes, extreme level of concurrency, and deepening storage hierarchy of upcoming exascale systems, the HDF5 library has to be enhanced to handle several challenges. In the ExaHDF5 project funded by the Exascale Computing Project (ECP), we are developing various features to improve the performance and productivity of HDF5. In this paper, we presented a few of those feature enhancements, including integration of Virtual Object Layer (VOL) that opens up the HDF5 API for alternate ways of storing data, Data Elevator for storing and retrieving data by using faster storage devices, asynchronous I/O for overlapping I/O time with application computation time, full SWMR for allowing concurrent reading of data while a process is writing an HDF5 file, and querying to access data that matches a given condition. We have presented an evaluation of these features, which demonstrates significant performance benefits (up to 6x with Data Elevator that uses SSD-based burst buffer compared with using disk-based Lustre system, and 1.2x to 8x benefit with SWMR compared with non-SWMR implementation). We have compared these features separately as the use cases are distinct. For instance, the Data Elevator and the async I/O features use the VOL infrastructure. Full SWMR and querying serve different use cases from masking the I/O latency. With the ongoing ExaHDF5 effort, these features will be integrated into HDF5 public release to have a broad performance advantage impact for a large number of HDF5 applications.

References

[1] Folk M, Heber G, Koziol Q, Pourmal E, Robinson D. An overview of the HDF5 technology suite and its applications. In *Proc. the 2011 EDBT/ICDT Workshop on Array Databases*, March 2011, pp.36-47.

[2] Li J W, Liao W K, Choudhary A N *et al.* Parallel netCDF: A high-performance scientific I/O interface. In *Proc. the 2003 ACM/IEEE Conference on Supercomputing*, November 2003, Article No. 39.

[3] Lofstead J, Zheng F, Klasky S, Schwan K. Adaptable, metadata rich IO methods for portable high performance IO. In *Proc. the 23rd IEEE International Symposium on Parallel Distributed Processing*, May 2009, Article No. 44.

[4] Dong B, Byna S, Wu K S *et al.* Data elevator: Low-contention data movement in hierarchical storage system. In *Proc. the 23rd IEEE International Conference on High Performance Computing*, December 2016, pp.152-161.

[5] Dong B, Wang T, Tang H, Koziol Q, Wu K, Byna S. ARCHIE: Data analysis acceleration with array caching in hierarchical storage. In *Proc. the 2018 IEEE International Conference on Big Data*, December 2018, pp.211-220.

[6] 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.

[7] Wu K. FastBit: An efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 2005, 16(16): 556-560.

[8] Racah E, Beckham C, Maharaj T, Kahou S E, Prabhat, Pal C. ExtremeWeather: A large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events. In *Proc. the 31st Annual Conference on Neural Information Processing Systems*, December 2017, pp.3402-3413.

[9] Byna S, Chou J C Y, Rübel O *et al*. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *Proc. the International Conference on High Performance Computing, Networking, Storage and Analysis*, November 2012, Article No. 59.

[10] Chen J H, Choudhary A, de Supinski B *et al*. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2009, 2(1).

[11] Dong B, Wu K S, Byna S, Liu J L, Zhao W J, Rusu F. ArrayUDF: User-defined scientific data analysis on arrays. In *Proc. the 26th International Symposium on High-Performance Parallel and Distributed Computing*, June 2017, pp.53-64.