HDF5 Cache VOL: Efficient and Scalable Parallel I/O through Caching Data on Node-local Storage

Huihuo Zheng
Argonne National Laboratory
huihuo.zheng@anl.gov

Venkatram Vishwanath
Argonne National Laboratory
venkat@anl.gov

Quincey Koziol
Amazon.com, Inc
qkoziol@amazon.com

Houjun Tang
Lawrence Berkeley Laboratory
htang4@lbl.gov

John Ravi
North Carolina State University
jjravi@ncsu.edu

John Mainzer
The HDF Group
mainzer@hdfgroup.org

Suren Byna
Lawrence Berkeley Laboratory
sbyna@lbl.gov

***Abstract*—Modern-era high performance computing (HPC) systems are providing multiple levels of memory and storage layers to bridge the performance gap between fast memory and slow disk-based storage system managed by Lustre or GPFS. Several of the recent HPC systems are equipped with SSD and NVMe-based storage that is attached locally to compute nodes. A few systems are providing an SSD-based "burst buffer" intermediate storage layer that is accessible by all compute nodes as a single file system. Although these hardware layers are intended to reduce the latency gap between memory and disk-based long-term storage, how to utilize them has been left to the users. High-level I/O libraries, such as HDF5 and netCDF, can potentially take advantage of the node-local storage as a cache for reducing I/O latency from capacity storage. However, it is challenging to use node-local storage in parallel I/O especially for a single shared file.**

In this paper, we present an approach to integrate node-local storage as transparent caching or staging layers in a high-level parallel I/O library without placing the burden of managing these layers on users. We designed this to move data asynchronously between the caching storage layer and a parallel file system to overlap the data movement overhead in performing I/O with compute phases. We implement this approach as an external HDF5 Virtual Object Layer (VOL) connector, named *Cache VOL*. HDF5 VOL is a layer of abstraction in HDF5 that allows intercepting the public HDF5 application programming interface (API) and performing various optimizations to data movement after the interception. Existing HDF5 applications can use *Cache VOL* with minimal code modifications. We evaluated the performance of *Cache VOL* in HPC applications such as VPIC-IO, and deep learning applications such as ImageNet and CosmoFlow. We show that using *Cache VOL*, one can achieve higher observed I/O performance, more scalable and stable I/O compared to direct I/O to the parallel file system, thus achieving faster time-to-solution in scientific simulations. While the caching approach is implemented in HDF5, the methods are applicable in other high-level I/O libraries.

Parallel IO, storage hierarchy, node-local storage, HDF5, Caching, Prefetching, deep learning

I. Introduction

Large scale scientific simulations usually generate and analyze massive amounts of data. A critical requirement of these applications is the capability to move and manage the data efficiently. To bridge the performance gap between memory and traditional global disk-based file systems, many pre-exascale supercomputers have been adding a complex storage hierarchy including node-local storage (NLS) with SSDs, NVMe, and burst buffers. It has increasingly become common to include NLS in modern supercomputer design. We show in Fig. 1 a high-level architecture of compute nodes with NLS and global (remote) storage using parallel file systems (PFS), such as Lustre and IBM's Spectrum Scale (previously known as GPFS). In Table I, we show a list of recent supercomputers and their storage options for both NLS and PFS.

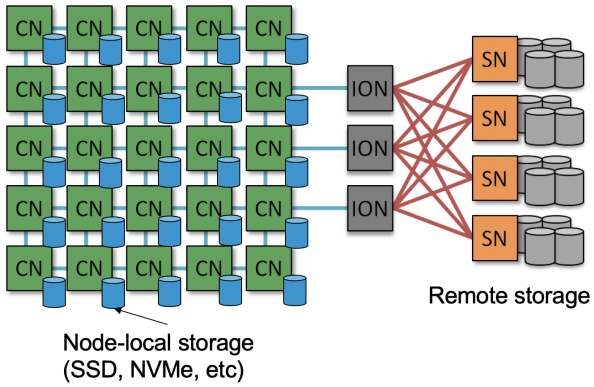


Fig. 1. Typical HPC storage hierarchy: node-local storage for short-term and fast storage of data and a global parallel file system for long-term storage.

TABLE I
STORAGE HIERARCHY OF VARIOUS PRE-EXASCALE SYSTEMS

System	file System	Node-local storage
Theta/ThetaGPU [1]	Lustre	128GB SSD / 15TB NVMe
Sierra [2]	GPFS	1.6TB NVMe
Summit [3]	GPFS	1.6TB NVMe
Fugaku [4]	Lustre	1.6 TB NVMe (per 16 nodes)

The global PFS storage layer managed is separated from the compute nodes. I/O access thus involves transferring data through an interconnect. Whereas the NLS devices are directly attached to compute nodes through SATA or PCIe connections, offering higher bandwidth. Each host compute node can assess its own NLS device locally without going through the inter-

connect. Scientific applications and I/O libraries development has traditionally focused on data movement directly between the compute nodes memory and the global storage layer, and has not made full usage of the complex storage hierarchy. It is thus crucial to study how to integrate other storage layers such as NLS into the data movement workflow to improve overall I/O performance at scale.

Compared to a global PFS, NLS has the following properties: first, because accessing data on the NLS does not need to go through an external interconnect, it avoids network contention from other jobs running concurrently on the system, resulting a much more stable I/O performance with better scalability. Second, the aggregate I/O bandwidth of all the NLS in the entire system is typically higher than the peak bandwidth of the PFS. For example, Theta has an aggregate write bandwidth of about 3 TB/s for SSDs, which is 5 times of the Lustre peak bandwidth of about 650 GB/s [1]; Summit has an aggregate write bandwidth of 9.7 TB/s for NVMe, which is about 4 times of the GPFS peak bandwidth (2.5 TB/s) [3]. While the global PFS is shared by all concurrently running jobs, NLS is often exclusively used by the job running on the compute nodes. This allows using all available bandwidth of NLS without interference. However, since NLS devices are not directly connected with each other, it is challenging to incorporate them into parallel I/O workflow. Besides, NLS are usually accessible only during job allocation; therefore, one has to manually move data to the PFS before the job finishes. Due to these complexity of hardware in storage layers and the lack of existing software solutions, to the best of our knowledge, NLS is still used mostly as a local scratch space for storing data temporarily and serves as a slow memory extension to DRAM on the compute nodes.

To take full advantage of the NLS layer, there is a necessity for simple software solutions that are transparent in using NLS for caching data both in the write and read directions and for moving the data between the NLS and the PFS without user involvement. To achieve this goal, we present in this paper an approach to integrate NLS as an intermediate storage layer for caching data temporarily in parallel I/O workflows. Specifically, for write, we stage the data to the NLS, and then migrate them to the PFS asynchronously using background threads. This allows the data migration from NLS to PFS to overlap with the computation, resulting in reducing a large portion of I/O overhead. For parallel read, we cache the data to the NLS when first time they are read from the PFS and read them directly from the NLS for future requests.

One challenge in using NLS is how to unify all the storage devices into a single piece of storage layer to allow efficient remote access of other compute nodes' storage. This is crucial especially for the read direction. Many approaches were proposed to address this challenge. BurstFS [5] and a follow-up library called UnifyFS [6] are user-level file systems proposed for unifying NLS, both of which require a separate file system installation. In UniviStor [7], the authors proposed an MPI-IO abstraction layer (ADIO) [8] for handling NLS as a single layer. However, both UnifyFS and UniviStor intercept the POSIX IO or MPI-IO interfaces and require users to launch servers as another job running concurrently on separate CPU cores to instantiate the user-level file systems. Many supercomputers unfortunately do not support such a framework. In this paper, we proposed a shared memory-mapped approach. We create files on each NLS device and map them into the processes virtual memory using mmap; we then expose that local memory-mapped storage space to all the compute nodes through an MPI Window, which allow us to perform I/O operations on the shared storage space through one-sided remote memory access (RMA). There is no need to run a separate file system server in this approach.

We implement the whole framework in the HDF5 [9] library as an external passthrough Virtual Object Layer (VOL) connector named *Cache VOL* [10]. The VOL is an abstraction layer within the HDF5 library that intercepts object-level API operations on HDF5 files (such as "file open", "dataset write", "group create", etc) and forwards those operations to plugins, called "VOL connectors" [11]. These connectors are dynamically loadable at runtime using environment variables and enable third-party developers to build customized storage solutions for HDF5 users without having to change application code. Using the VOL framework, we hide all the complexity of intermediate data movement inside the library, such as data caching and staging, data migration between the NLS and the PFS. Specifically, data staging is performed within *Cache VOL* itself, whereas the asynchronous data migration is performed by stacking *Cache VOL* with another VOL connector, *Async VOL* [12], [13].

Cache VOL will benefit applications with heavy check-pointing I/O, as well as applications that involves repetitive intensive read, such as machine learning and deep learning applications. We demonstrated the benefits of using *Cache VOL* in three applications, VPIC-IO [14] for check-pointing I/O, AlexNet and CosmoFlow [15] for deep learning training. With *Cache VOL*, we achieve higher and more scalable observed I/O rate and reduce the overall time-to-solution.

Major contributions of this work are summarized as follows:

- We propose a framework to combine caching with asynchronous data migration to hide I/O overhead behind the computation and achieve efficient parallel I/O at scale.
- We propose a memory-mapped share file system to unify all the NLS devices into a single piece of storage layer, allowing easy and efficient access of remote nodes' NLS.
- We implement the framework in an HDF5 VOL connector, deploying which requires no change or very minimal changes of the code.
- We demonstrate the effectiveness of caching and asynchronous data migration framework in traditional HPC simulation applications and deep learning applications.

The remainder of the paper is organized as follows. We first discuss the background on the HDF5 library and VOL in Section II. We then outline the design and implementation of *Cache VOL* in Section III, and present performance evaluations on leadership-class supercomputers such as Theta/ThetaGPU

and Summit, in Section IV. Finally, we discuss related works in Section VI and conclude the paper in Section VII.

II. Background

A. HDF5 and Virtual Object Layer (VOL)

HDF5 is a popular high-level I/O library that provides an API to store and retrieve data [9], [16]. HDF5 uses principles of a self-describing file format, where metadata that describes the data is also stored with the data. The data objects (called Datasets) are "decorated" with metadata objects (called Attributes). The objects are stored in Groups in a hierarchy that is similar to those on Unix file systems, with a "Root" directory as the top directory. HDF5 has been used heavily by scientific applications at various supercomputing facilities [17]. HDF5 provides parallel I/O using MPI-IO [18], [8]. Using the optimizations provided by MPI-IO, HDF5 provides a portable file format for parallel applications on HPC systems.

While HDF5 is heavily used, storage hardware architectures as well as software systems such as object data management systems (i.e., Amazon S3) have been transforming scientific data management. To allow HDF5 users to take advantage these advancements, HDF5 library recently added a new feature, called Virtual Object Layer (VOL). The VOL infrastructure allows external library developers to intercept the HDF5 API and redirect I/O calls to use alternate storage methods. For instance, HDF5 VOL "connectors" are available to access a new HPC-oriented object data management system called DAOS [19]. Another connector is available to perform I/O asynchronously, which allows overlapping I/O with computation phases using background threads [13].

HDF5 VOL connectors can be categorized as to types: terminal and passthrough. The terminal VOL connectors intercept the API and then store data on hardware, typically in a different file format than that of HDF5. For instance, the DAOS VOL connector is terminal, that uses DAOS object format to store HDF5 objects. On the other hand, the asynchronous I/O VOL connector is a passthrough VOL that uses the HDF5 "native" VOL connector as the terminal VOL for storing the data in the HDF5 file format. Similar to the asynchronous I/O VOL connector can use the native VOL connector, multiple VOL connectors can be stacked one top of another. In this paper, we introduce *Cache VOL* stacking on top of the asynchronous I/O VOL connector, which then uses the native VOL connector maintaining the HDF5 file format to store the data on PFS.

III. Design and implementation of *Cache VOL*

We have a set of design goals for *Cache VOL*:

- The application shall be able to call the same I/O functions in a similar way as if it was directly dealing with memory and global storage. All the complexity such as data caching and data movement between the NLS and PFS should be done inside the I/O library and be hidden from the users.
- The need for additional hardware resource inside the library should be kept minimal and controllable if possible. For example, one should avoid hidden memory allocation and memory copy as much as possible. We also would like to avoid running any kinds of background server as that may need dedicated computing resource. The application should be able to run with similar configurations as before.
- The application should be able to adopt the framework with minimal code change, even without recompiling the code. This will significantly reduce the amount of efforts in software development and maintenance, particular for those applications with multiple layers of I/O software stack. For example, E3SM [20] uses netCDF [21] which then calls HDF5 as a lower level backend; Python workloads typically use h5py [22] which is a Pythonic interface for HDF5.
- The framework should be portable to all architecture platforms. Different systems may have different hardware configurations. The framework shall depend only on universal properties, such as namespace and capacity of NLS. This information should be easily obtainable.
- The framework shall, to large extent, also be detached from the lower level I/O library. In the context of HDF5, we choose to implement the framework as an external VOL plugin rather than integrating it into the HDF5 library. From the developer perspective, this makes library maintenance easier; from the user perspective, the framework is less dependent on the version of the HDF5 library pre-installed in the system.

In the following, we will discuss the details of our design and implementation. Even though our actual implementation is done in the context of HDF5, the framework can be implemented in other generic I/O libraries as well; except that in HDF5, the VOL framework provides us an easy way to achieve all the design goals above.

A. Parallel write

In the write direction, data is transferred from the compute node memory to the PFS storage. We use the NLS as a cache to the PFS. Specifically, for each write request, data is written to the NLS and then flushed to the PFS storage. The flushing process (data migration) shall be done asynchronously in the background to allow hiding majority of the I/O overhead behind the computation. If there is enough computation to overlap with the data migration, the whole process shall appear as if the application is written data to a low latency PFS. The whole process is schematically shown in Fig. 2, where the red solid arrows denote writing data to the NLS cache; and the blue dash arrows denote the background data migration.

Regarding the background data migration, there are two potential options: (1) using an asynchronous write function from the I/O library, such as MPI_File_iwrite in MPI-IO; (2) using background threads to perform I/O while the main threads are doing computation. One can have a task pool. For each write request, the application writes the data to the NLS and adds a data migration task to the pool; the library then executes the data migration tasks from the pool one by one in

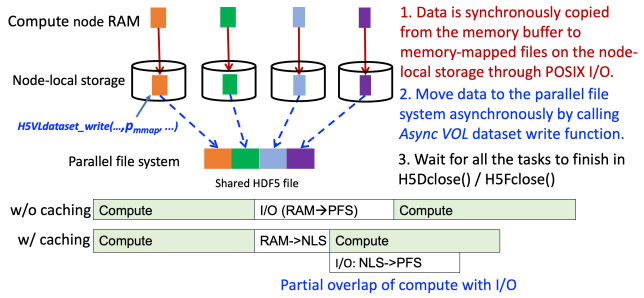


Fig. 2. Dataset write in *Cache VOL*. The data in the write buffers are copied to the node-local storage first, and then migrated to the parallel file system asynchronously by background threads in *Async VOL*. Data migration overlaps with the compute work right after data has been staged to the node-local storage, resulting a higher observed write rate.

the background. The write function call returns right after the data has been written to the NLS cache without waiting for the data migration to finish. In this sense, the write function call appears as a semi-blocking call. The write buffers are immediately reusable for other purposes after the function returns.

The above framework can be implemented in any I/O libraries through intercepting the write function. In the case of HDF5, we implement the whole process in *Cache VOL* as follows:

Data staging: Each process creates a file on the NLS for staging data. When a dataset write function (*H5Dwrite*) is called, each process writes the data to the file on the NLS through POSIX I/O and adds a data migration task to the task pool. The task contains relevant information such as the file space ID, memory space ID as well as the offset of the staging file at which data was written. The task pool is managed in a first-in-first-out fashion.

Data migration: HDF5 *Async VOL* provides an asynchronous dataset write function implemented through background threads. We directly use the asynchronous dataset write function to perform the data migration (see Fig. 2). The VOL framework allows us easily to adopt *Async VOL* into the workflow simply through stacking it under *Cache VOL*.

In order to avoid uncontrollable increase of the memory footprint, we use memory map to avoid explicit extra memory allocation. We use mmap to map the files on the NLS into the virtual memory of the system, and then use a pointer to address the data on the NLS (p_{mmap} in Fig. 2), without reading the data into a memory buffer. With mmap, memory mapped files are loaded into memory one entire page at a time. The system will dynamically evict data in previous pages if the memory is not enough.

To guarantee all the data are flushed to the PFS before the application ends, we wait for all the asynchronous tasks to finish in the close functions such as dataset close, group close, and file close.

Finally, we want to point out that with the background threads approach, the data migration might potentially compete with other parts of the simulation for computing resource. However, given the fact that modern supercomputers are mostly heterogeneous, if majority of the compute works can be offloaded into accelerators, dedicating one thread for each process on the host for I/O shall not be an issue. Data migration involves MPI I/O which uses the interconnect resource. This might potentially interfere with any concurrent MPI communication in the simulation. Understanding the potential performance impact because of resource contention in asynchronous I/O is one of our future research topic.

B. Parallel read

In the read direction, data is transferred from the PFS to the compute node memory. To incorporate the NLS into the workflow, one can prefetch the data from the PFS and cache it to the NLS, so that the application can read data directly from the NLS when actual read happens. However, this generic case requires us to be able to efficiently predict what data the future requests will read, which is a very challenging research topic. While we intend to support this generic case in *Cache VOL*, in this paper, we limit our study to a simpler repetitive read scenario, in which the application reads the same dataset multiple times, and we focus on improving the I/O performance for the read in second and onward iterations.

Repetitive read is a typical I/O pattern in machine learning applications which involve reading the same dataset tens of or hundreds of or thousands of times. In the first iteration, as the data is being read from the parallel file system, we cache a copy to the NLS; then at later iterations, the application will directly read data from the NLS. The read function call will check whether the requested data has already been cached or not. If so, it will read directly from the NLS; otherwise, it will go to the PFS to get the data and then cache a copy to the NLS. The whole process is schematically shown in Fig. 3.

The caching process can be done either synchronously by the main threads, or asynchronous through background threads. The latter might have less overhead for the first iteration compared to the former, as it allows hiding the caching behind the computation.

Compared to parallel write, the implementation for parallel read requires more complex management of the NLS since it generally involves accessing data from the remote node's storage. For example, in the case of machine learning, during the training process, if data shuffling is used, each worker will process different subsets of dataset at different epochs. An epoch is defined as an iteration at which the application reads through the entire training dataset once. Therefore, each worker might need to read data which was previously cached to a remote node's storage. We have to address the following two questions: (1) how to distribute the entire dataset among all the NLS on different compute nodes; (2) how to efficiently store (and load) the data to (and from) the NLS.

Regarding distributing the dataset, we simply divide the dataset into equal partitions among all the MPI processes. Each process will manage caching for one of the partitions.

Regarding the store and load, one can potentially use MPI point-to-point communications, such as MPI_Send and

MPI_Recv to transfer data among the process. For example, as shown in Fig. 3(a), if Process A reads in data (highlighted as green in Process A's RAM) that is supposed to be cached to Process B's NLS storage space, Process A can send the data to Process B. Process B then writes data to its own NLS. Likewise, if Process A needs data (highlighted as green in Process A's RAM)from Process B's NLS storage space as shown in Fig 3(b), Process B can read the data from its own NLS storage space and send the data to Process A. However, this might be very inefficient, since each read request needs to involve two processes.

We propose a more efficient approach which involves mmap and one-sided MPI communication such as MPI_Put for store and MPI_Get for load. We call the whole system, a memory-mapped shared file system. First, memory-mapped files are created on each NLS drive. The size of each file is equal to the size of the dataset partition to be cached. We then associate the mmap buffer to an MPI Window, through which each process exposes part of the NLS space to be shared to all the other processes. All the processes can then access the shared NLS storage space through RMA calls such as MPI_Put and MPI_Get. The framework proposed here is similar to other approaches proposed in the literature [23], [24], [25].

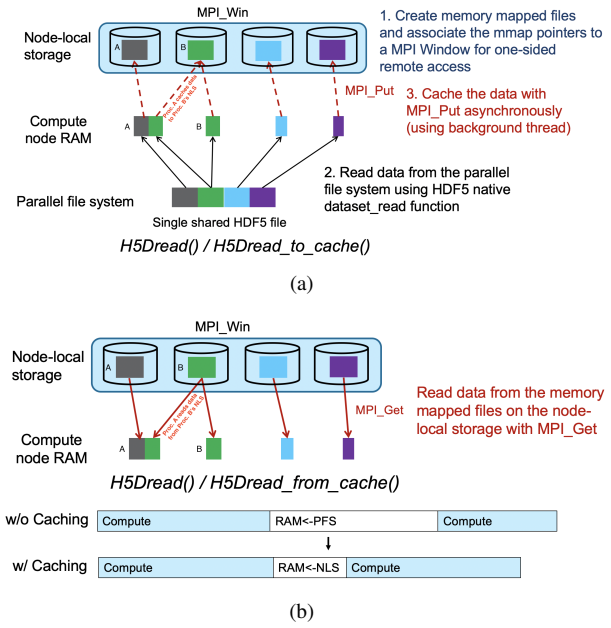


Fig. 3. *Cache VOL* parallel read. (a) on-the-fly caching data to the NLS. Once any new data are read from the parallel file system, we cache them to the NLS with MPI_Put; (b) prefetching the entire dataset to the NLS at once; (c) reading data from NLS with MPI_Get.

Similar to the parallel write, because of the use of memory map, no explicit memory allocation is needed. The overall memory footprint is thus under control without incurring an out-of-memory issue due to caching. What is more, if the entire dataset can fit into the aggregate DRAM, all the future reads will involve only MPI communication with no I/O overhead. Without caching data to the NLS, however, the future reads will still involve I/O overhead from the PFS since a single process does not cache the entire dataset in DRAM and it has to read data from the PFS if it is not cached already on the RAM. This is a side benefit of using memory map, which will be discussed in more details in Section IV.

The above whole framework of read caching can be easily implemented in any other I/O libraries through intercepting the read call. For HDF5, we intercept the dataset read function at the VOL level.

C. Cache space management

When there are more than one file using the NLS, it is important to manage the space in an optimal way to avoid any data overriding among different files. We manage the cache replacement in granularity of a file. Specifically, when a file is created or opened, *Cache VOL* will attempt to reserve a portion of NLS space according to the size of the dataset. If there is not enough space available, it will attempt to remove caches from other files to free up space according to the specified cache replacement policy, based on the history of cache accesses recorded. We support LRU (least recently used), LFU (least frequently used), and FIFO (first in first out). After a cache is removed, the next I/O requests associated to that file will be directed to the PFS. Caches will also automatically be removed once the file or the dataset is closed.

D. Enabling Cache VOL in an application

With the VOL framework, it is very easy for existing HDF5 applications to adopt *Cache VOL* without too much code modification. One simply needs to set environment variables HDF5_PLUGIN_PATH and HDF5_VOL_CONNECTOR to specify the location of the connector(s) and which connector(s) to be loaded in runtime and their relative order in the stacking chain. In the example below, three connectors are stacked, *Cache*, *Async*, and the native terminal VOL.

```
HDF5_PLUGIN_PATH=$HDF5_ROOT/../vol/lib
HDF5_VOL_CONNECTOR="cache_ext config=SSD.cfg;
    under_vol=512;under_info={under_vol=0;under_info={}}"
```

The information of the NLS, such as the size and path, is provided through a configure file, *SSD.cfg* in this case.

To enable caching for all the HDF5 files, one can set the environment variables HDF5_CACHE_WR and HDF5_CACHE_RD to yes. To enable it only for a specific file, one can set the values of HDF5_CACHE_WR or HDF5_CACHE_RD to true in the file access property list as follows:

```
/* enabling Caching VOL in parallel write */
...
hid_t pls_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(pls_id, comm, info);
H5Pset_fapl_cache(pls_id, "HDF5_CACHE_WR", True);
H5Fcreate(..., pls_id)
for(int iter=0; iter < nw; nw++) {
    H5Dcreate();
    H5Dwrite();
    ... // compute works
    H5Dclose();
}
H5Fclose();
```

```
/* enabling Caching VOL in parallel read */
...
hid_t pls_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(pls_id, comm, info);
H5Pset_fapl_cache(pls_id, "HDF5_CACHE_WR", True);
H5Fopen(..., plist);
H5Dopen(...);
// prefetch the dataset
H5Dprefetch(...);
for(int iw=0; iw < nw; nw++) {
    ... // some works before I/O
    H5Dread();
    ... // compute works
}
H5Dclose();
H5Fclose();
```

To take full advantage, it is important to insert compute work between the dataset write call and the dataset close call to overlap with the background data migration. This can be done by postponing dataset close calls. Best practices are provided in the user documentation [26].

IV. Experimental Setup

We conducted performance evaluations of *Cache VOL* on pre-exascale supercomputers, Theta/ThetaGPU [1] and Summit [3]. Theta is a Cray XC40, 11.7 petaflops system at Argonne Leadership Computing Facility (ALCF), based on the second-generation Intel Xeon Phi™ processor. The whole system contains 4,392 Intel Knights Landing nodes interconnected with Cray Aries Dragonfly. Each node has 16 GB of MCDRAM, 192 GB of DDR4 memory, and 128 GB of SSD. ThetaGPU is an extension of Theta. It is a 3.9 petaflops system comprised of 24 NVIDIA DGX A100 nodes. Each node has eight A100 GPUs, with 1 TB of DDR4 memory, 320 GB of GPU memory, and 4 NVMe drives each of 3.84 TB.

Theta and ThetaGPU are connected to three Lustre file systems at ALCF, including Eagle which was used for our evaluation [27]. Eagle is a Lustre file system residing on an HPE ClusterStor E1000 platform equipped with 100 Petabytes of usable capacity across 8480 disk drives. It has 160 Object Storage Targets and 40 Metadata Targets with an aggregate data transfer rate of 650 GB/s. In our experiment, we set the Lustre stripe size to be 16 MB, and the stripe count to be 64.

Summit is an IBM AC922, 200 petaflops system at Oak Ridge Leadership Computing Facility (OLCF), comprised of 4,608 nodes, each contains two 22-core IBM Power9 processors and six NVIDIA Tesla V100 GPUs interconnected with dual-rail Mellanox EDR 100Gb/s InfiniBand. Most Summit nodes contain 512 GB of DDR4 memory for CPUs, 96 GB of High Bandwidth Memory (HBM2) for GPUs, and 1.6TB of NVMe that can be used as a burst buffer. Summit is connected to an IBM Spectrum Scale filesystem providing 250PB of storage capacity with a peak write speed of 2.5 TB/s.

We first demonstrate the benefit of using *Cache VOL* in different situations using a set of microbenchmarks. We then select three workloads to evaluate the benefits of using *Cache VOL* in real applications, including a traditional HPC application, VPIC-IO [14], with heavy check-pointing I/O, and two deep learning applications, AlexNet [28] and CosmoFlow [15]. Both involve intensively reading of large among of datasets repetitively. The experiments for VPIC-IO were done on all three systems, and the experiments for AlexNet and CosmoFlow were done on ThetaGPU.

We run the applications with 16 MPI processes per node on Theta and Summit, up to 2048 and 256 nodes respectively, and 8 MPI processes per node on ThetaGPU up to 16 nodes.

A. Mircobenchmarks

1) Check-pointing microbenchmark: In the check-pointing microbenchmark, each process writes 16 MB of data to a shared HDF5 file at each iteration. At each iteration, we emulate the computation by having the main threads sleep. We vary different amount of emulated computation and measure the observed write bandwidth:

$$\text{Observed bandwidth} = \frac{\text{Amount of data}}{\text{Observed I/O time}}, \tag{1}$$

where the observed I/O time is the total time from creating the file to closing the file, excluding the emulated computation time with sleep operation. With this benchmark, we can investigate the benefit of hidden I/O behind the compute.

2) Read streaming microbenchmark: In this read microbenchmark, we are trying to mimic the machine learning workloads, where each process reads data from a shared HDF5 file in one batch per step. All the processes together read through the entire dataset in one iteration which might consist of multiple steps. The read pattern can be either sequential or random. In the sequential case, each process will be reading the same data in different iterations; whereas in the random case, each process will be reading different data in different iterations. We consider different scenarios: (1) the size of dataset is larger than the aggregate DRAM, so that no DRAM caching effect exists; (2) the size of the dataset is smaller than the aggregate DRAM, so that in the second and onward iterations, data will be directly read from the DRAM cache; (3) the size of the dataset is smaller than the aggregate DRAM, but there are other parts of the simulation occupy the DRAM, resulting no DRAM caching effect.

B. Scientific Applications

1) VPIC-IO: VPIC-IO is an I/O kernel extracted from VPIC [29], a plasma physics application for simulating the dynamics of plasma particles. The benchmark is currently included in h5bench [14], a suite of parallel I/O benchmarks or kernels representing I/O patterns that are commonly used in HDF5 applications on high performance computing systems. During the simulation, the application iteratively writes a large amount of check pointing data with a regular I/O pattern. There are a total of 8 million particles per process. Each process writes 8 properties associated with each particle, with a total of 256 MB, to a single shared HDF5 file.

2) AlexNet: AlexNet is a 2D convolutional neural network model, which was one of the first deep networks proposed for ImageNet classification [28]. The network contains eight layers, five convolutional layers with filters of size 11×11, 5×5 and 3×3, and three fully connected layers of size 4096. The datasets contain about 1.2 million images of 1000

categories. The goal is to train the CNN model to be able to classify the images. The datasets are stored as raw images. We preprocessed and stored them in single HDF5 file as a multiple dimensional array (1281167, 224, 224, 3) of uint8 datatype. The HDF5 file is about 180 GB. The AlexNet model is implemented using TensorFlow [30] with Horovod [31] for data parallel training at scale. The dataset is loaded from the HDF5 file with h5py and fed to the model through tf.data Pipeline [32] during the training. The entire dataset is loaded multiple times repetitively, one per epoch.

3) CosmoFlow: CosmoFlow is a 3D convolutional neural network model for learning the universe at scale [15], [33]. The model is implemented in TensorFlow Keras [30] with Horovod [31] for data parallel training at scale. The datasets contain 524288 samples for training and 65536 samples for validation. Each sample is a four dimensional array of size $(128, 128, 128, 4)$ which is a subset crop of a full universe image $(512, 512, 512, 4)$. The samples are initially stored in TFRecord format. We stored all of them in single shared HDF5 files as multiple dimensional arrays. The sizes of the HDF5 files are 8 TB and 1 TB respectively for training and validation. We also added support to the data loader allowing loading data from the HDF5 files with h5py and feeding to the model through tf.data Pipeline [32] during the training.

V. Results

To demonstrate the benefit of using *Cache VOL*, we compare the performance of *Cache VOL* with baseline HDF5. We report either the observed I/O rate or the overall time-to-solution.

A. Microbenchmarks

1) Check-pointing benchmark: We run the benchmark on Theta with 8 nodes, 16 processes per node. Each process is writing 16 MB data to a shared HDF5 file 8 times per iteration. The total number of iterations is 32. We vary the amount of compute per iteration. The baseline for writing this amount of data directly to the PFS is 112.35 seconds. It is shown in Table II that as we increase the amount of compute time, there is more overlap between the data migration and compute; the observed write time thus decreases. In particular, if the compute / IO ratio is larger than 1, the data migration is completely hidden behind the compute, resulting an observed write bandwidth, 4330 MiB/sec, which is close to the pure SSD write bandwidth, 4420 MiB/sec. Therefore, in order for *Cache VOL* to perform well, the application needs to have sufficient compute to overlap with data migration.

2) Data streaming benchmark: The results in Table III show performance improvement with *Cache VOL* in different scenarios compare to the baseline. With *Cache VOL*, data is cached to the NLS at first iteration, and read directly from there; with baseline, data is read directly from the PFS.

The experiments in (a) were performed on a single-gpu node on ThetaGPU, which has 128 GB DRAM and 15.36 TB NVMe. First, even with 256 GB dataset which is larger than the size of DRAM, *Cache VOL* still functions well. This shows that *mmap* is indeed able to map files larger than the DRAM into the virtual memory. In this case, there is no DRAM caching and data is read from the PFS at each iteration for the baseline, or read from the NVMe drive if *Cache VOL* is used. With *Cache VOL*, one achieves 1.5 - 2x higher read bandwidth than the baseline because NVMe has a higher read bandwidth than Lustre for both sequential and random read.

The experiments in (b) and (c) were performed on four Theta nodes, each node has 192 GB DRAM and 128 GB SSD. The size of the dataset is 64 GB which can be cached entirely into DRAM if no other parts of the application are competing for the memory. This is exactly the case for (b). DRAM caching effect exists in the second and onward iterations, in which the sequential read is essentially reading data from the DRAM cache. For the random read in the baseline case, majority of data is still read from the PFS, whereas in the *Cache VOL* case, data is obtained either from the local node's DRAM through memory copy or from remote node's DRAM through MPI_Get, resulting a read rate of 13.5GB/sec much higher than the baseline read rate. Therefore, *Cache VOL* can take better advantage of the DRAM caching effect for the random read when the dataset is small. This is the reason we see improved performance in AlexNet and CosmoFlow when *Cache VOL* is used.

In (c), we purposely read a lot of dummy files at each iteration to evict the cached data on the DRAM. This is to mimic the case where other parts of the application consume the DRAM so that the dataset cannot be cached entirely into the DRAM even if it is smaller than the size of DRAM. We again see higher random read rate in *Cache VOL* because the data is read from SSD and transferred to remote nodes through MPI. This is faster than reading from the PFS.

B. VPIC-IO

We calculate the observed write rate of VPIC-IO according to Eq. 1. We configured VPIC-IO to write 20 timesteps of data, all to a single HDF5 file with each time step in a different HDF5 group. We added sleep time which represents the computation time in real application runs, that is sufficient for the data migration from NLS to PFS to fully overlap with, which is typical with the VPIC application. In these experiments, we have set the sleep time to be 200 seconds on Theta and 20 seconds on Summit. In an actual VPIC simulation, computation time is typically more than 1000 seconds per iteration [29]. It is a future research topic to see whether the sleep function is a good substitute for computation. Real computation could impact the work of background threads that are being used for data migration as we have mentioned in III. Where the background threads are placed will also possibly be an important parameter to tune in the context of real simulations.

In Fig. 4, we find that with caching data on the NLS, the observed I/O rate scales linearly and eventually outperforms the write rate of the baseline. On Summit, with caching on NVMe, we achieved about 2 GB/sec per node with almost linear scaling efficiency. At small scale ($<$128 nodes), this is lower than the baseline, because of the lower per node

TABLE II

OBSERVED WRITE BANDWIDTH FOR THE CHECK-POINTING MICROBENCHMARK. THE TEST IS DONE ON 8 THETA NODES WITH 16 PROCESSES PER NODE. AT EACH ITERATION (32 IN TOTAL), EACH PROCESS WRITES 16 MB × TO A SHARED HDF5 FILE 8 TIMES. THE BASELINE FOR WRITING THE SAME AMOUNT OF DATA DIRECTLY TO THE PARALLEL FILE SYSTEM IS 112.35 SECONDS. THE COMPUTE / IO RATIO IS CALCULATED AS THE RATIO BETWEEN THE EMULATED COMPUTE TIME AND THE BASELINE WRITE TIME. THE AGGREGATE SSD WRITE BANDWIDTH IS 4420 MIB/SEC.

Compute time (sec)	0	32	64	96	128	160	192	224
Compute / IO ratio	0	0.28	0.57	0.85	1.14	1.42	1.71	2.00
Total time (sec)	175.16	199.24	216.58	228.34	249.40	280.56	312.97	344.92
Observed wrt time (sec)	175.16	167.24	152.58	132.34	121.40	120.50	120.97	120.92
Observed wrt bandwidth (MiB/sec)	2993.63	3134.97	3436.28	3961.74	4318.86	4349.03	4333.96	4336.00

TABLE III

READ BANDWIDTH (MIB/SEC) FOR THE STREAMING MICROBENCHMARK. EACH PROCESS READ DATA FROM A SHARED HDF5 FILE BATCH BY BATCH. AT ONE ITERATION, ALL THE PROCESSES TOGETHER READ THROUGH THE ENTIRE DATASET ONCE. FOR BASELINE, DATA IS READ DIRECTLY FROM THE PARALLEL SYSTEM; FOR CACHE VOL, AT FIRST ITERATION, DATA IS READ FROM THE PARALLEL FILE SYSTEM AND CACHED ON THE NODE-LOCAL STORAGE, AND AT LATER ITERATIONS, DATA IS READ DIRECTLY FROM THE NODE-LOCAL STORAGE

(a) 256 GB dataset on a single-gpu node

iteration	1	2	3	4
baseline (seq)	1648.83	1879.51	2044.39	1873.59
baseline (rnd)	1194.52	1300.26	1261.07	1285.54
Cache VOL (seq)	889.16	2684.45	3287.66	3318.28
Cache VOL (rnd)	759.74	2956.41	3372.95	3389.87

(b) 64 GB dataset on 4 KNL nodes (w/ DRAM cache)

iteration	1	2	3	4
baseline (seq)	7333.41	43543.90	108265.64	108278.99
baseline (rnd)	2761.32	2982.64	3129.56	3249.07
Cache VOL (seq)	6934.20	96640.38	95646.93	100286.40
Cache VOL (rnd)	1894.40	13568.01	13588.03	13472.66

(c) 64 GB dataset on 4 KNL nodes (w/o DRAM cache)

iteration	1	2	3	4
baseline (seq)	9136.53	9874.35	10937.79	10710.01
baseline (rnd)	2273.24	2058.01	2219.96	2167.5s
Cache VOL (seq)	3686.85	6469.02	7341.70	8122.71
Cache VOL (rnd)	1560.31	6140.09	3913.69	4398.17

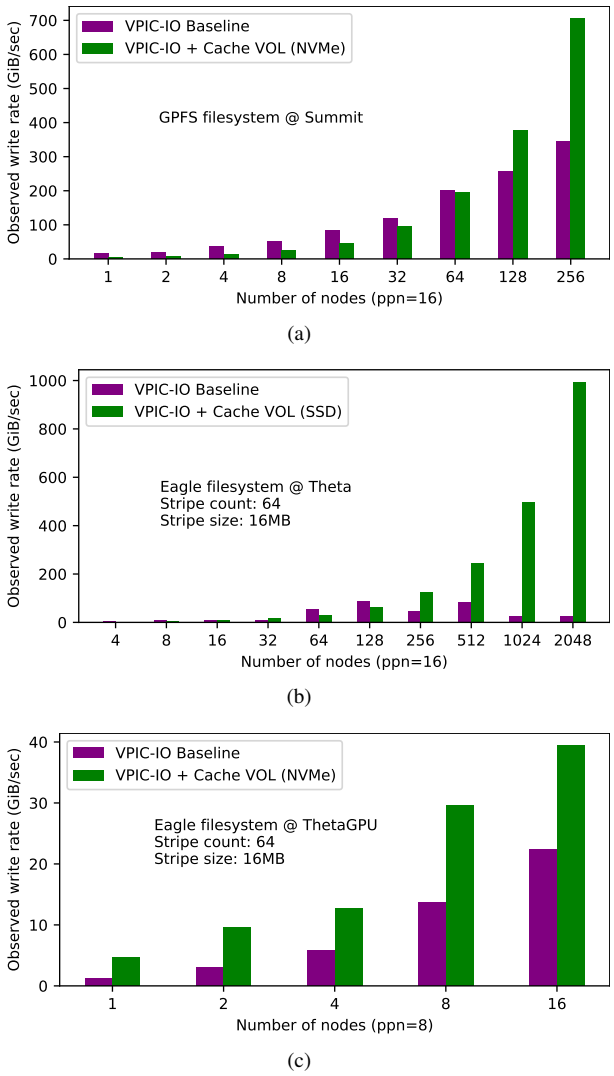


Fig. 4. VPIC-IO observed write rate on (a) Summit, (b) Theta, and (c) ThetaGPU. The number of time steps is 20. The write rate reported here is the average over the 20 time steps. The emulated time is 20 seconds per time step on Summit and ThetaGPU, and 200 seconds per time step on Theta.

bandwidth on NVMe (2 GB/sec) compared to GPFS (12 GB/sec). At large scale (> 128 nodes), however, baseline HDF5 does not scale due to interconnect contention from other jobs as well as the scaling bottleneck of a single HDF5 file. Caching data on NVMe, thus eventually outperforms the baseline at large scale.

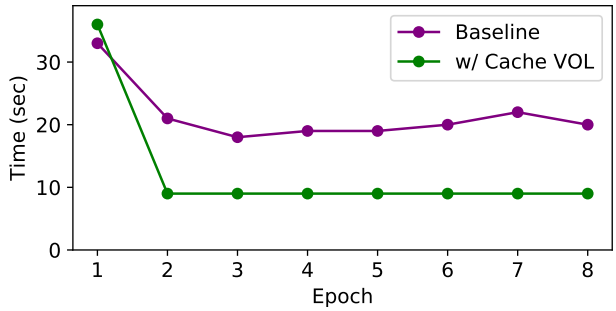
Similarly, on Theta with caching data on SSD, the observed write rate scales linearly and achieves a write rate about 500 MB/sec per node close to the 700 MB/sec peak performance of SSD; the baseline write rate, however, saturates at about 100 GB/sec at around 128 nodes and does not scale beyond that. On ThetaGPU, *Cache VOL* already shows benefit even at one node, because of the high bandwidth of NVMe.

C. AlexNet and CosmoFlow

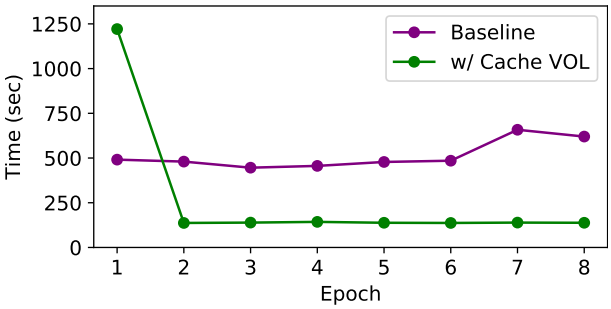
AlexNet and CosmoFlow are two read-intensive deep learning applications. At each training step, each process reads a random batch of samples from a shared HDF5 file and performs training. All the processes together read the entire dataset in one epoch. Data is shuffled at the end of each epoch, resulting a random I/O access pattern. Different data are read in different epoch for each process. This minimizes the DRAM caching effect. The training was performed on ThetaGPU with 128 Nvidia A100 GPUs. We measure the time for each training epoch.

In Fig. 5, we show the training time per epoch for AlexNet and CosmoFlow. We see that loading data directly from

the NLS cache reduces overall time-to-solution significantly. Except the first epoch which involves loading data from PFS and caching it to the NLS, the training time for later epochs is reduced to 1/3 - 1/2. As the training typically takes hundreds of epochs to finish, the overhead from the first epoch is negligible. The overhead in the first iteration can potentially be removed by performing caching asynchronously which is part of our future work.



(a)



(b)

Fig. 5. Improvement of training throughput by caching data on the node-local storage: (a) AlexNet and (b) CosmoFlow. The training were done on 16 DGX nodes with 128 Nvidia A100 GPUs.

VI. Related Work

There have been many software developments for transparent data movement in multiple levels of storage hierarchy, such as Data Elevator [34], UniviStor [7]. Data Elevator uses an HDF5 VOL plugin to intercept the I/O calls from the application and redirects them to the burst buffer. A Data Mover then moves the data from the burst buffer to a PFS in the background. Data Elevator relies on a shared burst buffer unlike *Cache VOL* developed in this work which is targeted at NLS. The Data Mover runs concurrently either on a separate set of nodes or on the same set of nodes but different cores.

UniviStor [7] was developed to efficiently manage data movement across distributed and hierarchical storage. It integrates node-local and shared storage devices into a unified storage space using a distributed metadata service to manage the address space. UniviStor also supports data caching at the fast storage and flushing them to the persistent PFS storage at file close. Similar to the Data Elevator, a UniviStor server needs to be started before launching the client applications on the same set of nodes. This increases the computing resources needed for running the application. Running Data Mover or UniviStor server together with the application on the same set of nodes might not be supported by the job schedule in some systems such as Theta. In contrast to Data Elevator and UniviStor, *Cache VOL* uses background threads for data migration, which can be deployed in all platforms. The background threads are put into sleep when there is no I/O to avoid occupying extra hardware resources.

There are many file systems designed for shared or local burst buffers. DataWarp [35] is an infrastructure for managing the PCI-attached SSDs located on the service nodes as a shared burst buffer. With DataWarp, the shared burst buffer can be used as a cache to the PFS, a scratch space to the application, or swap for the compute nodes [36]. BurstFS [5] and UnifyFS [6] on the other hand, are user level file systems which unify the NLS on all compute nodes and present a shared namespace, enabling the applications to use NLS for shared files. Common features for these user-level file systems contain: intercepting the applications I/O calls and send them to a server; a server running on the compute nodes for efficient metadata service to locate the data segments. Besides, DAOS (Distributed Application Object Storage) [37], [19] is an object storage system that encapsulates data of storage stack in DAOS containers and provides distributed transactional object store. *Cache VOL* currently does not rely on any underlying unify file system for NLS. For parallel write, the data staging process is local to each compute node; for parallel read, we used memory map files together with one-sided RMA to achieve efficient remote data access. Since we only focus on sample-based datasets and each read will select complete samples, it is relatively simple for locating the data. However, in future, we plan to support more generic read, which need more complex metadata service. *Cache VOL* can potentially benefit from the user-level shared file systems mentioned above.

Regarding caching data to the NLS, tf.data pipeline library also supports caching dataset either in memory or on local storage on the first epoch [38]. The later epoch will then automatically read data from the cache, similar to *Cache VOL*. However, the caching scheme provided by tf.data pipeline library is local to each process. Each process also read data only from the its own cache. Hence, it does not support remote cache access, and cannot support generic shuffling within the entire dataset. Insufficient shuffling might lead to potential degradation of training accuracy, the impact of which has yet to be investigated. Meanwhile, tf.data caching is restricted to TensorFlow but not other frameworks such as PyTorch [39]. *Cache VOL* however support all frameworks as long as the datasets are stored in HDF5 format. It is also rather straight forward to convert other formats to HDF5.

VII. Conclusion

We designed and implemented an HDF5 external VOL connector, *Cache VOL*, which incorporates node-local storage into the parallel I/O workflow for data caching / staging. This significantly improves the overall I/O performance. Specifically, we have demonstrated in various real applications that

by using the *Cache VOL*, one achieve higher and more scalable I/O performance over the direct I/O to the parallel file system. The VOL connector is designed in a way that is easy to be adopted by application developers with minimal code change.

ACKNOWLEDGMENTS

We thank Scot Breitenfeld and Elena Pourmal from the HDF Group for helpful discussion. This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231 (Project: Exascale Computing Project [ECP] - ExaHDF5 project). 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, as well as the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

REFERENCES

[1] "Theta/thetaGPU machine overview." https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview, accessed on September 25, 2020.

[2] "Using lc's sierra systems." https://hpc.llnl.gov/training/tutorials/using-lcs-sierra-system#NVMe, accessed on October 16, 2020.

[3] "Summit system overview." https://postk-web.r-ccs.riken.jp/spec.html, accessed on October 16, 2020.

[4] "Post-K (Fugaku) information." https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, accessed on September 25, 2020.

[5] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An Ephemeral Burst-Buffer File System for Scientific Applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 807–818, 2016.

[6] "UnifyFS: A file system for burst buffers." https://unifyfs.readthedocs.io/en/latest/, accessed on October 18, 2020.

[7] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated hierarchical and distributed storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 134–144, 2018.

[8] R. Thakur, W. Gropp, and E. Lusk, "An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces," in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*, pp. 180–187, IEEE, 1996.

[9] The HDF Group, "Hierarchical Data Format, version 5," 1997. http://www.hdfgroup.org/HDF5.

[10] "Reference omitted for review purpose." Reference omitted for review purpose.

[11] The HDF Group, "HDF5 VOL user guide." https://bitbucket.hdfgroup.org/scm/hdffv/hdf5doc.git, accessed on September 25, 2020.

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

[13] H. Tang, Q. Koziol, J. Ravi, and S. Byna, "Transparent asynchronous parallel I/O using background threads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 891–902, 2022.

[14] "H5bench: a parallel i/o benchmark suite for HDF5." https://github.com/hpc-io/h5bench.git.

[15] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, Prabhat, and V. Lee, "CosmoFlow: Using deep learning to learn the universe at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, IEEE Press, 2018.

[16] "HDF5." https://portal.hdfgroup.org/display/HDF5/HDF5, accessed on September 25, 2020.

[17] S. Byna, M. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "Exahdf5: Delivering efficient parallel i/o on exascale computing systems," *JCST*, 2020.

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

[19] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: A proposal for an exascale storage system," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 585–596, 2016.

[20] "Energy Exascale Earth System Model Project." https://github.com/E3SM-Project/E3SM.

[21] "Official github repository for netcdf-c libraries and utilities.." https://github.com/Unidata/netcdf-c.

[22] "Hdf5 for python – the h5py package is a pythonic interface to the hdf5 binary data format.." https://github.com/h5py/h5py.

[23] S. Rivas-Gomez, R. Gioiosa, I. B. Peng, G. Kestor, S. Narasimhamurthy, E. Laure, and S. Markidis, "MPI windows on storage for HPC applications," in *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, (Chicago, Illinois), pp. 1–11, Association for Computing Machinery, Sept. 2017.

[24] R. Matsumiya and T. Endo, "Scalable RMA-based Communication Library Featuring Node-local NVMs," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, Sept. 2018. ISSN: 2377-6943.

[25] O. Tatebe, S. Moriwake, and Y. Oyama, "Gfarm/BB — Gfarm File System for Node-Local Burst Buffer," *Journal of Computer Science and Technology*, vol. 35, pp. 61–71, Jan. 2020.

[26] "Reference omitted for review purpose." Reference omitted for review purpose.

[27] "ALCF Storage and Networking." https://www.alcf.anl.gov/alcf-resources/storage-and-networking, accessed on October 15, 2021.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84–90, May 2017.

[29] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. J. T. Kwan, "Advances in petascale kinetic plasma simulation with VPIC and roadrunner," *Journal of Physics: Conference Series*, vol. 180, p. 012055, jul 2009.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 265–283, USENIX Association, Nov. 2016.

[31] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv:1802.05799*, 2018.

[32] "tf.data: Build TensorFlow input pipelines." https://www.tensorflow.org/guide/data, accessed on October 15, 2021.

[33] "CosmoFlow TensorFlow Keras benchmark implementation." https://github.com/sparticlesteve/cosmoflow-benchmark.git.

[34] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-contention data movement in hierarchical storage system," in *HiPC*, pp. 152–161, IEEE, 2016.

[35] NERSC, "DataWarp Usage." http://www.nersc.gov/users/computational-systems/cori/burst-buffer/example-batch-scripts/.

[36] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and Design of Cray DataWarp," *Cray User Group CUG*, p. 11, 2016.

[37] Intel, "Intel Distributed Asynchronous Object Storage (DAOS)," 2014. https://daos-stack.github.io/.

[38] "tf.data: Build TensorFlow input pipelines: cache." https://www.tensorflow.org/api_docs/python/tf/data/Dataset#cache, accessed on October 15, 2021.

[39] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017.