Data Elevator: Low-contention Data Movement in Hierarchical Storage System

Bin Dong, Suren Byna, Kesheng Wu, Prabhat, Hans Johansen, Jeffrey N. Johnson, and Noel Keen
Lawrence Berkeley National Laboratory, USA
Email: {dbin, sbyna, kwu, prabhat, hjohansen, jnjohnson, ndkeen}@lbl.gov

***Abstract*—Hierarchical storage subsystems that include multiple layers of burst buffers (BB) and disk-based parallel file systems (PFS), are becoming an essential part of HPC systems to address the I/O performance gap. However, the state-of-the-art software for managing these hierarchical storage subsystems, such as Cray DataWarp, requires user involvement in moving data among storage layers. Such manual data movement may experience poor performance because of resource contention on the I/O servers of a layer for serving data movement in the hierarchy as well as regular read/write requests. In this paper, we propose a new system, named *Data Elevator*, for transparently and efficiently moving data in hierarchical storage. Users specify the final destination for their data, typically a PFS. *Data Elevator* intercepts the I/O calls, stages data on a fast persistent storage layer (for example, an SSD-based burst buffer), and then asynchronously transfers the data to the final destination in the background. *Data Elevator* reduces the resource contention on BB servers via offloading the data movement from a fixed number of BB server nodes to compute nodes. The number of the compute nodes is configurable based on the data movement load. *Data Elevator* also allows optimizations, such as overlapping read and write operations, choosing I/O modes, and aligning buffer boundaries. In our tests with large-scale scientific applications, *Data Elevator* is as much as 4.2× faster than Cray DataWarp, and 4× faster than directly writing data to PFS.**

I. Introduction

As high-performance computing (HPC) systems rapidly grow in computing power, HPC applications can generate massive amounts of data. However, the improvement in the speed of disk-based storage systems has been much slower than that of memory, creating a significant I/O performance gap [24], [12]. To reduce the performance gap, the storage subsystem is going through extensive changes [8], [5], by adding multiple levels of memory and storage in a hierarchy, as shown in Fig. 1.

As multiple layers of storage, especially those acting as burst buffers (BB)[1], are introduced into a HPC system, the complexity of data movement among the layers increases significantly, making it harder to take advantage of the high-speed or low-latency storage systems [2], [5]. I/O system software and middle-ware to manage these intermediate layers of storage could help obtaining superior I/O performance. Ideally, the presence of multiple layers of storage should be transparent to applications without having to sacrifice I/O performance [22]. Meanwhile, it is critical to optimize write

[1]We will frequently use "BB" to refer a SSD-based burst buffer, and "PFS" to refer to a disk-based parallel file system, in the rest of the paper.

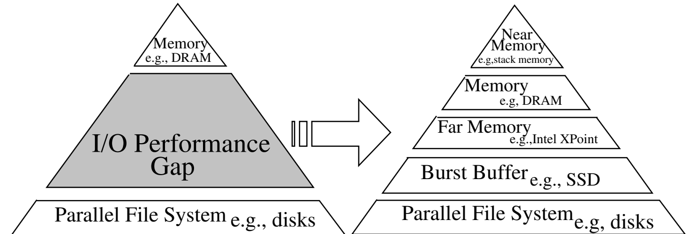


Fig. 1: An example HPC storage hierarchy to fill the I/O performance gap between main memory and disk.

performance across multiple storage layers because most scientific simulations generate large amounts of data and storing the data for further analysis is needed to reduce effort and energy consumption in rerunning simulations [24], [19], [26].

A few software solutions have been developed for handling BB-based hierarchical storage systems. For example, the latest solutions include DataWarp from Cray [9] and Integrated Memory Engine (IME) from DDN [22]. A pre-exascale HPC system, named Cori at the National Energy Research Scientific Computing Center (NERSC)[2] has Cray DataWarp installed. On this system, the BB space is only usable while a job is running and therefore users must move their data to PFS explicitly if they want to store the data permanently. Users can issue DataWarp staging commands after a job finishes to move the data from BB to PFS. Users can also modify their simulation codes to call DataWarp programming interface to perform moving data to PFS asynchronously. In either case, DataWarp uses a fixed number of BB server nodes to perform the data movement, indicating a physical bound on the I/O parallelism. To make things worse, especially for write-intensive simulations, these BB nodes also need to serve regular write requests from applications concurrently during the data movement. This causes contention on BB server node resources (e.g., BB node memory and I/O bandwidth) and results in poor data movement performance from BB to PFS.

To address the issues mentioned above, we design and implement *Data Elevator* system for users to run their simulations, especially write-intensive applications, more efficiently and without any user involvement in data movement. Specifically, *Data Elevator* intercepts I/O calls from applications using binary instrumentation method. This approach removes user involvement of modifying source code of applications to perform data movement. Applications issue I/O calls to write data to the final destination of data on PFS, and *Data Elevator*

[2]http://www.nersc.gov/users/computational-systems/cori/

intercepts and stages the file on BB for faster I/O. Once the data is written to the faster BB, the application can continue with its computation, while the *Data Elevator* moves the data to PFS asynchronously. To reduce the 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 [14]. When the data is in BB, *Data Elevator* also allows performing *in transit* analysis tasks on the data, before it is moved to the final destination of the file specified by the application. In summary, the contributions of this paper are the following:

- Conducting performance analysis of the first real BB-based hierarchical storage system on a pre-exascale HPC system. Our findings, including the resource contention on BB servers and poor performance of MPI collective I/O on BB, are novel observations.
- Designing and developing *Data Elevator* system to support low-contention data movement in hierarchical storage systems. *Data Elevator* offloads the data movement task from BB servers to computing nodes, using a different data flow path from DataWarp. A challenge we faced is reducing the movement overhead along the path. We successfully demonstrated that combining various well-known optimization techniques, including overlapping reading from BB and writing to PFS and striping alignment on Lustre, can address the challenges.
- Providing asynchronous I/O support via *Data Elevator* for permitting a simulation to move data from BB to PFS while the simulation concurrently performs its computations. *Data Elevator* also supports *in transit* analysis while the data is in BB. Such *in transit* analysis can reduce the data to be written to PFS.

We have evaluated the performance of *Data Elevator* by comparing it with Cray DataWarp, the state-of-the-art software for BB. Our evaluation shows that *Data Elevator* is 35.2% faster in moving data across storage layers. We have also successfully applied *Data Elevator* to two real scientific applications: a plasma physics simulation named VPIC [4] and a global atmospheric dynamics simulation called CAMR [16]. The results show that *Data Elevator* is as much as 4.2 times faster than DataWarp in completing the write operations and 4 times faster than writing directly to PFS.

The rest of the paper is organized as follows: We describe the background and the motivation to our work in Section II. In Section III, we present the design of *Data Elevator* and our current implementation. In Section IV, we describe our experimental setup and in Section V, we evaluate the performance of *Data Elevator*. We discuss related research in Section VI and conclude the paper with a discussion of future work in Section VII.

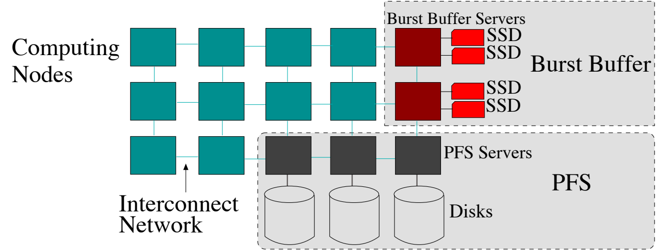


Fig. 2: Burst Buffer on Cori Supercomputer at NERSC.

II. Background

We briefly describe Cray's DataWarp and HDF5's Virtual Object Layer (VOL) technologies that we used in designing *Data Elevator*.

A. Burst Buffer and DataWarp

Building hierarchical storage systems is a cost effective strategy to reduce the I/O latency of HPC applications. The most recent hierarchical storage system for HPC is the burst buffer (BB) being installed at Cori, a Cray XC40 system at NERSC. We show a high-level architecture of Cori with a BB and its other components in Fig. 2. The BB contains a group of specialized server nodes and each server has two Intel P3608 3.2 TB NAND flash SSD modules installed on PCI-E bus. The internal architecture and tasks of SSD (e.g., garbage collection or wear leveling) are managed by the server nodes. The BB only manages the aggregated storage pool from each SSD. These BB server nodes are connected to computing nodes and Lustre [13] file system via high-speed interconnect network.

Cray DataWarp [9] software aggregates the storage space of all BB server nodes together as a single storage image. A user can request and reserve a part of the BB space through a SLURM job script, for an application to stage the data. Since SSD has persistence property, the allocated BB space for user is always cleaned by the DataWarp at the back-end. The BB request size is ≈ 200GB on Cori and when a user's request is large (i.e., $> 200GB$), multiple BB nodes are allocated and user's data is distributed across the allocated nodes. The stripe size is fixed at 8 MB. In contrast to Lustre, where the stripe size can be changed by a user, BB stripe size cannot be changed. In addition, the BB is shared across all users, which may cause I/O contention from several applications.

As shown in Fig. 2, BB and PFS (Lustre) on Cori are two independent components. Users need to manage the data movement between Lustre and the BB manually. To facilitate data movement, DataWarp provides job script based commands (e.g, `stage_out`) and a programmable library. Users can call these commands to move the data synchronously via job script or by modifying their application source code to use the DataWarp library to move the data asynchronously. Internally, after receiving the data movement commands from users, BB server nodes move the data from the BB to Lustre. Since these BB server nodes also need to serve the I/O requests from applications concurrently, using the BB servers to move data, especially for terabyte-scale or even larger volume of data, could cause significant contention for the resource on the server nodes. Hence, using DataWarp for performing the

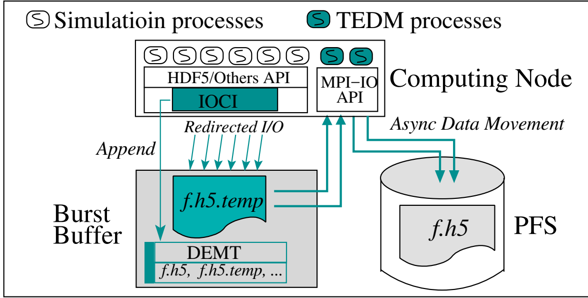


Fig. 3: High-level overview of *Data Elevator* for burst buffer and parallel file system

data movement could result in slow performance, as was we observed in our tests.

Moreover, since the number of DataWarp servers is fixed when building a HPC system, its parallelism for moving data from the BB to PFS is limited. If considering that a BB server may be also shared by several users concurrently, the performance of moving from burst buffer to Lustre could become worse. Hence, using the BB to temporarily store the data from simulation code may consume more time than directly writing data to a disk-based file system. In this study, we are addressing the issue of poor performance in moving data from a BB to a PFS and also providing users a transparent and flexible way to use hierarchical storage systems.

B. HDF5 and VOL Plug-in

HDF5 is a popular scientific data and file format library [6] used by many scientific applications. HDF5 provides efficient data organization on disk and supports parallel I/O using MPI-IO. Virtual Object Layer (VOL) is a new and light-weight abstraction layer internal to the HDF5 library and is right below the HDF5 API [6]. VOL supports interception of the HDF5 API calls that could potentially touch an I/O request to a file and forward those calls to a plug-in "object driver". VOL also provides flexibility to monitor the file access properties, e.g., file name, file path, etc. In this study, we use the VOL feature to monitor the file open and close calls of an application using HDF5 to perform I/O. We also use this feature to intercept the application data write and read calls and redirect these calls to a burst buffer.

III. Data Elevator Design and Implementation

The high-level goal of *Data Elevator* is to provide a transparent and efficient mechanism for moving data across several layers of hierarchical storage subsystem of HPC. In this section, we describe the design and implementation details for *Data Elevator* in moving data between BB and PFS.

A. Data Elevator Design Overview

The issues we target to address in this paper are: 1) to provide a transparent mechanism for using a 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 first issue arises because burst buffers are introduced in HPC systems as independent storage spaces. 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 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, we design *Data Elevator* to perform 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. We present a high-level architecture of *Data Elevator* in Fig. 3. Overall, *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 a VOL plug-in (details in §III-B). While the current implementation supports HDF5-based I/O, the implementation can be extended to other I/O interfaces, such as MPI-IO and PNetCDF [14]. DEMT contains a list of metadata records, e.g., file name, for the files to be redirected to BB (details in §III-C). The Data Mover (TEDM) component is responsible for moving the data from a burst buffer to a PFS based on the metadata. TEDM component can share the nodes with the application job or run using a separate set of compute nodes (details in §III-D). In Fig. 3, the TEDM shares two of the eight CPU cores with a simulation job (that uses the remaining six cores) on a computing node.

To use *Data Elevator*, users only need to compile their existing application code using the HDF5 library with the IOCI VOL plug-in. Then, users can start their application and TEDM at the same time with their preferred configurations, e.g., the number of processes. When the application writes their data to a parallel file system (PFS), IOCI traps I/O calls, creates temporary files on burst buffer, and redirects the data to burst buffer. Meanwhile, the metadata information, e.g., file name and the progress information of writing, will be appended to a metadata table by the IOCI.

Data Elevator applies various optimizations to improve I/O performance in writing data to BB (details in §III-B). 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.

Data movement from BB to PFS starts after a file is written to the BB and is closed. At this stage, the data is in a persistent state and the file is deleted only after the data is written to the destination specified by the application. In case of any failure of moving the data from BB to PFS, the Data Mover restarts to the entire file transfer to avoid any consistency. While we have not implemented at the moment, we can use checksums to guarantee the integrity of data.

B. *Light-weight I/O redirection and optimizations*

One key objective of *Data Elevator* is enabling users to use a hierarchical storage system effectively without modifying their existing application source code. To this end, we design the IOCI plug-in to intercept popular I/O interfaces. A main design consideration in developing IOCI is to minimize the overhead of I/O redirection. When an application opens or creates a file on PFS (e.g., file *f.h5* in Fig. 3), the IOCI plug-in captures the file name and creates a corresponding temporary file (e.g., file *f.h5.temp* in Fig. 3) on the BB. The file pointer to the temporary file is passed back to the caller in place of the file pointer to the file on PFS. Using this file pointer, the subsequent write operations for the file on PFS are redirected to the file on the BB. IOCI adds the information of the file name on disk and the file name on the BB to the metadata table as a new record. To detect the completion of writing the file, IOCI also intercepts file close calls. Once a file close call is detected, IOCI updates the *status of the file* in the metadata table. To mask the overhead of creating new file in IOCI, the Data Mover pre-allocates a list of files while it is idle. The only cost of the I/O redirection is to create a new record in the metadata table and update the completion status of the record.

By default, *Data Elevator* takes an entire file as a unit of I/O redirection, instead of smaller units such as disk block, memory page, etc., to reduce overhead of managing metadata for potentially a large number of smaller units of data. Using HDF5 dataset and HDF5 group as basic granularity of I/O redirection may not be portable because I/O libraries other than HDF5 may not support these concepts. For these reasons, *Data Elevator* treats the whole file as the default unit for I/O redirection, but it also provides other options to users.

Optimizations in writing data to a burst buffer. Although SSD-based burst buffers are faster than disk-based PFS, in *Data Elevator*, we use optimizations to improve writing data to BB depending on the HPC system characteristics. For instance, in our tests with the burst buffer on Cori, we found that the performance of MPI-IO with collective buffering to write data to the burst buffer is worse than that of MPI-IO with independent I/O. We found that it is caused by lock management issues of the storage manager for BB [3]. On disk-based Lustre file systems, existing simulation codes usually deliver better I/O performance using the collective I/O mode than in the independent I/O mode. Therefore, most existing simulation codes use collective I/O. To avoid the need for users to change from collective I/O to independent I/O, the IOCI plug-in is designed to detect collective I/O function calls in applications. If *Data Elevator* finds applications using collective I/O, the independent I/O mode is used for writing data to burst buffer. In the current implementation with HDF5 library, the IOCI plug-in also intercepts `H5Pset_dxpl_mpio` and other related functions, and disables collective I/O, when detected.

C. *Consistent metadata management for coordinating jobs*

We use a metadata table named DEMT on BB for the communication among simulation application job, any *in transit* analysis job and the Data Mover job of *Data Elevator*. DEMT also works as a journal to recover from any potential errors during data movement. This approach removes the complexity of involving all jobs to maintain and monitor the status of other jobs. Since these jobs may access DEMT concurrently it is critical that they have a consistent view of it.

Status	Description
W	Start writing to BB
B	Finish writing to BB
A	Start analysis
M	Finish analysis
D	Start moving to PFS
F	Finish moving to PFS

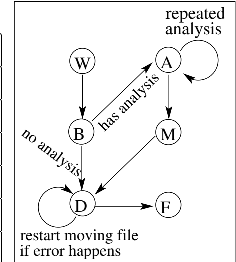


Fig. 4: The metadata indicating the status of a file being maintained by *Data Elevator* and the state transitions.

The DEMT contains four columns: the *name of file on PFS*, the *name of temporary file on BB*, the *file status* and the size of the file on PFS. The *name of file on PFS* is the name of the file used by the simulation job to store its data on PFS. For a data analysis job, the *name of file on PFS* is the file containing the data to be analyzed. The *file status* includes one of six categorical values shown in Fig. 4 at any instance.

The status transition diagram is also shown in Fig. 4. To provide a consistent view of the DEMT among multiple jobs, only a single process (usually MPI rank 0) from a certain job is used to access the DEMT and only a job can update it at one time. Specifically, a new record in DEMT containing the *file name on PFS* and *file name on BB* is appended once from an application job via the IOCI plug-in. The record is deleted by the Data Mover job after moving the data to the PFS. Hence, there is no conflicting access to the *file name on PFS* and *file name on BB*. After a new record is appended to DEMT, **W** and **B** is sequentially assigned to the *file status* by the IOCI plug-in. Until its status changes to **B**, a data analysis job can update its status to **A** and **M**. Only after **M** of a file is detected, the Data Mover job can update the status to be **D** and move

its data to PFS. Once data is written to the destination, the Data Mover updates the status to **F**. For simulations without any *in transit* jobs running, the Data Mover can directly move the data of a file after its status changes to **B**.

For supporting in-transit data analysis tasks that do not update DEMT, this consistency strategy can be simplified further to improve the overall execution time of these jobs. Specifically, after the status of file changes to **B**, the Data Mover and analysis jobs can start to work on the data on the BB at the same time. In this case, no status updates from analysis jobs are needed. The Data Mover updates the status from **B** to **F** when the data is moved to PFS. At the same time, analysis job can delete the record and file after it is finished.

D. Low-contention data movement

The Data Mover is responsible for moving data between a burst buffer and a PFS efficiently with low contention on the BB server nodes. Toward this goal, we propose to run a Data Mover job on computing nodes. With this approach, the data to be written to PFS from the BB is first moved to memory on the compute nodes and then written to PFS. The Data Mover nodes use efficient parallel I/O techniques in writing the data to PFS. Since moving the data from BB to the memory is faster than writing data to PFS, resource contention on BB servers will be less compared to using BB servers for writing to PFS. The CPUs of BB servers can spend more time on serving other I/O requests, while moving the data to PFS is offloaded to the Data Mover nodes. Writing data from the compute nodes gives flexibility to apply various parallel I/O performance tuning optimizations compared to DataWarp, which as fixed data movement strategy installed on BB servers. Toward the goal of improving performance of the Data Mover job, we have explored various optimizations.

Scalable parallel data movement. The Data Mover job is highly parallel in its design and have the potential to use an arbitrary number of parallel processes to perform data movement. As a result, the time used to move data can be sharply reduced, thus reduce the chance of resource contention on burst buffers. When a new file on the burst buffer is ready to be moved, Data Mover checks the file size and partitions the file evenly among its available processes. For the file whose size is larger than the aggregated physical memory size of compute nodes that the Data Mover job is running on, the file is moved in several parts, called chunks. The chunk size is determined by the available memory size for the compute nodes. Moreover, to fully utilize the parallel bandwidth from compute nodes to parallel file systems, the Data Mover job can be run on all computing nodes where the simulation job is running, which is called *shared mode* (as shown in Fig. 3). Users can also choose to run Data Mover on dedicated nodes, which is called *disjoint mode*.

Overlapping reading from BB and writing to PFS. The two main tasks of a Data Mover job is to read data from a burst buffer and to write the data to a PFS. To optimize these tasks, we propose overlapping the read and the write operations. Such overlapping is possible because the burst buffer and the parallel file system are two independent components in a HPC system. Overlapping reading and writing can keep both of them busy and reduce the overall time.

Stripe size alignment. Toward improving the performance of writing data to PFS, we align the size of an I/O request from Data Mover job with the stripe size of the PFS. It is possible to choose different stripe size for the moved file because IOCI (in previous subsection) delays the file creation on PFS. The *Data Elevator* can set the stripe size using Lustre command or API when it decides to move the file to PFS. Stripe size alignment can reduce the number of Object Storage Targets (OSTs) involved for a single file request. In other words, each write request from Data Mover is sent to a single OST and therefore reduces the I/O contention of different TEDM requests. *Data Elevator* also uses independent I/O rather than collective I/O on Cori. Since Data Mover performs reading and writing on the file in contiguous pattern, using independent I/O can avoid extra memory copy to MPI-IO aggregators.

We implemented Data Mover using a master-worker architecture, where MPI rank 0 functions as the master and all the other MPI ranks are workers. As an optimizations, the master process checks the metadata table periodically and when it finds that a file is available to be moved from BB to PFS, it broadcasts the file name to all the other processes to start moving the file. Using a single MPI rank to access the metadata avoids any scaling issues because it only requires retrieving the file name to be moved. All processes work on the same file concurrently using the MPI-IO library.In our current implementation, the Data Mover job needs to be initiated with the application job by a user. However, HPC facilities can run the Data Mover job as a service without requiring a user to run the job.

E. In transit Data Analysis

Several scientific simulations are performing analysis of data while the data is either in memory or in a burst buffer to avoid costly data access to parallel file systems. This type of analysis is called *in situ* or *in transit* analysis. These analysis applications can be start in conjunction with the simulations and read the data written by the simulations. Using the same method of redirection of writing data produced by a simulation to the burst buffer, *Data Elevator* also supports *in transit* analysis through redirecting the read requests of analysis applications accessing data from the PFS to accessing from the burst buffer. Specifically, the analysis codes can also link to the IOCI plug-in for redirecting their PFS requests to the burst buffer. This plug-in can automatically intercept the file open calls and update metadata for the file under analysis.

IV. System Configuration

We have conducted our evaluation on Cori Phase 1, a Cray XC40 supercomputer at NERSC. Cori Phase 1contains $1,630$ 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' (as shown in Fig. 2). The Burst Buffer is managed by DataWarp from Cray and have 144 DataWarp server nodes. 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 8MB, which cannot be modified by normal users. As such, we set the striping size of Lustre file system also to be 8MB.

V. Performance Evaluation

To evaluate *Data Elevator*, we have conducted two sets of experiments. We first tune various configuration parameters of *Data Elevator* using parallel I/O kernels extracted from scientific applications. We then use two real applications for demonstrating the advantages of *Data Elevator*. Following the typical workflow in scientific applications, we assume all data files need to be stored on the disk-based PFS for future analysis. We use *end-to-end execution time* of applications and *end-to-end data movement time* as the metrics for performance evaluation.

- *End-to-end execution time* measures the elapsed time of an application. This time includes computing and communication times to run a simulation and I/O time to access data from or to PFS. When the burst buffer is involved, I/O time includes the time to write the data to the burst buffer and the time to move the data to the PFS.
- *End-to-end data movement time* is a portion of the end-to-end execution time. It specifically measures the I/O time that an application spends to move data from the memory to Lustre. For DataWarp and *Data Elevator*, it includes the time to write the data to the burst buffer and the time to move data from the burst buffer to the PFS.

A. Tuning Data Elevator using I/O benchmarks

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 generates a single HDF5 file. VPIC-IO [4] 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 64GB in size. Chombo-IO is derived from Chombo, a popular block-structured adaptive mesh refinement (AMR) library [1]. The generated file has a problem domain of $256 \times 256 \times 256$ and is of 146GB in size.

Writing data to the burst buffer. Data Elevator can automatically set or unset the MPI-IO collective buffering mode in writing data to a burst buffer. In Fig. 5a, we show the I/O rate of VPIC-IO with and without MPI-IO collective buffering and without that mode using different numbers of cores. We have observed that independent I/O mode outperforms collective buffering mode significantly. VPIC-IO uses collective buffering, by default, to write its particle data as that mode obtained the best performance for writing data to Lustre [4]. As we observe that independent I/O is performing better on the burst buffer, we select independent I/O regardless whether the user requested collective buffering mode or not. This improves performance and does not require any application code change.

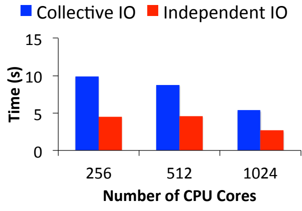
Overlapping of reading intermediate file from the burst buffer and writing to Lustre improves the performance of *Data Elevator*. In Fig. 5b, we show the end-to-end data movement time with and without overlapping the read and the write operations. For both benchmarks, we can clearly observe that overlapping reading from the burst buffer and writing to the PFS can reduce the data movement time by 46% for VPIC-IO and by 32% Chombo-IO. Thus, we conclude that overlapping the read and the write operations is an effective strategy. In the remaining tests, we overlap the read and the write operations.

Aligning the request size of the data movement with the stripe size of the Lustre PFS is another optimization strategy used in *Data Elevator* to accelerate the data movement to PFS. We compare the performance of writing the data with and without aligning the Data Mover's I/O request size with Lustre stripe size. On average, aligning the request size with the stripe size increases write performance by 65%. This result indicates that such alignment is an effective strategy for *Data Elevator*. In the rest of the tests, we have set the requests size to be equal to the stripe size.

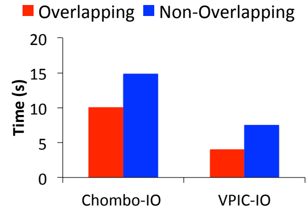
Using the compute nodes in shared or disjoint modes. In Fig. 5d, we show the end-to-end data movement time with *Data Elevator* sharing a small portion of each node where the data producing application is running and that using a set of dedicated nodes, for both I/O kernels. Running *Data Elevator* in the shared mode can save up to 92% of the time spent on the data movement on average for the two benchmarks. In the shared mode, we have used 16 CPU cores out of the 32 cores on each node, distributing 1024 the Data Mover job processes on 64 nodes. The simulation application was running at the same scale on the 64 nodes. In the disjoint mode, 1024 the Data Mover job processes were run on 32 nodes and the 1024 simulation processes are running on another 32 nodes. We observed that the performance of Data Mover is better in the shared mode because more bandwidth is available between compute nodes and PFS.

Varying the number of Data Elevator nodes in data movement. Using a large fraction of CPU cores for the Data Mover may not be feasible because simulations have high demand for the cores, we have varied the fraction of cores for *Data Elevator*. In Fig. 5e, we have varied the size of the Data Mover job in proportion to the simulation job as "1:1", "2:1" and "4:1". For instance, in "4:1" configuration, simulation job runs on 1024 cores and the Data Mover job runs on 256 cores. The results show that as we decrease the number of CPU cores for running the data movement job from 1024 to 256, the time increases only by 26% on average for both benchmarks. Although we use "1:1" configuration in this paper because the simulations we tested are not CPU demanding, the ratio is a configurable option in *Data Elevator*.

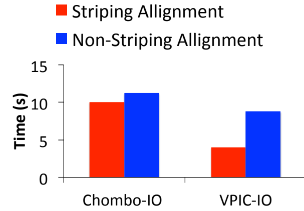
Metadata overhead in Data Elevator. Managing metadata table is one of important tasks in *Data Elevator*. This overhead has to be negligible for efficient use of *Data Elevator*. In Fig. 5f, we compare the metadata management overhead



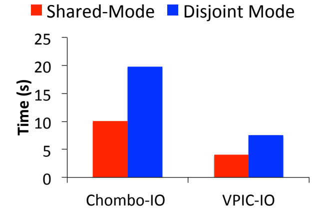
(a) The time of writing VPIC-IO data to burst buffer with collective I/O or independent I/O.



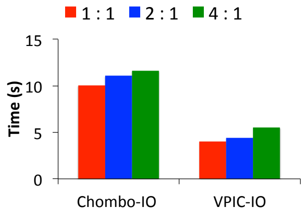
(b) Data movement time with and without overlapping reading from BB and writing to Lustre.



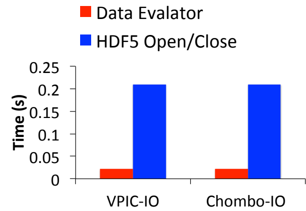
(c) Data movement time with or without striping alignment.



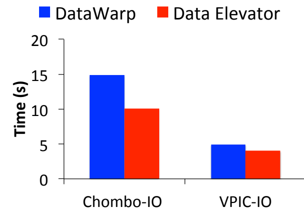
(d) Data movement time running as shared or disjoint modes.



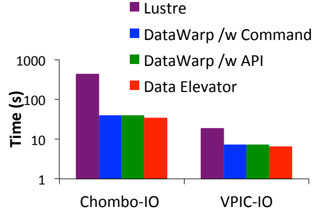
(e) Performance with the varying number of CPU cores.



(f) Metadata operation overhead in *Data Elevator*.



(g) Performance comparison of moving data from BB to Lustre.



(h) End-to-end data movement time.

Fig. 5: Performance evaluation using VPIC-IO and Chombo-IO, both has a single step and writes data to a single HDF5 file.

with opening or closing a single HDF5 file. The metadata management includes appending new record to metadata table and updating the status of a file in the table. As can be observed, the time for metadata operation in *Data Elevator* is a negligible 11% of opening or closing a file.

B. *Performance Evaluation using benchmarks*

Comparison with DataWarp. With the optimizations and configurations described above, we compare the *Data Elevator* with DataWarp, the state-of-art software for the burst buffer. In this test, DataWarp uses all 144 nodes to move data from the burst buffer to the Lustre PFS. In *Data Elevator*, we used 64 compute nodes to move data as using "1:1" configuration discussed above. For both benchmarks, *Data Elevator* outperforms DataWarp `stage_out` command, as shown in Fig. 5g. *Data Elevator* is 22% faster than DataWarp command for moving VPIC-IO data, and is 48% for Chombo-IO data.

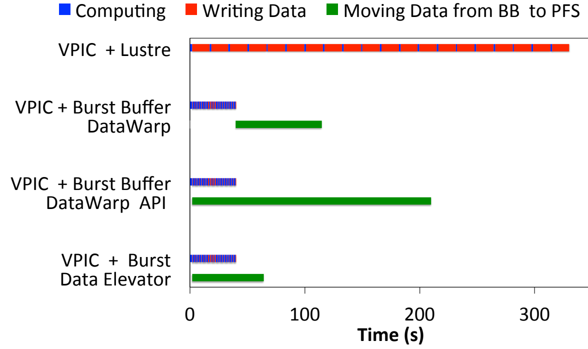
Comparison of the end-to-end data movement time. We show the end-to-end data movement time in running these two benchmarks with different configurations in Fig. 5h. PFS performance includes the time to directly write data to Lustre. For DataWarp and *Data Elevator*, the time for writing data to burst buffer and the time for moving the written data to parallel file system are included. DataWarp provides a job script based command (i.e., `stage_out`) and a library-based API to move data. With the command, DataWarp can move data only at the end of an application. Using the API, benchmark codes need to be modified to move the data after the file is written. The staging out of data can be done asynchronously in this mode. We test these two modes of DataWarp separately. Their performance is equal in this test because both benchmarks are running just for a single time step. From these results, it easy to identify that writing the data to burst buffer having smaller end-to-end data movement time. In short, using DataWarp and *Data Elevator* can reduce the end-to-end execution time, however, *Data Elevator* outperforms DataWarp based techniques by 14% for both benchmarks.

C. *Performance evaluation with science simulations*

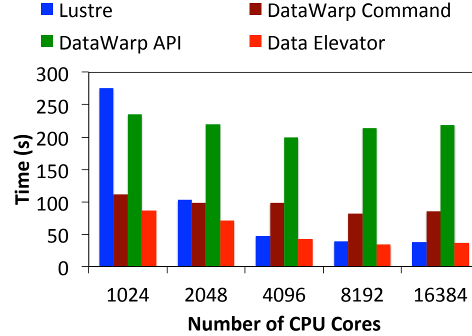
We evaluate *Data Elevator* using two real science applications: a plasma physics simulation studying space weather (VPIC), and a climate simulation studying atmosphere dynamics (CAMR). Both of these simulations perform computations and produce output files periodically that need to be stored to storage for future analysis.

1) Plasma physics simulation: VPIC (vector particle-in-cell) is an important code for simulating magnetic reconnection via monitoring electron motion in 3D space [4]. In evaluating *Data Elevator*, we used a medium scale (for matching the job size limit on Cori). We have run an open-boundary magnetic reconnection simulation for 20 time steps. At the end of each step, VPIC writes a single HDF5 file containing properties of particles. The file size for each step in this simulation was $88GB$, with a total of 1.7TB data for the 20 steps. Since the portion of computation in this simulation is small, VPIC tends to be extremely I/O intensive in our tests.

In Fig. 6a, we show the end-to-end execution time in detail for a test case with 1024 CPU cores. Using Lustre PFS to store the data, VPIC has the largest end-to-end execution time. Staging data onto burst buffer temporarily reduces the end-to-end execution time significantly for both DataWarp and *Data Elevator*. Since the `stage_out` command of DataWarp can only be issued in a batch job script after the simulation is completed, the DataWarp command option has to wait for data from all the time steps is written to the burst buffer. The asynchronous API of DataWarp library enables to move data to PFS once a time step file is on the burst buffer. However, because DataWarp servers are perhaps busy with requests from other applications along with writing the frequently arriving files from the VPIC simulation, resource (e.g., cache) contention on BB servers makes the performance of data movement is low even with asynchronous option. With



(a) A view of end-to-end execution time for 1024 cores.



(b) Comparison of end-to-end data movement time at scales.

Fig. 6: Strong-scaling performance comparison, where the number of particles in VPIC simulation remains constant.

TABLE I: Configurations of CAMR for weak scaling tests

# of CPU Cores	1024	2048	4096	8192
CubedSphereFactory.num_cells	1024	2048	4096	8192
File size per checkpoint	43GB	170GB	680GB	2.7TB

Data Elevator, the contention on the burst buffer between I/O requests of applications and the data movement to Lustre are reduced by offloading data movement job to computing nodes. As a result, we see significantly better asynchronous data movement performance with *Data Elevator*. We observed the same pattern for the end-to-end execution time in our tests with larger number of CPU cores. Because of the page limit, we only report the end-to-end execution time for 1024 CPU cores here.

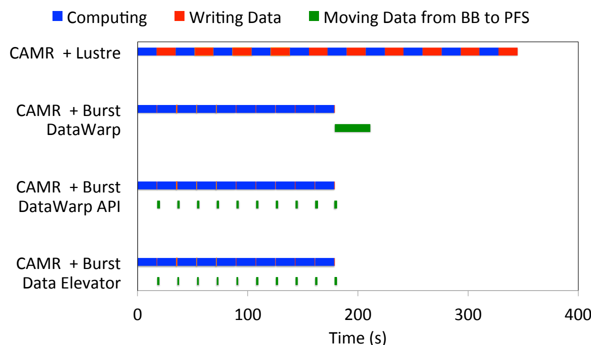
In Fig. 6b, we show the end-to-end data movement time for all 20 time steps with VPIC strong scaling. Overall, the end-to-end data movement time decreases gradually when using more cores. When the number of CPU cores is 1024, PFS takes more time than DataWarp because only 32 compute nodes write to PFS, whereas DataWarp uses 144 nodes to move data to PFS. As we increase the number of processes (i.e., nodes), the writing data to PFS becomes faster. The time for DataWarp stays the same because the number of DataWarp servers remain fixed at 144. Since *Data Elevator* used various optimizations to reduce the data movement overhead, it always performs the best at all scales. On average, using *Data Elevator* to move data for VPIC is 1.7X, 1.8X, and 4.2X faster than PFS, DataWarp Command, and DataWarp API, respectively.

2) Climate simulation: CAMR (Climate Adaptive Mesh Refinement) [16] is a code for simulating atmospheric dynamics in global climate models. The application's I/O patterns are typical for climate applications: relatively small data files are input once, and subsequent checkpoint and analysis files are output periodically during the simulation. We have run this code for 20 time steps. The checkpoint interval is set to be 2, giving 10 checkpoint files in total. We provide the details of this configurations in Table I. Since the largest scale for the number of cubed-sphere factory cells learned from domain scientist is 8192, we ran our scaling tests for CAMR using up to 8192 CPU cores.

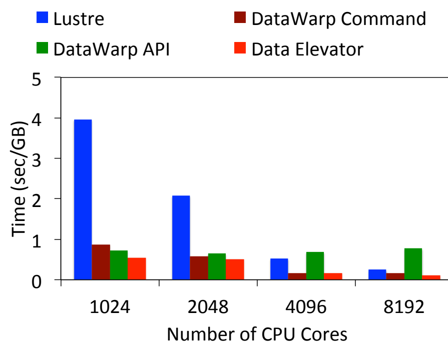
In Fig. 7, we compare performance of *Data Elevator* with scaling the problem size of CAMR as we increase the number of CPU cores. Compared with VPIC, CAMR is more computing intensive. Hence, from Fig. 7a, we can see the time between the data movement from burst buffer to PFS by DataWarp API and *Data Elevator*. Being consistent with the results from VPIC analysis, writing data to PFS directly takes the longest time. Writing data to the burst buffer for staging reduces the end-to-end execution time. Also, DataWarp command can only move data after CAMR has finished and DataWarp API and *Data Elevator* permits asynchronous data movement. The scaling performance from 1024 to 8192 CPU cores is compared in Fig. 7b. While DataWarp API is performing comparable to *Data Elevator* up to 4K cores, in the case of using 8K cores where the amount of data reaches 2.7TB per time step, the API performs slow. With 8K-core test, *Data Elevator* outperforms the DataWarp API approach by 6.7X Hence, the performance of moving the large file using DataWarp API while simulation is running is extremely poor, as we analyzed in previous sections. On the other hand, we observe that *Data Elevator* performs constantly the best at all scales. On average, the end-to-end data movement time of *Data Elevator* is 4.0X, 1.2X, 3.3X faster than that of PFS, DataWarp Command, and DataWarp API, respectively.

D. In transit analysis for VPIC simulation

One of the advantages of *Data Elevator* is the flexibility of performing in transit analysis while the data is in the burst buffer. We chose VPIC simulation since domain scientists from plasma physics perform certain runtime analysis, e.g., filtering particles with certain conditions. Here, we use a query-driven analysis, with query "Ux > 0.97", to find the targeted particles, where "Ux" is the velocity of a particle in x direction. To help post-hoc query-driven analysis, users build a certain type of index (e.g., bitmap index) in advance while the simulation is sill running [4]. Hence, we used FastQuery [7] to build the bitmap index. In addition to building the index, we also ran the above mentioned query after the index is built. FastQuery supports HDF5 data format and it can link to *Data Elevator* library to redirect its data read stream from PFS to burst buffer. But without *Data Elevator*, users need to stage their data using DataWarp onto PFS first, then build index and finally filter particles. The overhead of staging out data using DataWarp is detailedly discussed in previous sections. Here, we focus on

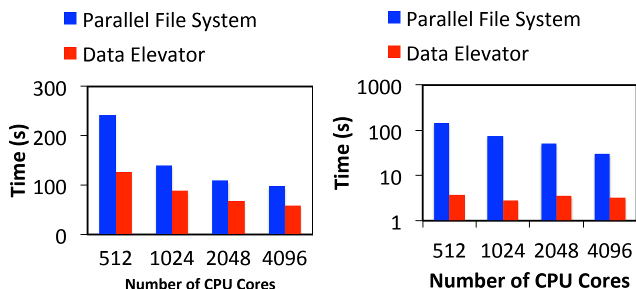


(a) A view of the end-to-end execution time for 1024 cores.



(b) Comparison of end-to-end data movement time at scales.

Fig. 7: Scaling results of CAMR simulation as we increase the problem size with the number of processes. As the data size of CAMR increases non-linearly with respect to the number of cells and also to the number of processes (as shown in Table I), we use the movement time per gigabyte data, i.e., "sec/GB", for comparing the end-to-end data movement time.



(a) Build Index (b) Evaluate Query

Fig. 8: In transit data analysis for VPIC simulation data

the performance of building index and of running the query on the Lustre PFS and on the burst buffer using *Data Elevator*. We show that the performance using the data of a single step of VPIC in Fig. 8. It is clear that using *Data Elevator* to build the index and to evaluate a query on the burst buffer via *Data Elevator* is faster than accessing data from PFS. Specifically, building index on burst buffer via *Data Elevator* is 1.9X faster than building index from the staged out file on PFS. Filtering the particles on burst buffer via *Data Elevator* is 6.6X faster than filtering the particles after they are stored on PFS.

VI. Related Work

Several efforts like FlashCache [25] and CBDMA [23] explored hierarchical storage systems on a single node [17]. Building large-scale burst buffer based hierarchical storage for HPC systems and exploring its optimizations, however, started recently and there are a few related research efforts.

Liu et al. [20] studied a disk-based external storage system augmented with a tier of SSD-based burst buffers located in a set of I/O nodes on HPC. Wang et al. [29] also studied a burst buffer architecture that is attached to computing nodes via emulation. We conducted our research on a burst buffer in a production HPC environment. In our study, the burst buffer is managed as a single storage space separated from memory and disk-based parallel file system.

Wang et al. [28], [29] proposed TRIO to use a burst buffer for running I/O intensive simulations on HPC. In TRIO, the SSD devices are located close to memory and each file inside SSD is viewed as a set of blocks. Wang et al. also explored methods for quickly moving these blocks from the SSD to a PFS. Burst buffer was also explored to aggregate written data at SSD on computing nodes [21]. In contrast, our work aims at a different architecture for burst buffers where fast SSD drives are managed as single storage space and the whole file is viewed as a single unit. We also explored new file-level optimizations, e.g., overlapping and string size alignment, for efficient data movement.

DataSpaces framework [10] studied staging of data across deep memory hierarchies [15]. This work aims at using both DRAM and SSDs to support dynamic data staging for the data coupled between data producers and consumers of workflows. The authors also explored an application aware data placement in memory hierarchy to reduce the overhead of reading data from lower levels. This paper focuses on efficient data movement from burst buffer to PFS, where new optimizations are still needed. Cray DataWarp [9] is the state-of-art system for managing burst buffers. DataWarp provides user an independent and file-based storage space for applications. Meanwhile, DataWarp provides job script based commands and an API for users to move the data between the burst buffer and PFS. In this work, we analyzed the limitations of DataWarp and compared its performance with *Data Elevator* and using extensive evaluation. Tiwari et al. [27] explored a method, named Active Flash, to use computing capability embedded in SSD flash drives for performing in-situ data analysis. Other post data analysis work [11], [18] also exist. In this work, we provided a *Data Elevator* library to allow applications to use the burst buffer for *in transit* analysis and to run analysis codes on compute nodes that offer significant flexibility.

VII. Conclusions

HPC storage subsystem is going through revolutionary changes by including new storage layers such as burst buffer (BB), known as hierarchical storage system. During our first-of-a-kind analysis on a cutting-edge hierarchical storage system, we found that the data movement across different layers needs extensive user involvement and its performance is also poor because of the resource contention on BB servers. In this paper, we proposed *Data Elevator* that uses asynchronous

I/O to support transparent usage of burst buffers. To support efficient data movement, *Data Elevator* uses a new and low-contention data movement path via computing nodes. We also explored and discussed the possibility of using a few well-known techniques, such as overlapping, to reduce the overhead of data movement. Our performance evaluations show that *Data Elevator* is 35% faster than DataWarp in the data movement from the BB to the PFS at a small scale of 1024 processes. We have applied *Data Elevator* to real simulation applications from plasma physics and climate modeling domains. The end-to-end data movement time required by those applications using *Data Elevator* can be reduced by up to $4.2\times$. *Data Elevator* also supports in transit data analysis. The in transit index construction and query evaluation with *Data Elevator* is 1.9X and 6.6X faster than with PFS.

We are expanding *Data Elevator* to support data movement among more storage layers, such as node-local memory. We are also working to support prefetching or pre-staging the data from a PFS to a BB or to a node-local memory. We also plan to evaluate the energy efficiency of *Data Elevator*, as discussed in Active Flash [27].

Acknowledgment

This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231. We also thank the Burst Buffer Early User Program of NERSC. This research used the computing resources of NERSC.

References

[1] M. Adams, P. Colella, D. T. Graves, J. Johnson, N. Keen, T. J. Ligocki., D. F. Martin., P. McCorquodale, D. M. P. Schwartz, T. Sternberg, and B. V. Straalen. Chombo software package for amr applications - design document. Technical report, Lawrence Berkeley National Laboratory Technical Report LBNL-6616E, 2013.

[2] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at Los Alamos National Lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[3] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. V. Straalen, N. Wright, K. Antypas, and Prabhat. Accelerating Science with the NERSC Burst Buffer Early User Program. In *Cray User Group conference*, 2016.

[4] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *Proceedings of the SC '12*, pages 59:1–59:12, Los Alamitos, CA, USA, 2012.

[5] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, DC, USA, 2010.

[6] M. Chaarawi and Q. Koziol. HDF5 Virtual Object Layer. Technical report, Available: https://confluence.hdfgroup.uiuc.edu/display-/VOL/Virtual+Object+Layer, 2011.

[7] J. Chou, K. Wu, and Prabhat. FastQuery: A Parallel Indexing System for Scientific Data. In *CLUSTER*, pages 455–464. IEEE, 2011.

[8] P. Cicotti, J. Bennet, S. Strande, R. Sinkovits, and A. Snavely. Evaluation of I/O technologies on a flash-based I/O sub-system for hpc. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ASBD '11, pages 13–18, New York, NY, USA, 2011. ACM.

[9] Cray. DataWarp User Guide S-2558-5204. Technical report, Available: http://docs.cray.com/books/S-2558-5204/S-2558-5204.pdf, 2016.

[10] C. Docan, M. Parashar, and S. Klasky. Dataspaces: An interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, New York, NY, USA, 2010. ACM.

[11] B. Dong, S. Byna, and K. Wu. SDS: A Framework for Scientific Data Services. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 27–32, New York, NY, USA, 2013. ACM.

[12] B. Dong, X. Li, L. Xiao, and L. Ruan. A new file-specific stripe size selection method for highly concurrent data access. In *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*, GRID '12, pages 22–30, Washington, DC, USA, 2012.

[13] D. Fellinger. The State of the Lustre File System and The Lustre Development Ecosystem. Technical report, 2003.

[14] D. Hildebrand, A. Nisar, and R. Haskin. pNFS, POSIX, and MPI-IO: a tale of three semantics. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 32–36, 2009.

[15] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C. S. Chang, and M. Parashar. Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1033–1042, May 2015.

[16] H. Johansen, E. Goodfriend, D. Rosa, J. N. Johnson, N. D. Keen, P. Ullrich, and P. McCorquodale. Adaptive high-order conservative discretization for global atmospheric flows. Technical report, 2016. In preparation.

[17] T. Kgil and T. Mudge. FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 103–112, New York, NY, USA, 2006. ACM.

[18] J. Liu and Y. Chen. Fast data analysis with integrated statistical metadata in scientific datasets. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, Sept 2013.

[19] J. Liu, B. Dong, S. Byna, and K. Wu. Model-driven data layout selection for improving read performance. In *Proceedings of High Performance Data Intensive Computing*, HPDIC'14, 2014.

[20] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, USA*, pages 1–11, 2012.

[21] X. Ouyang, S. Marcarelli, and D. K. Panda. Enhancing Checkpoint Performance with Staging IO and SSD. In *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, pages 13–20, May 2010.

[22] M. Romanus, R. B. Ross, and M. Parashar. Challenges and considerations for utilizing burst buffers in high-performance computing. *CoRR*, abs/1509.05492, 2015.

[23] J. S. Introduction to the Storage Performance Development Kit (SPDK). Technical report, Intel Corporation, 2015.

[24] A. Shoshani and D. Rotem, editors. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC Press, 2010.

[25] M. Srinivasan. FlashCache: A Write Back Block Cache for Linux. Technical report, Available: https://github.com/facebook/flashcache/blob/master/doc/flashcache-doc.txt, 2011.

[26] H. Tang, X. Zou, J. Jenkins, D. A. Boyuka, S. Ranshous, D. Kimpe, S. Klasky, and N. F. Samatova. *Improving Read Performance with Online Access Pattern Analysis and Prefetching*, pages 246–257. Springer International Publishing, Cham, 2014.

[27] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 119–132, Berkeley, CA, USA, 2013. USENIX Association.

[28] T. Wang, S. Oral, M. Pritchard, K. Vasko, and W. Yu. Development of a burst buffer system for data-intensive applications. *CoRR*, abs/1505.01765, 2015.

[29] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu. Trio: Burst buffer based I/O orchestration. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 194–203, Sept 2015.