Toward Scalable and Asynchronous Object-centric Data Management for HPC

Houjun Tang[1], Suren Byna[1], François Tessier[2], Teng Wang[1], Bin Dong[1], Jingqing Mu[3],
Quincey Koziol[1], Jerome Soumagne[3], Venkatram Vishwanath[2], Jialin Liu[1], Richard Warren[3]

[1] Lawrence Berkeley National Laboratory, [2] Argonne National Laboratory, [3] The HDF Group

***Abstract*—Emerging high performance computing (HPC) systems are expected to be deployed with an unprecedented level of complexity due to a deep system memory and storage hierarchy. Efficient and scalable methods of data management and movement through this hierarchy is critical for scientific applications using exascale systems. Moving toward new paradigms for scalable I/O in the extreme-scale era, we introduce novel object-centric data abstractions and storage mechanisms that take advantage of the deep storage hierarchy, named Proactive Data Containers (PDC). In this paper, we formulate object-centric PDCs and their mappings in different levels of the storage hierarchy. PDC adopts a client-server architecture with a set of servers managing data movement across storage layers.**

To demonstrate the effectiveness of the proposed PDC system, we have measured performance of benchmarks and I/O kernels from scientific simulation and analysis applications using PDC programming interface, and compared the results with existing highly tuned I/O libraries. Using asynchronous I/O along with data and metadata optimizations, PDC demonstrates up to *23*$\times$ speedup over HDF5 and PLFS in writing and reading data from a plasma physics simulation. PDC achieves comparable performance with HDF5 and PLFS in reading and writing data of a single timestep at small scale, and outperforms them at a scale of larger than ten thousand cores. In contrast to existing storage systems, PDC offers user-space data management with the flexibility to allocate the number of PDC servers depending on the workload.

I. Introduction

Transformative changes in scientific data management strategies are needed to handle the complexity of multi-layer storage hierarchy in exascale architectures and the imminent deluge of scientific data and metadata. As HPC is rapidly moving towards exascale, new architectures are equipped with multiple layers of storage, such as storage class memory or NVRAM on compute nodes, SSD-based burst buffers shared by compute nodes, disk, or tape-based storage systems. HPC systems being installed at several Department of Energy facilities are already designed with these storage layers. For instance, the current flagship system at the National Energy Research Scientific Computing Center (NERSC), called Cori, has an SSD-based burst buffer and a disk-based Lustre parallel file system. On the other hand, scientific data is continuously growing. For example, plasma physics simulations studying solar weather already produce hundreds of terabytes of data [1]. Similarly, experimental and observational facilities, such as Linac Coherent Light Source (LCLS), are projected to generate several petabytes of data. In addition, scientific data also comes with rich metadata, which needs to be managed in a scalable manner to allow efficient information retrieval. In order to tackle these issues, a transformative change in managing scientific data on HPC systems is critical.

Traditional parallel file systems, such as Lustre [2] and GPFS [3], use striping for placing data over a multitude of storage servers to facilitate concurrent data access. However, their adherence to POSIX constraints limits their scalability. They are also faced with serious challenges in handling the upcoming architectural changes as well as the massive amount of data. For instance, parallel file systems are typically unaware of the multi-level storage hierarchy and need external middleware to glue various layers together. Another critical deficiency is metadata management, where files are associated with limited prescriptive metadata and understanding the metadata is left entirely to the user.

The desire to overcome the limitations of current parallel file systems has spawned several efforts which explore object-based storage. For instance, RADOS [4], Amazon S3 [5], and OpenStack Swift [6] have been developed for managing data as objects and storing them in a flat namespace. While these technologies work well in distributed cloud environments, where the whole objects are accessed each time, HPC applications have unique requirements such as accessing the data in small subsets. MarFS [7] attempts to bring a cloud-style file system to HPC environment with a near-POSIX storage layer. Limitations of these systems include the lack of asynchronous and transparent data movement, lack of support for extensible metadata describing information in data, and a programming interface for scalable object management. DAOS [8] is an object-based file system solution currently under development for, it provides asynchronous data movement and manages objects in a hierarchical storage with multiple layers.

To address shortcomings of existing parallel file systems and object storage systems, i.e., asynchronous data movement, extensible user-defined data, and scalable metadata management, and to include an improved *object-centric* data management interface, we introduce the Proactive Data Containers (PDC) system in this paper. In the PDC system, objects are first-class citizens, where applications can map their data structures, such as multi-dimensional arrays, as

objects. The PDC system services run in user-space, co-located with applications in a scalable manner, and provide asynchronous data movement across multiple layers of storage hierarchy.

PDC manages metadata and data objects separately. Metadata contains predefined information about a data object, such as name, ownership, timestamps, etc., as well as unlimited number of user-defined tags as rich metadata. We provide a programming interface for creating, updating, and querying metadata to find target data objects. A challenge that many object-centric file systems face is a performance bottleneck in accessing the metadata servers to locate the data files. Moreover, each layer of storage has a different file namespace. To overcome these challenges, we dedicate a small fraction of compute resources in user-space for running the metadata services separately from data management services. Because computing resources are generally available in abundance in HPC systems, using a fraction of these resources for I/O services has minor impact on applications. For example, in a compute node containing 32 CPU cores, we use 1 core to run PDC server. All servers together manage an integrated namespace, with objects stored in one or more storage layers.

To manage data objects, we enabled PDC with asynchronous I/O and have adopted various optimizations. For example, we use a node-local data aggregation approach for avoiding a large number of small data movement requests. PDC also dynamically selects storage devices in the hierarchy for storing data. Applications can define regions of data objects and access individual regions without moving the entire object. Overall, the novel contributions of this paper are the following:

- An object-centric data model and programming interface for managing data objects on HPC systems.
- A user-space, distributed service for asynchronous and transparent data movement across a storage hierarchy.
- Integration and optimization of a scalable, distributed, and in-memory metadata management service into PDC.
- Storage layer-aware optimizations such as node-local data aggregation and automatic Lustre tuning to reduce data access latency.

We have evaluated the PDC system production supercomputing systems located at the National Energy Research Scientific Computing Center (NERSC) and at the Argonne Leadership Computing Facility (ALCF). We have measured performance of I/O kernels extracted from real-world scientific applications. We show that using PDC achieves a multi-fold speed up compared to highly tuned existing I/O libraries such as HDF5 and PLFS.

The remainder of the paper is organized as follows: We introduce the Proactive Data Containers (PDC) system in Section II. In Section III, we present our experimental setup used in evaluating PDC and in Section IV, we analyze the scaling behavior of PDC and performance of a plasma physics simulation I/O kernel. We discuss literature relevant to PDC in Section V and conclude the paper in Section VI.

II. Proactive Data Containers System

Current I/O standards, such as POSIX-IO and MPI-IO, present fundamental challenges in the areas of semantic-based data movement optimizations, asynchronous operations, scalable metadata operations, and scalable support for consistent distributed operations. Object-centric mechanisms have been proposed but the existing approaches have not yet been able to express data structures that can transcend through all the layers of the memory and storage hierarchy.

We introduce Proactive Data Containers (PDCs), whose principal goal is to provide users with an efficient and scalable scientific data management system for the upcoming exascale storage systems. PDC starts with moving away from the existing file-oriented approaches, and explores novel, object-centric data management approaches. A PDC is a container of objects, where the objects are managed by PDC services and are placed in any level of the storage hierarchy (i.e., NVRAM, disk, tape, etc.) as shown in Figure 1. It provides efficient data movement operations in critical areas of the exascale data management software stack. With the PDC system managing data objects and their placement in the storage hierarchy transparently, users are relieved from the burden of managing files on their own. Data transformations according to future use of the data or analysis in the data path, if enabled, may take place 'proactively' while the data is in a container. In this paper, we focus mainly on the data I/O services, without the emphasis on proactive analysis.

Our implementation of PDC adopts a client-server approach to monitor and manage I/O operations on objects.

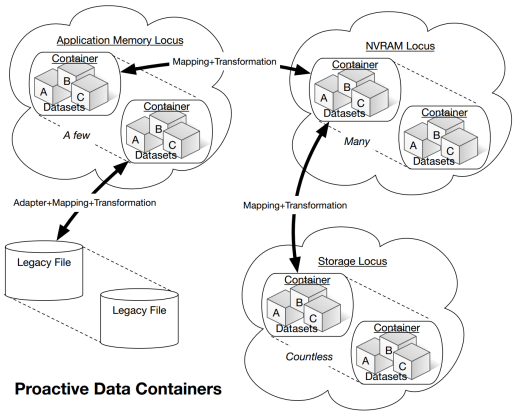

Figure 1. Proactive Data Containers can reside at any level of the storage hierarchy, and access data as objects or in legacy files. PDC and objects within them can be mapped to one another temporarily or permanently, and have transformations occur during I/O.

With this approach, PDC servers are able to asynchronously handle the I/O operations in the background while a client application continues its computation, thus exploiting the compute and storage resource strengths of each location. PDC APIs give applications the ability to create containers and place objects at various memory/storage locations along with defining object mappings and data transformations.

A. Object-centric Programming Interface

The main data constructs of PDC include *Containers*, *Objects*, *Regions*, and *Properties*. A **Container** is a collection of objects that share similar attributes defined by a user (e.g., all data variables produced by a simulation or an experiment). An **Object** is a generic term to describe byte streams in an abstract manner. In PDC, data objects can be *arrays* or *key-value* pairs. PDC also uses the concept of spatial **Regions** that address the common approach in most scientific applications that partition the problem domain into smaller sub-regions. Each region contains the actual data as well as metadata associated with it. A region is the basic unit for expressing data movement operations in PDC. These three entities also include **Properties**, or metadata, which contain the descriptive information set by the user, or generated automatically by PDC.

```
1  // Create a PDC and set container properties.
2  pdc_id = PDC_init("PDC");
3  cont_prop = PDCprop_create(CONT_CREATE, pdc_id);
4  cont_id = PDCcont_create("VPIC",cont_prop);
5
6  // Create and set object properties
7  obj_prop = PDCprop_create(OBJ_CREATE, pdc_id);
8  PDCprop_set_obj_dims(obj_prop,ndim,dims);
9  PDCprop_set_obj_type(obj_prop,PDC_FLOAT);
10 // Set tags and map data buffer
11 PDCprop_set_obj_tags(obj_prop, "nparticles=8M, nvar
       =8,ts=1");
12 PDCprop_set_obj_buf(obj_prop,&my_data[0]);
13
14 // Create an object
15 obj_id = PDCobj_create(cont_id,obj_name,obj_prop);
16
17 // Write the object to storage
18 PDC_region_info_t my_region;
19 my_region.ndim      = ndim;
20 my_region.offset    = myoffset;
21 my_region.size      = mysize;
22 PDC_Client_write(obj_id,my_region);
23
24 // Query object with specific properties
25 PDC_query(obj_name,query_prop,&n_obj,&obj_ids);
26
27 // Read the data of objects returned from query
28 for (i=0; i<n_obj; i++)
29    PDC_Client_read(obj_ids[i],my_region);
30
31 // Close object and PDC
32 PDCobj_close(obj_id)
33 PDC_close(pdc_id)
```

Figure 2. An example of the current PDC API for creating, writing, and querying objects.

The PDC system allows an application to describe its memory buffers in an object-oriented and self-describing way. Typical representations include structures of arrays (SOA) and arrays of structures (AOS). In Figure 2, we show an example of using the PDC API for creating a container (line 4), creating an object (line 15), and setting their corresponding metadata (line 3 and 7 to 12). Line 25 to 29 show an example of querying an object and reading the contents of a region in an object. It also includes the I/O and metadata query routines.

B. User-space Client-server Architecture

PDC is implemented as a user-space framework using a client-server architecture, as shown in Figure 3. PDC servers are responsible for both metadata and data management operations. There are two modes that a user can run the PDC servers in. In **shared mode** the PDC server processes are co-located on the compute nodes alongside an application's processes and share the CPU and the memory resources. In **dedicated mode**, the PDC server processes run on dedicated nodes that are separate from the nodes allocated to run an application. We focus on the former to allow PDC to utilize shared memory for more efficient data movement between node-local clients and servers. A user can start any number of PDC servers suitable for their workload. Typically we run one PDC server using one core per compute node, while the remaining cores on the node are used to run an application.

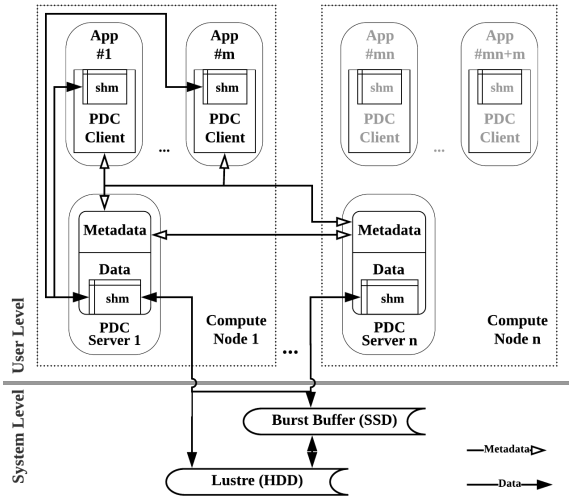

Figure 3. PDC client-server architecture overview. In this figure we show that each compute node runs 1 PDC server, and m client application processes, which is a typical PDC configuration. The application issues data related requests through the client API. Both the clients and the server manage their own shared memory segments for fast inter-process data transfer within the same node. Additionally, only the servers interact with the storage system for data movement, and the clients access data through its node-local server. For metadata related operations, as the metadata for a specific object may be located on another server process, the clients may need to contact remote servers to access it.

PDC clients provide communication between an application and the PDC servers. The client is implemented as a standard library and provides a set of data and metadata operation APIs. The client-server communication is implemented using Mercury [9], which implements Remote Procedure Calls (RPCs) optimized for HPC systems. In our experiments, we have configured the Mercury library to use the BMI plug-in (the communication component of OrangeFS [10]) with the TCP protocol. This can be replaced with faster protocols that can make use of RDMA, such as OFI libfabric over Cray GNI.

C. Namespace Management

PDC separates data and metadata objects, and allows applications to locate the data objects by accessing metadata. Both data and metadata objects are managed in a *flat namespace*, as opposed to the hierarchical directory structure used by existing file systems. The traditional hierarchical namespace requires that users keep a record (cache) of the long directory path for performance reasons to access files. Locating a file for the first time requires a traversal of the directory tree from the root. On the other hand, a flat namespace has all metadata objects on the same level. Each metadata object can be located directly and independently.

We have integrated the Scalable Object-centric Metadata (SoMeta) framework [11] into the PDC system for managing metadata objects. SoMeta offers scalable metadata operations such as creating, querying, and updating metadata objects. It provides a tagging approach to enable a logical organization (such as grouping by directory) for all metadata, and provides efficient querying that allows users to locate a specific metadata object or in finding related metadata objects with high efficiency and scalability. More details of SoMeta implementation are available in [11].

D. Asynchronous I/O

Asynchronous I/O provides the ability to overlap computation with I/O operations. An application can issue an I/O request and perform other computations until it gets asynchronously notified without waiting for its completion. For scientific applications that involve large amounts of I/O, overlapping computation and I/O can offer significant improvement of throughput via read-ahead (prefetch) or write-behind operations. In contrast, with synchronous I/O, an application is blocked awaiting the completion of I/O operations.

We enable asynchronous I/O with the PDC's client-server architecture to allow all I/O operations to be asynchronous by default. If synchronous I/O is needed, it is implemented as asynchronous I/O followed immediately with a status check and wait. In Figure 4, we illustrate the procedures to perform asynchronous read and write operations. The application issues an I/O request that is sent to the node-local PDC server. In case of a write request, the client creates

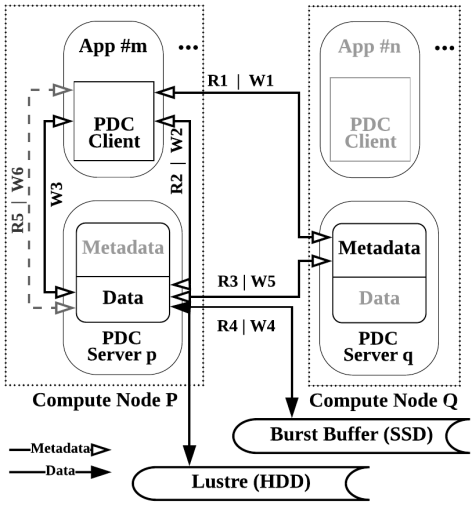

Figure 4. Flow of PDC I/O operation – R1: Query object and get metadata; R2: Init and send read request, proceed to next task; R3: Get storage metadata; R4: Read from storage devices; R5: Get server's shared memory information and copy data to user's buffer; W1: Create object and regions; W2: Init and send write requests, proceed to next task; W3: Open shared memory from client; W4: Write to storage devices from clients' shared memory; W5: Update storage metadata; W6: Confirm data is written.

a shared memory segment, copies the data that are to be written to storage, and then sends the related information along with the write request to the server. It can then reuse the original data's buffer and proceed to its next task without waiting for the I/O to finish. Upon receiving the request, the server enqueues the request to its I/O queue and responds with a confirmation message to the client immediately. When the client needs to confirm that the previously sent I/O request is completed by the server, it can send another request with the identifying information to the server. The server responds with the status of the I/O request, which can be "finished", "in progress", or "failed". In case of read requests, the server creates a shared memory segment, reads the requested data into it and sends the address to the client. The client uses the address to open the shared memory and copies the data to the user-specified buffer.

E. Storage Hierarchy-Aware Data Management

Scalable data movement involves efficient movement of data concurrently from a set of 'M' clients to a set of 'N' PDC servers and eventually to persistent storage systems. It takes into account the underlying system interconnect topology, storage system characteristics, and the application communication patterns. Data movement in PDCs can be initiated by the application explicitly accessing objects located on the various storage layers, or by using object mapping and projection operations. Users can inform PDC of which storage devices are available through the use of environment variables (e.g., the location of a burst buffer that a user has access to). PDC will then use these storage resources at runtime and pro-actively moves data as required.

PDC is aware of the available deep memory hierarchy through the library configuration and/or runtime environment variable set up. Many HPC systems, such as the Cori supercomputer at NERSC, have burst buffers that are SSD-based storage devices and sit between main memory and hard disk-based storage systems. They provide high throughput I/O alternatives for short-term "out-of-core" storage.

PDC provides two ways to move data across the memory hierarchy: the first is through *automatic data movement*; and the second is through *user-directed data movement*. For automatic movement, the PDC system initiates data transfers based on the knowledge accumulated by the server, including the characteristics of different storage devices, storage capacity, the size of the data, etc, to achieve high throughput. Another option is designed for advanced users with storage optimization knowledge. In this case, PDC allows users to set metadata hints to tag data regions with their desired storage location.

F. Optimizations for scalability

We have designed various optimizations to improve performance of data and metadata management in the PDC system.

1) Data management optimizations: **Node-local data aggregation.** PDC servers accumulate the I/O requests from the node-local clients and perform them consecutively at a later time. In our current implementation, data is moved from clients to the server (and vice versa) co-located on the same node using shared memory. For write operations, each server writes to an individual file and data from different node-local clients are written contiguously, in append only log-structured fashion. This approach reduces the potential file system metadata overhead with file-per-process mode and we have observed performance improvement over the single-shared-file approach at large scale. When reading data, PDC servers first retrieve the corresponding storage metadata to obtain the file location and offset information. The server then reorders read requests from the clients so that the actual reads to the storage devices can be performed as contiguous as possible. This strategy achieves significantly better performance than non-contiguous random accesses.

Automatic Lustre Tuning. The Lustre file system is widely used in HPC centers and provides high performance I/O services for applications to access massive amount of data. It is composed of a collection of I/O servers and disks called Object Storage Servers (OSSs) and Object Storage Targets (OSTs). File striping is a well known approach to increase I/O performance by writing or reading to/from multiple OSTs simultaneously, thus increasing the available I/O bandwidth to an application. Selecting the best striping strategy however, can be complicated for users that do not have experience with I/O optimization. Striping a file over too few OSTs will likely under-utilize the system's available bandwidth. Similarly, striping over too many will cause unnecessary overhead and lead to a loss in performance. Even for I/O experts, setting ideal Lustre stripe parameters and matching them with the MPI-IO hints can be troublesome. At a minimum, it requires a lot of manual work and subsequent application or data changes will require that these parameters be adjusted accordingly.

PDC relieves users of such burden and hides the complexity in storing data. As PDC takes full control of the actual I/O operations, we enabled PDC to select the optimal Lustre striping strategy dynamically based on the actual workload at run time. Our key strategy for storing data on Lustre is to use as many OSTs as possible for concurrent access, and have each OST accessed by as few writers as possible to reduce contention.

Assume N_{OST} is the total number of available OSTs, N_{Writer} is the number of writers, and $Rank$ is the server's MPI rank. We use the following equations for selecting the Lustre stripe count, stripe size, and the stripe offset.

$$StripeCount = \lceil N_{OST}/N_{Writer} \rceil \tag{1}$$

$$StripeSize = \begin{cases} 1MB & \text{if data size} \leq 1GB \\ 16MB & \text{otherwise} \end{cases} \tag{2}$$

$$OSTIndex = (Rank \times StripeCount)\%N_{OST} \tag{3}$$

2) Metadata management optimizations: In PDC, metadata objects are managed and distributed among the servers. Metadata operations, such as create, update, and query can be performed efficiently and in a scalable fashion, as described in our previous work [11]. However, as PDC servers are responsible for both metadata and data operations, it is crucial to orchestrate the two types of operations and minimize the overall overhead.

Based on our previous experiments, we have identified two aspects for improving the performance of metadata management in PDC. The first one is with collective metadata querying. As explained in previous sections, application processes (PDC clients) access data through metadata queries that retrieve or update the location of the corresponding data. In the case when a large number of client processes operate on a single object, they all need to get the metadata from a single metadata server. Even with our millisecond level communication overhead, when the number of concurrent requests become very high, the cumulative overhead could be non-negligible. To reduce this overhead, PDC aggregates the requests from client processes on the same compute node, and sends them to the target metadata server in bulk. Once results are returned, the server will distribute the corresponding data to the clients. With this optimization, the number of concurrent requests were significantly reduced.

Another metadata optimization used in PDC is to reorder metadata operations for a given sequence of data and metadata requests. To maintain consistency of data and metadata, PDC must keep an up-to-date records of all data it

manages (i.e., whenever there is a change with the data, the corresponding metadata must be updated). However, in some cases, the user may not need strict consistency but rather eventual consistency in exchange for better performance. One such scenario is when a scientific simulation is producing multiple variables for a certain number of timesteps. When data is being written, as each server sends its latest storage metadata update to the metadata server for each variable, the metadata server may also be performing I/O operations at the same time for another variable. As a result, the metadata update is blocked and the server initiating the metadata operations has to wait before proceeding with its next task. To resolve this issue, we provide users with the option to specify a relaxed consistency requirement. Before starting the write operations, the user can provide a hint through PDC APIs and inform the server regarding the number of expected writes so that the server can postpone the metadata update until all the writes are performed. This user-controllable "lazy update" can significantly improve the throughput and reduce the time spent on metadata updates. In the future, we plan to remove this "hint" requirement with further optimizations on the server side, and let the server decide of the best time to perform such metadata updates.

III. Experimental Setup

HPC Systems	Cori (NERSC), Cooley (Argonne)
Comparison	PDC, HDF5, and PLFS
Workloads	Benchmarks IO Kernels (VPIC-IO, BDCATS-IO)
Operations	Write, read with single and multiple time steps. Strong and weak scaling
Storage	Main Memory SSD-based Burst Buffer Hard disk drive (Lustre and GPFS)

We evaluated the performance of PDC with a series of experimental configurations, as shown in Table III. We run PDC systems on the Cori system at the National Energy Research Scientific Computing Center (NERSC). Cori is a Cray XC40 supercomputer with 1630 Intel Xeon "Haswell" nodes, where each node consists of 32 cores and 128GB memory. Its Lustre storage system is shared by all users and has 248 OSTs, and the burst buffer resources are allocated based on the specific request in the job submission script. We ran one PDC server on each compute node in our tests that share resources with user applications, that is, the PDC server occupies one core on each compute node, and the user's application runs on the remaining 31 cores. We also tested the VPIC-IO kernel on Cooley, an analysis and visualization cluster at Argonne Laboratory Computing Facility (ALCF). Cooley has 126 Intel Haswell E5-2620 nodes, each with 12 cores and 384GB of memory. The 27 PB of storage are managed with a GPFS file system.

We have used a set of benchmarks and I/O kernels representative of scientific simulation and analysis codes.

We developed a write benchmark that creates a single PDC object across all client processes with each process generating data for a non-overlapping sub-region of data. The read benchmark queries and then retrieves the metadata generated by the write benchmark. Each MPI process then issues requests to read a specific non-overlapping region of the global object via the PDC interface.

We used the VPIC-IO kernel, which is extracted from a plasma physics code called VPIC [12], which simulates magnetic reconnection phenomenon in space weather. In this kernel, each MPI process writes 8M (8×2^{20}) particles with each particle having 8 variables. VPIC data structures use 1-D arrays to represent each variable. The total length of each property array is equal to $n \times 8 \times 2^{20}$, where n is the number of MPI processes. BD-CATS-IO kernel is extracted from a parallel clustering algorithm, used for analyzing the data produced by particle simulations, such as VPIC. In this kernel, data related to the particles are read among all the MPI processes in a load-balanced distribution. While these kernels use random data, the I/O patterns exactly match that of the simulation and analysis. The original kernels use HDF5 for performing I/O and are highly tuned using MPI-IO and Lustre optimizations [1], [13]. In this paper, we have also implemented these kernels using PLFS and PDC objects. For PDC, each 1-D array is mapped as a PDC object, and each process works on a region of the object. The region is a 1-D array of 8M particles.

For all the results presented below, we have measured the elapsed time, which is the end-to-end time from the first I/O operation until the last I/O operation finishes. For PDC, the elapsed time includes the overhead of using PDC to both maintain the metadata and move the data between memory hierarchies. For both HDF5 and PLFS, it includes the file open and close times, along with the time for moving data from memory to a file system. For all the evaluations presented in this paper, we ran the experiments at least three times and reported numbers representing the best results.

IV. Evaluation

A. Strong scaling

In the strong scaling tests, we have fixed the total amount of data to be written as 512GB, and increased the number of client application processes. The number of PDC servers also increases with a fixed ratio to the number of clients as mentioned previously. In Figure 5, we report the total time taken for writing and reading a fixed amount of data (512GB) to Lustre with different number of clients. We can observe that PDC performs close to linear scalability, and the overhead is only a small fraction of the total I/O time. At the time of testing, the read performance was consistently worse than the write performance on this HPC system. Despite that slow read performance, PDC read operations scale linearly.

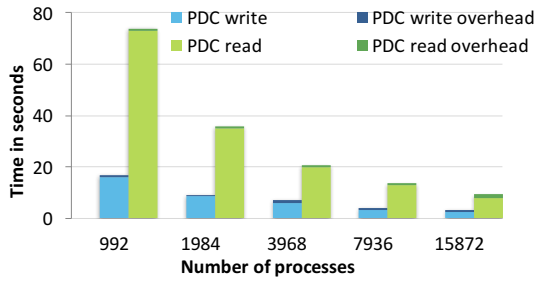

Figure 5. PDC strong scaling performance for writing and reading $512GB$ data on Lustre.

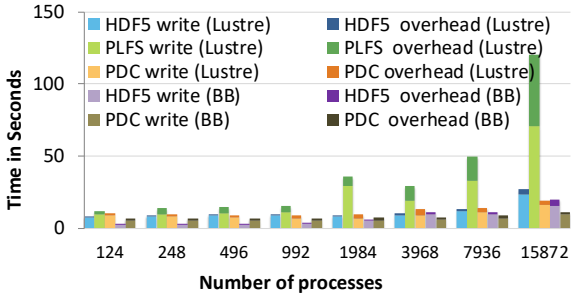

Figure 6. Total time for writing a single timestep from the VPIC-IO kernel to Lustre and Burst Buffer using HDF5, PLFS, and PDC on Cori. The data size increases with the number of client processes.

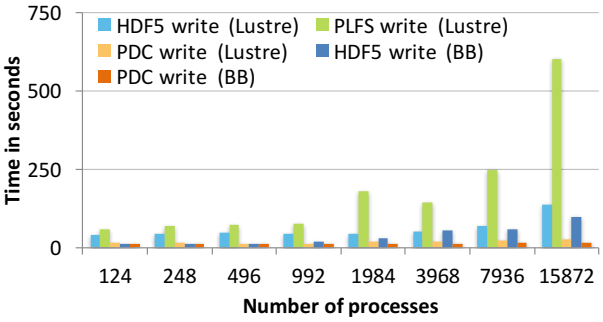

Figure 7. Total time for writing 5 timesteps from the VPIC-IO kernel to the Lustre and Burst Buffer on Cori. The data size increases with the number of client processes.

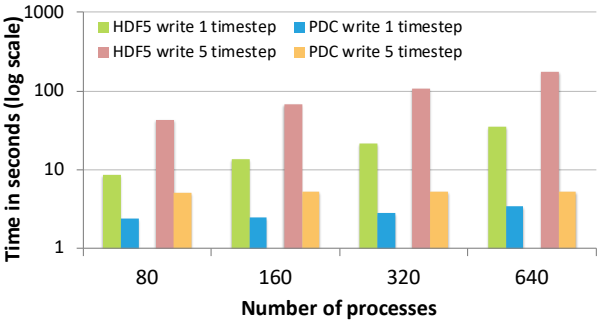

Figure 8. Total time including overhead for writing a single timestep and 5 timesteps from the VPIC-IO kernel to the GPFS file system on Cooley. The data size increases with the number of client processes. The y-axis is on logarithmic scale.

B. *VPIC-IO kernel – Weak scaling writes*

We ran the VPIC-IO kernel in two modes: single timestep and multiple timesteps. It is typical for simulation codes to run for a large number of timesteps to represent progression of real world scenario. These simulations write data for a timestep periodically. In the single timestep case in HDF5, we have written data to a single file, similar to the behavior of the original kernel. In the multi-timestep case of VPIC-IO, we have added a sleep time, representing computations between writing data. When running VPIC-IO with different number of timesteps, it is essentially a weak scaling test, with $256MB$ data written per client process.

Figure 6 compares the total write time to Lustre and to the burst buffer as observed by the client application. The breakdown of the actual I/O time as well as the overhead are also shown. For HDF5 and PLFS, all processes are writing/reading the same file(s) with 8 datasets. We used HDF5 with MPI-IO and Lustre optimizations such as avoiding file truncate and choosing optimal number of Lustre OSTs as suggested in previous research [14]. While for PDC, the number of resulting files on storage equals to the number of PDC servers multiplied by the number of objects (8). PDC outperforms PLFS in all cases and the speedup increases with the number of processes. When compared with HDF5, PDC is advantageous when running at larger scale due to server-side aggregation (e.g. PDC has a $1.4\times$ speedup over HDF5 with 15872 processes). PDC shows a more stable performance with relatively smaller performance variance when the number of clients increases. On the other hand, HDF5's overhead increases significantly. Such results demonstrates the better scalability of PDC.

Figure 7 compares the performance between HDF5 and PDC writing 5 timesteps data to Lustre and burst buffer on Cori. With PDC's support of asynchronous I/O, the client application only observes the last write time (others overlap with the computation time) plus PDC's overhead (negligible compared with the write time), as opposed to all 5-timestep's write time for HDF5 and PLFS. PDC achieves up to $5\times$ speedup over HDF5 and $23\times$ over PLFS.

Figure 8 presents the results obtained with PDC and HDF5 on the Cooley system using GPFS with a single timestep and 5 timesteps. On this platform, we notice an increasing gap in terms of performance between the two approaches. On 64 nodes (640 clients and 64 servers), the PDC implementation performs more than $7\times$ faster to store data compared to HDF5 on a single timestep while this factor reaches 35 with 5 timesteps.

C. *BD-CATS-IO – Weak scaling reads*

BD-CATS-IO is a read-intensive I/O kernel from the BD-CATS clustering system [15], which reads data generated by VPIC or VPIC-IO using the same I/O trace as the BD-

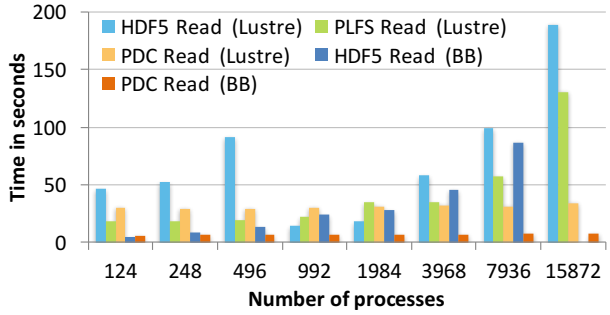

Figure 9. Total time for reading a single timestep data using the BD-CATS-IO kernel using HDF5, PLFS, and PDC. The data size increases with the number of MPI processes.

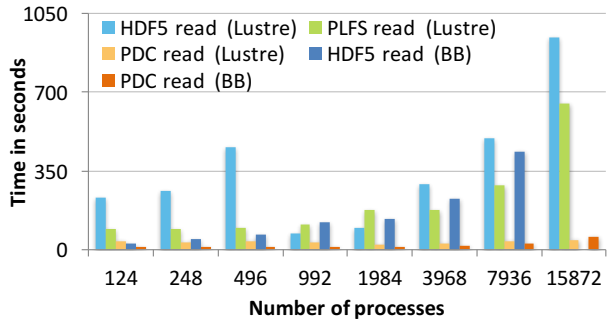

Figure 10. Total time for reading data of 5 timesteps from the BD-CATS-IO kernel from the Lustre and from the burst buffer. The data size increases with the number of MPI processes.

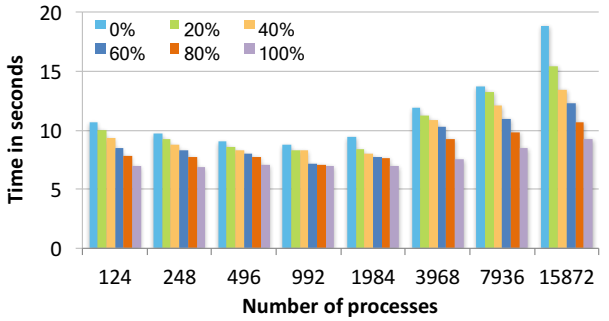

Figure 11. Write time with part of the data written to faster burst buffer and the remaining to slower Lustre file system on Cori. The percentage numbers represent the amount of data written to burst buffer.

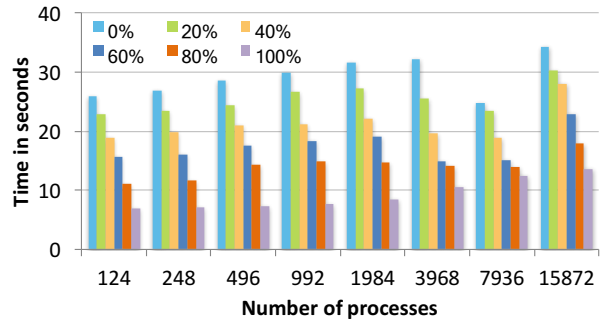

Figure 12. Read time with part of the data from the faster burst buffer and the remaining from the slower Lustre file system on Cori. The percentage numbers represent the amount of data read from the burst buffer.

CATS implementation of the DBSCAN algorithm. In Figure 9, we show the performance of reading a single timestep of data that were written in previous VPIC experiments. Similar to the write experiment results, PDC demonstrates better scalability; it offers comparable read performance with fewer clients, but outperforms HDF5 and PLFS by up to $5\times$ and $4\times$, respectively, when there are more than 4K clients.

In Figure 10, we compare the total time to read 5 timesteps of VPIC data using HDF5, PLFS, and PDC with asynchronous read optimizations. Similar to the multi-timestep write results, PDC demonstrates superior performance in all cases with up to $11\times$ speedup. We experienced system errors with the HDF5 read on burst buffer with 15872 clients, thus there is one missing value in the above two figures.

D. Using Multi-level Storage

It is obvious that the SSD-based burst buffer (BB) provides better I/O performance than the HDD-based Lustre. However, typically the total capacity of burst buffer is much less than that of Lustre. For example, NERSC's Cori supercomputer has $28PB$ of total Lustre storage disk space, but only $1.8PB$ of burst buffer, and each user can only request $50TB$ burst buffer allocation. To better utilize the burst buffer and achieve high I/O throughput, PDC can distribute the data between burst buffer SSDs and Lustre. Users can provide hints about which regions to be stored on faster burst buffers and which to be on Lustre.

Figures 11 and 12 show the write and read performance, respectively, with different distributions of data on burst buffer and on Lustre. We use a percentage of data regions on BB and the remaining on Lustre. We observe that the I/O time decreases as more data is stored on BB. Based on the available capacity, PDC provides users the flexibility to use storage hierarchies efficiently and transparently.

E. Spatial Data Access

Many scientific datasets are written once and read many times for post-processing. Analysis codes typically access a small region of large datasets to analyze, where a spatial region is specified by users. PDC supports efficient spatial region selection by merging any small regions based on their locality on storage devices. Figures 13 and 14 show the elapsed time for different number of clients reading with different selectivity, ranges from 5% to 20%, on Lustre and burst buffer. The amount of data accessed increased with the number of clients. The time in accessing the data is kept low even at 20% selectivity.

V. Related Work

Most file systems for both single and multiple node computing systems were designed to comply with POSIX-

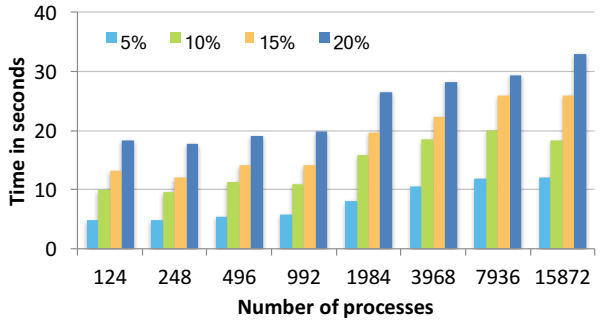

Figure 13. Time for reading various selected object regions specified by the client processes from Lustre on Cori.

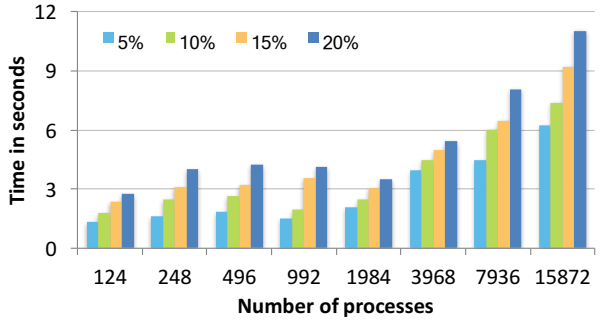

Figure 14. Time for reading various selected object regions specified by the client processes from burst buffer on Cori.

IO, which is part of the POSIX standard [16] back in the late 1980s, and defines the file access API, data model, and data consistency semantics. In POSIX-IO, data are usually viewed as byte streams. POSIX-compliant file systems includes Lustre [2], PVFS [17], GPFS [3], NFS [18], etc. POSIX-IO was not expected to work with highly concurrent programming models based on distributed memory or a combination of distributed and shared memory that are now the norm for HPC systems [19]. It is often regarded as a major limitation to scalable I/O performance. Deep memory and storage hierarchies in exascale systems [20], [21] will exacerbate POSIX-IO problems further.

The FastForward I/O project proposed DAOS, which provides database-like transactions [8]. RADOS [4], as part of Ceph [22], is a scalable and reliable object storage service for petabyte-scale storage clusters. The conventional parallel I/O stack consists of high-level libraries (HDF5 [23], NetCDF [24], ADIOS [25], etc.), I/O middleware (MPI-IO [26], TAPIOCA [27]), and I/O forwarding layer [28]. Several research efforts have focused on relaxing the POSIX semantics and on defining new data models in these layers. MPI-IO [26] guarantees that the non-overlapping or non-concurrent write operations will be handled correctly and changes to the data are immediately visible only to the writing process itself. TAPIOCA proposes to optimize collective I/O operations with topology-aware data aggregation and I/O scheduling. The HDF5, NetCDF, and ADIOS libraries provide an array-based data model to organize the data in semantic manner. The I/O forwarding layer can help aggregating I/O requests while they are transferred from client nodes to storage nodes. SDS provides methods to automatically reorganize data on disk for performance optimizations.

"Object-based storage" [29] is a generic term used to describe an abstract data container that consists of many byte-streams (or *objects*), each with related attributes. As the attributes are stored and transferred with the objects, object-based storage can efficiently express quality-of-service, transparent performance optimizations, data sharing, and data security qualities that a storage system can exploit. Research efforts to implement object-based storage on disks, in NVRAM, and in memory have been attempted separately, but none has integrated those efforts across the entire memory hierarchy.

NVRAM and various implementations of flash memory storage are portrayed as solutions to alleviate I/O performance issues of HPC systems. [30] proposes SSDUP that redirects data write to burst buffer when it detects random access that would cause high latency if written to HDDs. The data in the burst buffer are then flushed to HDDs when the size is more than half of the burst buffer capacity. [31], [32] proposes a framework that is able to utilize the deep memory hierarchy resources to improve read performance as well as runtime data sharing. In our previous work, Data Elevator [33] provides automatic data movement across multi-levels of deep storage systems. In this paper, we describe a new and full-fledged object-centric storage system to manage deep storage hierarchy.

Despite repeated exploration of various object-based storage solutions, there is no uniform object management across all the memory and storage layers of upcoming exascale systems. Generally, existing research focuses on an individual level without considering the presence of the other layers. When data moves through the hierarchy, semantic information embedded in objects is lost, resulting in poor performance. In addition, object-oriented mechanisms for expressing data structures that can transcend through all the layers of the hierarchy are unexplored.

VI. Conclusions and Future Work

Current I/O standards, such as POSIX-IO and MPI-IO, present fundamental challenges in the areas of scalable metadata operations, semantic-based data movement optimization, support for asynchronous operation, and scalable support for consistent distributed operations. To overcome these issues, we designed and developed the prototype of Proactive Data Containers that provides an object-centric data model and programming interface, with a distributed server architecture that scales to a large number of processes. PDC provides asynchronous I/O and transparent data movement in storage hierarchy with various data and metadata

optimizations to effectively limit the overhead. Experimental results demonstrate that PDC has a multi-fold performance speedup compared to HDF5 and PLFS at scale.

Our future work includes adding more autonomous features to the PDC servers to track data access patterns and operations that are performed repeatedly to enable proactive and topology-aware data movement optimizations without explicit application intervention. We will also explore methods to intelligently select the data to be stored in various layers of memory and storage hierarchy, using user-provided intent or learning-based approaches, as well as in-transit proactive data analysis that moves the analysis operations closer to where the data is stored for improved efficiency.

ACKNOWLEDGMENT

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 (Project: Proactive Data Containers, Program manager: Dr. Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center and Argonne Leadership Computing Facility, which are DOE Office of Science User Facilities supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and DE-AC02-06CH11357.

REFERENCES

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

[2] P. J. Braam *et al.*, "The Lustre storage architecture," 2004.

[3] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[4] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters," in *PDSW*, 2007, pp. 35–44.

[5] Amazon. Amazon Web Services. Http://s3.amazonaws.com.

[6] J. Arnold, *OpenStack Swift: Using, administering, and developing for swift object storage*. O'Reilly Media, Inc., 2014.

[7] J. Inman, D. Bonnie, M. Broomfield, H.-B. Chen *et al.*, "MarFS, Version 1," LANL, Tech. Rep., 2015.

[8] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and Friends: A Proposal for an Exascale Storage System," in *Supercomputing*, 2016, pp. 50:1–50:12.

[9] J. Soumagne, D. Kimpe *et al.*, "Mercury: Enabling remote procedure call for high-performance computing," in *CLUSTER*, 2013, pp. 1–8.

[10] M. Moore *et al.*, "OrangeFS: Advancing PVFS," *FAST poster session*, 2011.

[11] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-centric Metadata Management for High Performance Computing," in *CLUSTER*, 2017, pp. 359–369.

[12] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[13] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel I/O autotuning with performance modeling," in *HPDC*, 2014, pp. 253–256.

[14] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna *et al.*, "Taming Parallel I/O Complexity with Auto-tuning," in *Supercomputing*, 2013, pp. 68:1–68:12.

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

[16] S. R. Walli, "The POSIX Family of Standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[17] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, 2000.

[18] S. Microsystems, "NFS: Network File System Protocol Specification," 1989.

[19] K. J. Barker *et al.*, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner," in *Supercomputing*, 2008.

[20] M. Jung, W. Choi, J. Shalf, and M. T. Kandemir, "Triple-A: A Non-SSD Based Autonomic All-flash Array for High Performance Storage Systems," *SIGPLAN Not.*, vol. 49, pp. 441–454, 2014.

[21] F. Schürmann and et.al., "Rebasing I/O for Scientific Computing: Leveraging Storage Class Memory in an IBM BlueGene/Q Supercomputer," in *ISC*, 2014, pp. 331–347.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *OSDI*, 2006, pp. 307–320.

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

[24] J. Li *et al.*, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *Supercomputing*, 2003, pp. 39–.

[25] Q. Liu, J. Logan, Y. Tian *et al.*, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

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

[27] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *CLUSTER*, 2017, pp. 70–80.

[28] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross, "Improving I/O Forwarding Throughput with Data Compression," in *CLUSTER*, 2011, pp. 438–445.

[29] G. A. Gibson *et al.*, "A Cost-effective, High-bandwidth Storage Architecture," *SIGPLAN Not.*, vol. 33, pp. 92–103.

[30] X. Shi, M. Li, W. Liu *et al.*, "SSDUP: A Traffic-aware SSD Burst Buffer for HPC Systems," in *ICS*, 2017.

[31] W. Zhang and H. Tang and X. Zou and S. Harenberg and Q. Liu and S. Klasky and N. F. Samatova, "Exploring Memory Hierarchy to Improve Scientific Data Read Performance," in *CLUSTER*, 2015, pp. 66–69.

[32] W. Zhang, H. Tang, S. Ranshous, S. Byna, D. F. Martn, K. Wu *et al.*, "Exploring Memory Hierarchy and Network Topology for Runtime AMR Data Sharing Across Scientific Applications," in *Big Data*, 2016, pp. 1359–1366.

[33] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *HiPC*, 2016, pp. 152–161.