Toward Transparent Data Management in Multi-layer Storage Hierarchy of HPC Systems

Bharti Wadhwa[†], Suren Byna[‡], Ali R. Butt[†]
[†]Department of Computer Science, Virginia Tech
[‡]Lawrence Berkeley National Laboratory

Email:{bharti,butta}@cs.vt.edu; SByna@lbl.gov

***Abstract*—Upcoming exascale high performance computing (HPC) systems are expected to comprise multi-tier storage hierarchy, and thus will necessitate innovative storage and I/O mechanisms. Traditional disk and block-based interfaces and file systems face severe challenges in utilizing capabilities of storage hierarchies due to the lack of hierarchy support and semantic interfaces. Object-based and semantically-rich data abstractions for scientific data management on large scale systems offer a sustainable solution to these challenges. Such data abstractions can also simplify users involvement in data movement. In this paper, we take the first steps of realizing such an object abstraction and explore storage mechanisms for these objects to enhance I/O performance, especially for scientific applications. We explore how an object-based interface can facilitate next generation scalable computing systems by presenting the mapping of data I/O from two real world HPC scientific use cases: a plasma physics simulation code (VPIC) and a cosmology simulation code (HACC). Our storage model stores data objects in different physical organizations to support data movement across layers of memory/storage hierarchy. Our implementation sclaes well to $16K$ parallel processes, and compared to the state of the art, such as MPI-IO and HDF5, our object-based data abstractions and data placement strategy in multi-level storage hierarchy achieves up to $7\times$ I/O performance improvement for scientific data.**

I. Introduction

Parallel I/O for scalable computing systems such as HPC deployments need a transformative upgrade in the era of exascale computing to support emerging systems with deep memory and storage hierarchies. Realizing efficient storage and I/O mechanisms entails designing specialized software that takes full advantage of components introduced in these hierarchies. Traditional file and block based storage systems face severe challenges to meet the requirements of scientific applications. The dependence of parallel file systems on the POSIX-IO [22] enforces strict, costly data consistency requirements and lacks semantic data abstractions. However, these features are crucial for data movement while preserving semantics, storage and retrieval of scientific applications' data, especially in deep memory and storage hierarchies. In addition, parallel I/O makes it challenging to shoehorn new interfaces, such as taking advantage of multiple layers of storage and support for analysis in the data path. In the cloud computing environments, object-based storage systems, such as Openstack Swift [3], have become quite popular. However, most of these systems have been developed to store immutable cloud data and do not consider storage of large scale HPC application data. Other object-based storage systems such as Lustre [14] and Ceph [23] do support HPC applications. High-level I/O interfaces, such as HDF5 [9], manage data arrays and attributes as objects. However, the existing file systems and high-level library solutions do not support data storage in multiple layers of storage hierarchy, and just store the data in to one layer (disk- or SSD-based storage) of the stack. Hence, there is a crucial need for an efficient data storage and I/O system for HPC, which supports the complex memory/storage hierarchy and facilitate transparent data movements among the multiple layers of exascale storage stack.

In this paper, we propose an object-based storage interface to transparently store data in upcoming HPC storage hierarchy, and to manage semantics of data. This approach provides the flexibility of storing data into multiple layers of storage hierarchy. It provides the application users the options to store data partially into different layers of storage stack, thus enhancing performance by storing part of data closer to analysis process as needed. To achieve this flexibility, our proposed storage model targets storing scientific data using an object abstraction that remains uniform across the different layers of storage. The uniformity ensures that data can be moved between layers without unnecessary translation that can loose semantic information. It also supports optimizing storage of the data and the metadata inside the object and container abstractions in the storage hierarchy. These objects can move between the layers of exascale storage stack efficiently, and provide enhanced flexibility and data consistency. The object-based abstractions can store complex data structures such as multidimensional arrays and key-value stores in a consistent way in all layers of the storage stack, enabling high scalability and efficient management.

We implement the object-based data mapping atop the MPI-IO [21] interface and provide multiple options to store the applications' data into different layers of storage system (such as SSD-based burst buffers and/or disk-based Lustre file system), according to the user requirements. We illustrate the utility of our approach by applying it to two science simulation use cases: VPIC-I/O [5], [4] and HACC-I/O [12], [2]. Specifically, this paper makes the following contributions:

1) Introduction of an object representation for mapping scientific data into object abstractions that stay uniform across storage hierachies.

2) Introduction of a data object management system and storage strategies therein, aimed at facilitating semantic-preserving scientific data movement and storage between the layers of exascale storage stack.
3) Implementation and evaluation of the proposed storage model showing significant performance improvement.

II. Background: Object-based Storage

Object-based storage [10] is a generic term used to describe an abstract data container that consists of multiple 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 (QoS), transparent performance optimizations, data sharing, and data security qualities that a storage system can exploit. Object-based storage has been implemented for disks, NVRAM, and in memory. However, existing efforts do not integrate objects across the entire memory hierarchy. We studied the design and implementation of some of the popular object storage systems, and in the following present a brief discussion to highlight the breadth of features and use cases that object-based stores offer.

Lustre: Lustre File System [14] is an object-based, parallel distributed file system. It is mainly deployed on large-scale clusters to provide high data availability, independence to physical data location, and scalability. These features make Lustre a suitable choice for general-purpose back-end file system and also for supporting various scientific applications.

Ceph: Ceph [23] is an object-oriented distributed file system with a highly scalable design that makes it suitable for various types of applications. Similar to Lustre, Ceph also separates metadata from data and distributes the metadata over multiple nodes running a software component called `OSD`.

OpenStack Swift: Swift [3] is another widely-used and popular object storage system. Swift is designed to store cloud data such as large binary blobs, image and video files, backups, analytics data, and other unstructured data in the form of objects, with high availability, durability, and scalability.

DAOS: Distributed Application Object Storage (DAOS) [13] is the storage system designed for the ongoing Fast Forward [17] project to meet the requirements of exascale computing systems. DAOS is an extremely simple object model that encapsulates entire exascale data and metadata. It replaces POSIX with transactional, shared-nothing, distributed object store known as DAOS container.

Apache Spark: Spark [1] is an in-memory data processing and cluster-computing framework mainly used for enterprise big data analytics. Spark provides in-memory computations using Resilient Distributed Dataset (RDD) [26] abstraction, which is an immutable distributed collection of objects, for increased speed and data processing over MapReduce [6].

MarFS: MarFS [11] is a recently-developed object-based file system that presents cloud-style object-storage as scalable near POSIX filesystem. MarFS is becoming popular as it provides high scalability and a POSIX interface atop cloud-style object storage. These features also make MarFS attractive for HPC applications.

We found that these storage systems provide various benefits to manage and store data such as high performance, security, and scalability. But some of these systems are mainly used for cloud computing applications and are optimized for storage of immutable data, and thus may not be directly applicable to many HPC use cases.

Lustre and Ceph have been successfully supporting HPC application data at scale, but these parallel file systems for HPC manage data as streams of bytes via POSIX I/O and thus are susceptible to limitations at exascale due to lack of inherent support for object based abstractions. They lack a uniform and consistent data interface for each layer of the storage stack. Such uniformity is desirable as it can facilitate storage and scientific application data I/O, and supports seamless data movement across hierarchies. Simple and efficient methods for data management and storage through the memory hierarchy are critical for sustaining storage systems for scientific applications at exascale.

III. Design

A. Object Definition

The typical data structures used by scientific applications to represent data are multidimensional arrays. We define an object comprising a multi-dimensional array that contains either one property or multiple properties of a physical scientific object. An object can also contain an associative array (key-value store). Each object contains metadata (object ID, name, dimension, datatype, etc.), payload (actual data of a multi-dimensional array), extended metadata (attributes or properties of the payload, locations of products of data, etc.), and provenance (time of creation, owner of an object, time and user of accesses, etc.). A high-level view of an object and its components is shown in Figure 1. Multiple objects can be packed into a container to manage them as a collection. One can expand this definition to add more metadata and to support more data structures as payload.

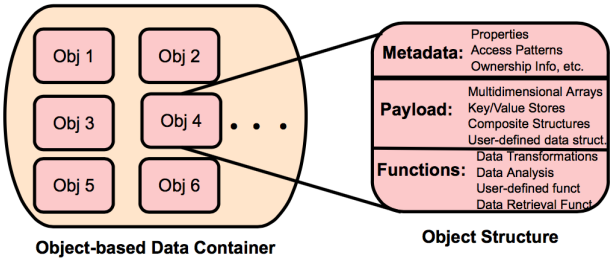

Fig. 1. The proposed object structure.

In parallel applications, objects are created in parallel on all participating processes, but have a global view of a single object. Each process creates a set of individual local sub-objects (one object per process), which are considered collectively as a subset of the global object (or object container). Once objects and containers are created, application data is mapped into them using object API, and *container-id*'s and *object-id*'s are returned to the client for future access.

Object API (provided to the application users)	
`cid=create_Container(attr, ..)`	Creates the data container with a global view which holds all the objects. Returns the container-id cid to user
`oid[i]=create_Object(cid, attr, local_size, plist, ..)`	Creates the objects for each process, with application data, semantic information and other attributes attr and properties list plist, Returns object-id oid to use
`write_Object(oid[], buf,... )`	Creates a memory buffer buf for in-memory computations, and writes objects to storage system
`finalize_container(c_id)`	Deletes the container and its component objects

TABLE I
OBJECT APIS FOR CREATING CONTAINERS/OBJECTS AND WRITING TO THE STORAGE SYSTEM.

An object can either represent a single property of a set of physical objects or a data structure containing multiple properties of the objects. We name the former as *basic objects*, and the latter as *composite objects*. Depending upon the application requirement, either basic or composite objects can be created on the compute nodes.

B. Object-based Storage Model for Multi-layer HPC Storage

In this section, we first present the object APIs that we provide to applications, then we discuss our system architecture, followed by the object storage design model for efficient objects storage.

a) Object APIs: We provide a number of object APIs to the application as listed in Table I. The APIs support object management functions such as creating containers and objects, and storing the objects into different layers of the storage system. Our design model offloads from the users to our system the task of managing MPI-IO or HDF5 calls to the separate layer. A key advantage of uniform object APIs is that data can be moved between layers without modification, which preserves semantics and enables seamless migration of data between storage hierarchy layers as needed.

b) System Architecture: Figure 2 presents the overall system architecture of our approach. The application interacts with the system at the top layer that exports the object APIs to the user. In this layer, our object model creates data containers and associated objects on the compute nodes and returns the *container-id* and *object-id*s to the user. There are n compute nodes, and each of them runs k processes, P_1 to P_k, to map application data into objects, O_1 to O_k. Each process creates one object locally in the memory of the compute node on which it executes. Another component of this layer is the object-based data container, which provides a global view of all the objects to the application.

The next component of the system is SSD-based burst buffers/nodes that offer persistent edge storage. This layer provides a closer (to compute) and faster storage system for meeting high performance needs. Objects created on the compute nodes are pushed onto this layer for short-term storage. To store the objects, our object storage model creates multiple object-shards (S_1, S_2, ..) in this layer.

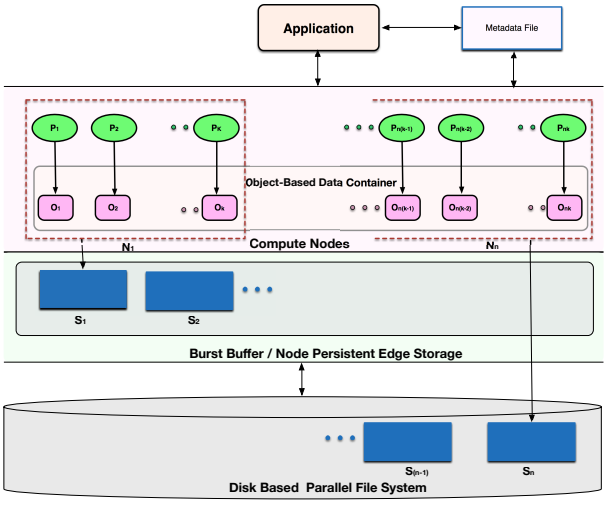

Fig. 2. High-level architecture of the proposed object storage system. P_{nk} is the k^{th} process on n^{th} compute node N_n. O_{nk} is the object created by process P_{nk}, S_n is n^{th} object-shard created by group of processes on N_n. Burst Buffers are SSD-based storage system. The parallel file system is HDD based, e.g., Lustre.

The third and the last layer of the system comprise a disk based parallel file system, e.g., Lustre. This layer provides long-term storage for objects (in the form of object-shards) that are in compute nodes or burst buffers.

c) Object-based Storage Model: Our storage model uses MPI-IO interface for mapping the data objects into files. The objects can be stored in the burst buffers, in the disk-based file system, or both according to the application requirements.

C. Object Creation and Data Mapping Process

First, as described earlier, our proposed approach creates an object-based data container that contains all the objects created on the compute nodes and set its properties such as its type, lifetime, and state. A unique *container-id* is assigned to this container, which is returned to the user after its creation and setup. Once a data container is created, each process creates a data object into this container to map the application data using the object-creation APIs listed in Table I . Based on the application requirements, either basic or composite objects are created. Next, application data is mapped into these objects using the object-mapping APIs and the *object-id*s are returned to the user. Objects sharing the same node-local memory are called sub-objects.

Once the objects are created in memory and application data is mapped into these objects, the next step is to examine the application requirements and determine if these objects need to be kept in memory, or stored into the storage layers. The storage model assigns each application a storage quota, based on which some sub-objects are stored into burst buffers and the rest are stored in Lustre. For example, if the storage quota is assigned as (25:75), 25% of all the sub-objects from the applicaiton will be pushed to burst buffer and the rest of the objects to the lower storage layer, i.e., Lustre. Our object interface contributes to seamless semantic-preserving data migration between these storage layers.

The model maps the data objects into object-shards as follows. As shown in Figure 2, in the top layer, the model divides all the processes into groups based on the nodes on which they run. Therefore, if there are n compute nodes and k processes on each node, we get n groups of processes where each group has k members. The grouping of the processes is based on the nodes so that all the local sub-objects can be stored collectively in physical proximity. The storage model can classify the sub-objects into one group per node or one group for a collection of two or more nodes.

Once the data objects are grouped together, our storage model maps them into object-shards and pushes them to the burst buffers or the disk-based parallel file system, depending on the storage quota assigned to the application data.

We experimented with 5 different object-sharding strategies using a representative application (VPIC-I/O [5]) as shown in Table II. Each strategy forms different number of object-shards per node for our use cases. The experimental details of these strategies are presented in the next section. We observed that Strategy-3, which creates $n/2$ object-shards for composite objects and $4n$ object-shards for basic objects performs the best for storing the objects in our studied application. These shards reside partially into the burst buffers and Lustre file system. If required, application can access the object-shards from burst buffers, bringing them back into memory, or send them to the lower storage layer.

Our storage model preserves the information of each object-shard (such as its name and location) into the metadata portion of the object as well as a separate Metadata file. This approach helps us manage the mapping of objects to object-shards without using a separate metadata server.

In summary, our design enables a flexible object-based store with uniform APIs, and allows for seamless migration of data between different hierarchies of the HPC storage stack.

IV. Evaluation

In this section, we present the implementation details of our object-based approach and its evaluation. Our implementation targets I/O of two scientific applications: VPIC-I/O [5], a particle physics simulation that simulates field data and particle data; and HACC-I/O [12], a benchmark that captures the I/O patterns and evaluates the performance for the HACC simulation code.

A. VPIC-I/O

In VPIC-I/O, original data structure contains 8 variables for each particle and these variables are stored as 1-D arrays using HDF5 datasets. Using our object interface, we implemented both types of objects – Basic and Composite. We mapped the particle data into objects, where each process creates one sub-object in the local memory of node to which it belongs. Then we used our object-based storage model to map these sub-objects into object-shards mainly using five object-sharding strategies (specified in Table II) based on number of shards formed on n compute nodes, to which the sub-objects are mapped.

Strategy	No. of Object-Shards for n Nodes		Size of each Object-Shard (GB)	
	Basic	Composite	Basic	Composite
1	8	1	32-525	256-4100
2	8n	n	1	8
3	4n	n/2	2	16
4	2n	n/4	4	32
5	n	n/8	8	64

TABLE II
Specifications of various Object-Sharding Strategies for VPIC-I/O.

We ran all our experiments on the Cray XC40 system 'Cori' installed at the National Energy Research Scientific Computing Center (NERSC) for a scale of up to $16,384$ cores/MPI processes. We measured the 'I/O rate' obtained for storing these object-shards in a file system using our storage model and compared the performance with that of writing the same amount of application data using HDF5 datasets. The 'I/O rate' is the ratio of the amount of total data written to the time taken for storing the data in the file system. We store the object-shards in 2 ways: (1) All object-shards into Lustre (**Horizontal Sharding**); (2) A fraction of total number of object-shards into burst buffers and rest into Lustre (**Vertical Sharding**). For the VPIC-I/O experiments, each MPI process writes 8 M particles. We use a Lustre OST count of 248 and stripe size of 32 M for all of our experiments.

1) Horizontal Sharding: In our tests, we have created and stored object-shards for both basic and composite objects.

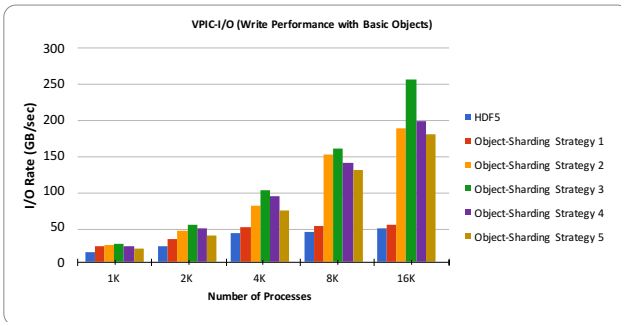

Fig. 3. Write performance for VPIC-I/O Basic Objects.

Basic Objects In the case of storing basic objects, each process creates 8 sub-objects (one for each particle property or variable). Each of these sub-objects are grouped together based on the compute nodes, and then mapped into object-shards. Figure 3 shows the I/O rate obtained for storing the basic objects for five object-sharding strategies compared to that of HDF5 [9] dataset storage. We observed a performance improvement of up to $5\times$ for storing all the objects under our approach (object-sharding strategy 3 giving the maximum performance). The performance difference rises significantly, especially on a scale higher than $4K$ (i.e., 4 x 1024 = 4096) processes. At higher scales, the larger number of object-shards facilitates parallel storage operations, which significantly reduces the contention among processes.

Composite Objects Figure 4 shows the I/O rate obtained to store all object-shards for composite objects into Lustre. As shown in Figure 4, we observe that, similar to basic objects,

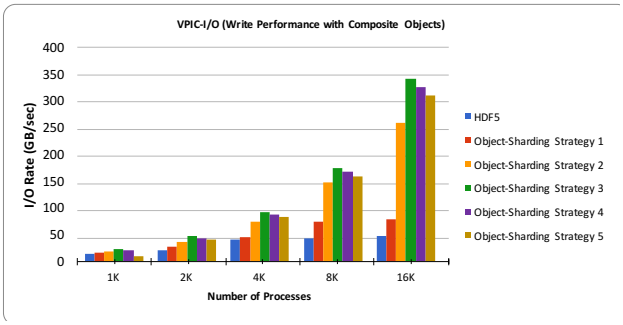

Fig. 4. Write performance for VPIC-I/O Composite Objects.

the performance for object-sharding strategy 3 is much better for composite objects too. This storage performance is mainly achieved due to efficient grouping of processes based on nodes, as all the processes belonging to one node are topologically nearer to each other.

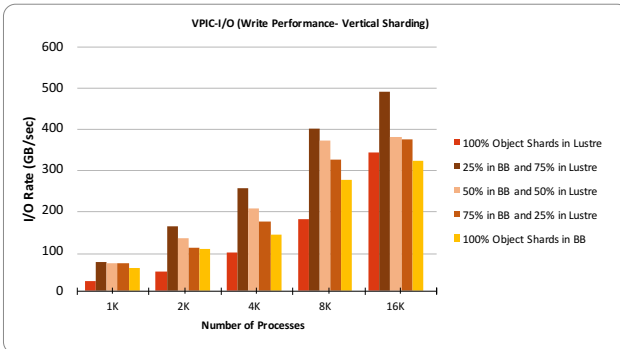

Fig. 5. Write performance for VPIC-I/O using Verical Sharding.

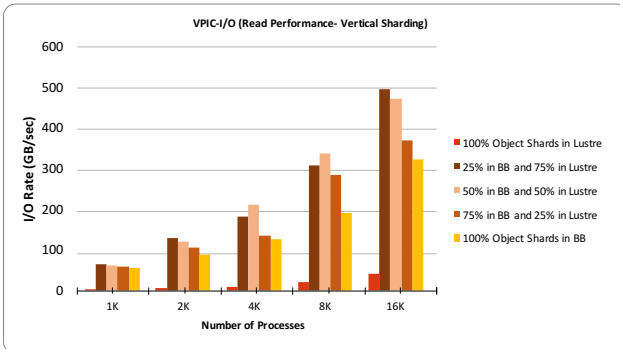

Fig. 6. Read performance for VPIC-I/O using Verical Sharding.

2) Vertical Sharding: As described earlier (Section III-B), one of the main contributions of our object storage design model is that, given its uniform storage APIs, it can store object-shards on multiple layers of storage. In this study, we have stored data partially on both SSD-based burst buffers and disk-based Lustre by assigning a storage quota for each application based on performance requirements. Some applications need to keep hot objects closer, which can be achieved by storing them into burst buffers.

In Figure 5, we show the performance of storing all object-shards for VPIC composite objects into Lustre compared to storing 25%, 50%, 75%, or 100% of them into buffers and rest into Lustre, using object-sharding strategy 3. We have statically chosen these distributions between the two layers to demonstrate the concept. Users may choose better distribution based on the knowledge of application requirements. We show in the figure a comparison of vertical sharding with horizontal sharding, where all object-shards are stored in Lustre. Storing all object-shards in burst buffers may not be feasible due to capacity limitations and by moving just 25% of the object-shards from Lustre to burst buffers, it provides up to $2.4\times$ improvement compared to the case where all of them are stored in Lustre and up to $8\times$ compared to HDF5 (Figure 4). Thus, our uniform API approach provides a flexible mechanism to bring objects that are being used frequently (hot data) closer to the computation in a seamless manner as needed.

We also measured the I/O rate for reading partial object-shards stored in the burst buffers and compared the performance with reading all of the shards from Lustre. In Figure 6, we show the performance of reading the object-shards for VPIC composite objects sharded vertically between Lustre and burst buffer using object-sharding strategy 3. As burst buffer layer is closer to computation, reading the object-shards partially from burst buffer improves the performance by up to $8\times$ compared to reading all of them from the Lustre.

B. HACC-I/O

For HACC-I/O, we implemented the objects as composite objects. Each particle of HACC simulation consists of nine variables representing some attributes of the application. For all the experiments, we stored data for $8\ M$ particles per process. All the experiments are done on the NERSC's Cori supercomputers for scale of up to 16384 cores. We used object-sharding strategies 1 to 4 (specified in Table II) for horizontal sharding of HACC-I/O application data using the composite objects. The baseline implementation of HACC-I/O uses MPI-IO [21] library.

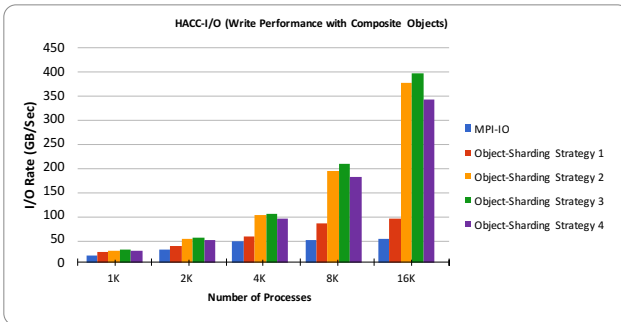

Fig. 7. Write performance for HACC-I/O Composite Objects.

1) Storage Performance using Horizontal Sharding: We show a comparison of I/O rate for storing all the object-shards under the four object-sharding strategies with storing application data using MPI-IO into one single file in Figure 7.

We observe that as in the case of VPIC-I/O, object-sharding strategy 3 (which creates $n/2$ object-shards on n nodes) performs significantly better. We observe a performance gain of up to $6\times$ for storing the object-shards with strategy 3 as compared to MPI-IO. It can also be seen in the figure that performance gain increases manifold at higher scales. This performance gain is achieved by significant contention reduction among processes for storing multiple object-shards in contrast to single shared file architecture.

2) Read Performance using Horizontal Sharding: We also measured the performance for reading objects from object-shards and compared the same with the MPI-IO single-file model. In Figure 8, we show that the I/O rate obtained to read all the objects shards using the three object-sharding strategies as well as using MPI-IO. We observed a performance gain of up to 30% using sharding strategy 3.

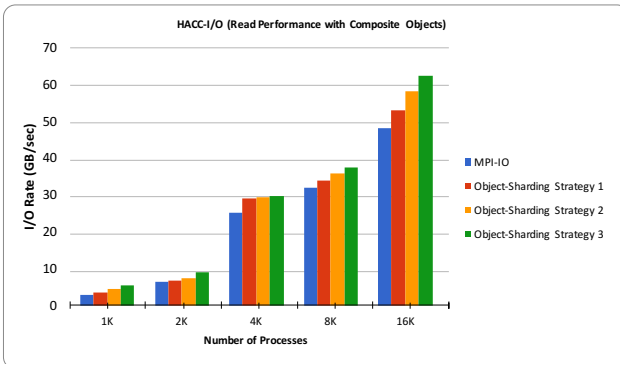

Fig. 8. Read performance for HACC-I/O Composite Objects.

V. Related Work

A number of recent research efforts explore object-based storage systems. The T10 standards [8], [19] were proposed to store data and attributes as objects. Based on T10 standards, Seagate, IBM (ObjectStore [7]), and Panasas (PanFS [25]) implemented prototypes [19] and demonstrated the capability of object-based storage systems. RADOS [24]], as part of Ceph [23], is a scalable and reliable object storage service for petabyte-scale storage clusters. Lustre views a file as multiple objects. For instance, each Lustre object is implemented as files on an object storage target (OST) that represents a local file system. HDF5 data model offers three types of objects-groups, datasets, and links between objects and stores them into file-based storage systems. DAOS (Distributed Application Object Storage) [13] is an object storage system that encapsulates data of storage stack in DAOS containers and provides distributed transactional object store. NVRAM and various implementations of FLASH devices have also been proposed as solutions to alleviate I/O performance issues of HPC systems. Rajimwale et al. [20] revisited the traditional data model of HDD on SSD devices and found that the object-based storage is more suitable for these new devices. Kang et al. [16] proposed object-based models to support hardware devices with different configurations. Application related information is allowed to be exchanged through the object interface and this provides for performance benefits and object-level reliability. Lee et al. [18] implemented a SSD based object storage system, where the attributes and I/O usage information is stored as metadata. Muninn is an object-based versioning key-value store [15] to enable transparent versioning on file systems.

Despite various studies of object-based storage solutions, there is no uniform object management across all the memory and storage layers that will be common in exascale systems. Generally, existing research focuses on an individual level without considering the presence of other layers. When data moves through the hierarchies, semantic information embedded in objects are 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. Our approach addresses this shortcoming, and offers an integrated solution for deep hierarchy storage for emerging exascale systems.

VI. Conclusions

We have presented an initial design of a novel object based storage interface to facilitate storage and I/O for HPC applications in exascale systems that will comprise deep memory and storage hierarchy. Our interface maps scientific applications' data into objects that can store both simple as well as complex data structures along with their properties to preserve semantic information in various layers of memory and storage. Our approach can offload the task of managing MPI-IO or HDF5 calls for data storage from the users to our system. To achieve high performance as well as storage efficiency, we store objects partially into SSD-based burst buffers or node-local persistent storage as well as on disk-based parallel file system (e.g. Lustre) based on application requirements. We have implemented our proposed model with two scientific use cases, VPIC-I/O and HACC-I/O, and evaluated it for a scale of up to $16K$ processes. Experimental results show that compared to the state of the art, our object-based storage model can improve the I/O performance by up to $7\times$. Based on this initial success, we are developing a runtime system to optimize sharding strategies, to move data asynchronously and pro-actively, and to support data processing while the data is in transit among different storage layers.

Acknowledgment

This work is supported in part 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 work is also sponsored in part by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility.

REFERENCES

[1] Apache Spark. https://wiki.openstack.org/wiki/Swift. Accessed: March 20 2017.

[2] CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks/#hacc.

[3] OpenStack Object Storage. https://wiki.openstack.org/wiki/Swift. Accessed: Jul 22 2016.

[4] Parallel I/O Kernel (PIOK) Suite. https://sdm.lbl.gov/exahdf5/software.html.

[5] 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 International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 59:1–59:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 27:1–27:10, New York, NY, USA, 2007. ACM.

[8] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 119–123, June 2005.

[9] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An Overview of the HDF5 Technology Suite and Its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA, 2011. ACM.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGPLAN Not.*, 33(11):92–103, Oct. 1998.

[11] G. Grider. MarFS: A Scalable Near-POSIX File System over Cloud Objects Background, Use, and Technical Overview, 2016.

[12] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. HACC: Extreme scaling and performance across diverse architectures. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, Nov 2013.

[13] Intel. DAOS Lustre Restructuring and Protocol Changes Design : For extreme-scale computing research and development (FastForward) storage and I/O, 2014. https://goo.gl/oCKLso.

[14] P. J.Braam. The Lustre Storage Architecture (Technical Report). Technical report, Available: http://wiki.lustre.org/., 2004.

[15] Y. Kang. *High-Performance, Reliable Object-Based NVRAM Devices*. PhD thesis, Storage Systems Research Center, University of California, Santa Cruz, 9 2014.

[16] Y. Kang, J. Yang, and E. Miller. Object-based SCM: An efficient interface for Storage Class Memories. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12, May 2011.

[17] Q. Koziol. Design and implementation of FastForward features in HDF5 for extreme-scale computing research and development (FastForward) storage and I/O. Technical report, The HDF Group, 2014.

[18] Y.-S. Lee, S.-H. Kim, J.-S. Kim, J. Lee, C. Park, and S. Maeng. Ossd: A case for object-based solid state drives. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–13, May 2013.

[19] D. Nagle, M. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4-5):401–412, 2008.

[20] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 21–21, Berkeley, CA, USA, 2009. USENIX Association.

[21] R. Thakur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.

[22] S. R. Walli. The POSIX Family of Standards. *StandardView*, 3(1):11–17, Mar. 1995.

[23] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[24] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 35–44, New York, NY, USA, 2007. ACM.

[25] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.