Exploring Scientific Application Performance Using Large Scale Object Storage

Steven Wei-der Chien[1](✉), Stefano Markidis[1], Rami Karim[1], Erwin Laure[1], and Sai Narasimhamurthy[2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
{wdchien,markidis,ramik,erwinl}@kth.se
[2] Seagate Systems UK, Havant, UK
sai.narasimhamurthy@seagate.com

Abstract. One of the major performance and scalability bottlenecks in large scientific applications is parallel reading and writing to supercomputer I/O systems. The usage of parallel file systems and consistency requirements of POSIX, that all the traditional HPC parallel I/O interfaces adhere to, pose limitations to the scalability of scientific applications. Object storage is a widely used storage technology in cloud computing and is more frequently proposed for HPC workload to address and improve the current scalability and performance of I/O in scientific applications. While object storage is a promising technology, it is still unclear how scientific applications will use object storage and what the main performance benefits will be. This work addresses these questions, by emulating an object storage used by a traditional scientific application and evaluating potential performance benefits. We show that scientific applications can benefit from the usage of object storage on large scales.

Keywords: Scientific applications · Object storage · Parallel I/O HPC · HDF5

1 Introduction

Parallel I/O is becoming one of the most serious performance bottlenecks in HPC applications as the number of processes writing/reading to/from the supercomputer I/O system keeps increasing at a considerable pace. An exascale supercomputer will likely support billions of processes [4] that can potentially access and update shared files, all at the same time. The implementation of existing HPC parallel interfaces, such as MPI I/O, HDF5 and NetCDF are all based on and complaint to the POSIX standard. The POSIX standard requires strong consistency when accessing and updating a file. In a parallel environment, strong consistency is achieved by a process acquiring a lock on the file, completing the operation and releasing the lock. One reason for the performance bottleneck of

HPC parallel I/O interfaces is the strong consistency POSIX requirement, and its implementation.

One of the possible disruptive solutions to address the lack of scalability of traditional parallel I/O would be the adoption of object storage technology [11]. Object storage is a well-spread technology in cloud computing and is currently being utilized by large tech companies. For instance, Amazon and Google, to mention a few, have implemented Amazon S3 and Google Cloud object storage; services which are used by many companies today. Object storage abandons traditional POSIX I/O concepts, such as directories, files, and certain file operations. Unlike many other parallel file systems, object storage provides a single flat global name space and supports only a few operations, among which are the PUT and GET operations. Performance scalability of object storage is partly due to the concept of *object immutability* and also due to the semantics of the PUT and GET operations. Objects are *immutable*, as in-place changes to the data in the object are not possible. A PUT operation creates an object, adds data to it and returns an object Universally Unique Identifier (UUID). A simple hash function or a combination of them can then be used to determine the location of the object in Object Storage Device (OSD). The main two advantages of object storage when compared to traditional approaches are:

1. Because objects are immutable, it is impossible for a node to write to an object that is being read by others. This allows removal of locks while reading data from an object, providing a lock-free data access.
2. Because physical object locations can be determined by object UUID and hashing function, it possible to directly access data without any locational metadata.

The main limitation of the object storage is that it requires additional software for metadata. In fact, metadata, such as the object name, creation time, etc., that is not comprised in the object storage needs to be stored and managed outside the object storage. For this reason, objects stores are usually equipped with a key-value store, providing users with a front-end interface and mapping object UUID to metadata.

A simplified diagram of object storage is presented in Fig. 1, showing an application putting two objects in the object storage, similar to CEPH object storage [27]. With the PUT operation, two objects are created and their UUIDs are retrieved. A hash function is used to determine the placement group (an object pool) and then the physical location on the OSDs. The PUT operation also inserts associated metadata to the metadata server.

While object storage is a promising technology that could potentially replace parallel file systems in the future, it is unclear how current scientific applications running on supercomputers will use object storage and what the potential benefits are. To the best of our knowledge, there are no large scale supercomputers that are directly using object storage. For this reason, our goal is to take an application with similar workload to typical HPC applications, and emulate object storage on it. We emulate object storage to write both individual and shared

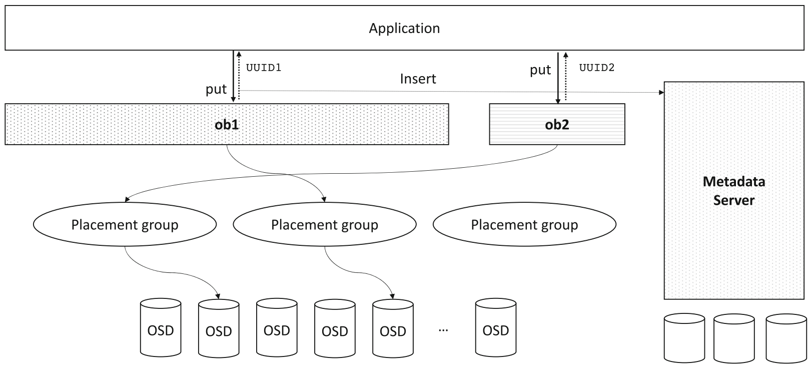

Fig. 1. Simplified diagram of an application writing two objects to the object store, similar to CEPH [27].

objects and to compare the I/O performance with existing parallel HDF5 implementations [12].

The paper makes the following contributions:

– We propose a methodology to evaluate the scalability of object storage at scale and develop a simple emulator to write and read objects on an object store serving large supercomputers.
– We deploy the object storage emulator in a representative massively parallel application and measure the I/O performance.
– We analyze the emulated object storage performance at scale, compare it with performance of parallel HDF5 and evaluate the object storage potential for extreme scale systems.

The paper is organized as follows. First, we provide background to this study and present related work in Sect. 2. Section 3 presents the design and implementation of an emulator for object storage at scale. Section 4 introduces the benchmarking environment. Section 5 presents the results. Finally, Sect. 6 summarizes the results, discusses the limitations of this work and outlines future work.

2 Background and Related Work

In this section, we provide an overview of common parallel I/O libraries and file systems, together with the related work.

POSIX I/O. POSIX (Portable Operating System Interface) is a specification defined by IEEE Computer Society for a standardized operating system interface and environment. Among other operations and concepts, POSIX defines the interaction between file descriptors and standard I/O streams [3]. From a programmer's perspective, POSIX I/O is characterized by the following interfaces: open(), close(), read(), write(), and lseek(). Additionally, the POSIX standard specifies the semantics of such operations. POSIX I/O is stateful and requires the operating system to maintain persistent states. In order to modify a file one must first open a descriptor, seek a location and read or write from there. File descriptors are not shared between processes and the system must maintain every descriptor opened by all processes.

Another feature of POSIX I/O is the strict consistency requirement. POSIX defines that after a successful write() call, any subsequent read() operations from the byte positions in the modified file must return data written by that write() operation [3]. The consistency semantics is often implemented with some form of locking mechanism [20]. In the case of parallel file systems, this is often implemented via distributed locking [14].

Parallel File Systems. Parallel file systems such as *Lustre* [2], *GPFS* [20] and *VPFS* [9] are created to support parallel I/O in a cluster environment. These file systems implement I/O forwarding at an extra layer between the storage system and computing system, which handles I/O on behalf of the computing systems. One of the widely adopted parallel file systems is Lustre. Lustre Metadata Servers (MDS) handles information such as physical locations, file names, permissions and timestamps. Data is striped and sent to different Object Stores Servers (OSS) [21]. Although Lustre is object store based, it exposes itself through POSIX I/O interface and is near POSIX compliant. POSIX consistency semantics is enforced through distributed locking. One performance bottleneck of Lustre lies on metadata management. Another bottleneck lies in file locking, which it is required to preserve consistency. Excessive striping can also negatively impact performance [28,29]. Performance of parallel writing to a single file can be improved by explicit configuration of striping, yet it increases failure risk and worsen performance of writes to non-shared files [5].

Parallel I/O Libraries. It is common to use parallel file systems with parallel I/O libraries. *MPI-IO* [23], *Parallel HDF5* [24] and *Parallel netCDF* [13] are parallel I/O libraries that work on top of parallel file systems. MPI-IO aims to provide a portable interface which addresses common parallel I/O patterns, such as collective I/O and non-contiguous access. MPI-IO exposes several POSIX like I/O interfaces with relaxed consistency requirements. MPI-IO guarantees that a write from one process is immediately visible to processes in the same communicator group in which the file was opened with atomic mode. Otherwise the content is only visible after explicit synchronization. Many I/O libraries, such as HDF5 and Parallel netCDF, are built upon MPI-IO to take advantage of portable parallel I/O. However, MPI-IO provides little performance improvement for contiguous access [5]. Due to its similarity to POSIX I/O, the performance of these two APIs is often similar.

Object Storage System. Object storage system is an architecture which manages data as objects instead of files [11]. Due to their scalability, object stores are widely adopted in cloud based systems. Object store operations are stateless and in object store semantics there are only two basic operations: GET and PUT. A PUT operation returns an ID which uniquely represents the object. Object store implementations usually provides a facility to map an assigned name to an ID, together with metadata which describes the object. This is often implemented with a key-value store. All objects are stored without structure and clients communicate directly to the storage node where data physically resides without requiring location lookup by hashing the object's ID [27]. Objects are immutable and it is impossible to concurrently create or update the same object. This eliminates the bottleneck caused by locking. In contrary to POSIX I/O, object stores support a weak form of consistency: eventual consistency. This means that a successfully returned PUT operation does not necessary require that the object will be visible immediately. Deterministic placement of object through ID hashing leads to the elimination bottleneck due to lookup.

CEPH [27] is one of the most commonly known object storage system. It exposes itself as through a POSIX interface and at the same time provides a number of POSIX I/O extensions which provides relaxed consistency. Unlike Lustre, any party can compute the physical location of an object by hashing its ID. For this reason, location metadata is completely eliminated. This reduces the stress on the metadata cluster. Additionally, it is possible to manipulate the underlying object store directly through librados [1]. Additional emerging object storages, targeting HPC workloads, are Seagate's Mero [15,16], DAOS [6] and DDN's Web Object Store (WOS) [10]. Studies have also been made on how HPC applications can interact with these transaction based storage systems [19]. The adoption of these systems enabled a wider range of underlying storage technologies to be used, such as hybrid-memory storage systems and Non-volatile memory storage systems [17,18].

I/O Pattern in Scientific Applications. The majority of scientific applications perform a large number of write operations to preserve intermediate states and final outcome of simulation variables for post-processing (visualization and data analysis) and check-pointing. In scientific applications, these operations occur typically at a given computational cycle, defined by users. Because of the computational cost of parallel I/O, I/O operations are kept at minimum in scientific applications. Typically, these outputs are only for the purpose of archiving, later post-processing and check-pointing for restarting simulations.

Parallel I/O operations either:

1. Write/Read one file per process (independent parallel I/O), or
2. Write/Read to the same shared file (cooperative parallel I/O). In this case, parallel I/O is performed using parallel I/O libraries, such as MPI-IO, Parallel HDF5 and NETCDF.

Studies have shown that parallel write to the same file often results in worse performance than writing to individual non-shared files [5,7,26], implying that existing parallel file systems are not well suited for parallel I/O. For Lustre, custom striping configuration can result in similar performance between the two approaches, but also results in higher failure risk.

In terms of concurrent write, it is rarely the case that a process requires the latest update through a read operation immediately after a write operation. This can be efficiently implemented through MPI point-to-point or collective where data is in-memory if data sharing is needed. Therefore, the strict read-after-write consistency requirement imposed by POSIX I/O is rarely required.

By decoupling metadata management and relaxing consistency requirements, object stores can potentially provide extreme scalability for parallel I/O. Scalability is achieved through deterministic placements and lock-free accesses. Since most scientific applications are write intensive and do no rely on POSIX consistency guarantees, we argue that object stores will be extremely valuable for scientific applications.

3 Emulating Scientific Applications Using Object Storage

Our goal is to assess the impact of object storage system on supercomputers in HPC scientific applications. In particular, we are interested in how object stores can improve scalability of such applications. Yet, HPC-ready object stores are not widely adopted. For this reason, we have designed and implemented a simple library that emulates the workings of an object storage system. Specifically, we emulate four key features of object storage systems, namely: flat namespace structure, object immutability, deterministic object placement and metadata management. We implement the API according to object store semantics. A GET operation retrieves an object and a PUT operation creates a new object as shown in Fig. 2. Metadata are stored as serialized binary files to mimic a key-value store where the file name is the key. Furthermore, we support chunking operations. This implies that it is possible to create an object concurrently by different processes.

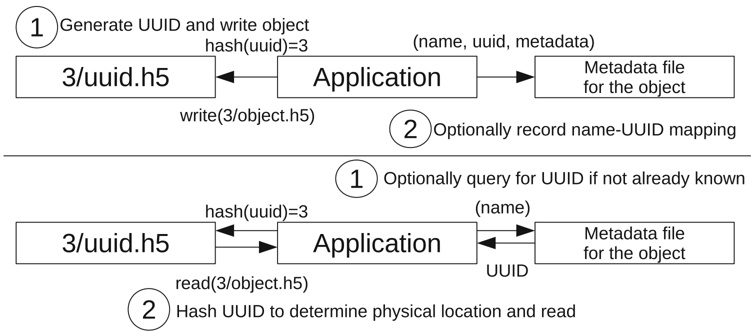
1
Generate UUID and write object
hash(uuid)=3
(name, uuid, metadata)
3/uuid.h5
Application
Metadata file for the object
write(3/object.h5)
2
Optionally record name-UUID mapping
1
Optionally query for UUID if not already known
hash(uuid)=3
(name)
3/uuid.h5
Application
Metadata file for the object
UUID
read(3/object.h5)
2
Hash UUID to determine physical location and read

Fig. 2. Illustration of how our emulator mimics object store PUT (top) and GET (bottom) operations in an application.

3.1 Emulator Implementation

We implement an emulator to mimic an object storage serving a large scale supercomputer as a C library. For the purpose of the experiment, we support the storage of multidimensional arrays in 64-bit double, 32-bit float and 32-bit integer data-type as objects. Objects are stored in one or more HDF5 files. These HDF5 files are represented by UUID and a part number. Metadata are represented in protobuf serialized data format. Furthermore, we emulate different Object Storage Devices (OSD) as different folders. Our emulator employs a weak form of consistency: eventual consistency. This means that an object being written will not be immediately visible to other processes, but will eventually be. We define that an object is visible after the metadata is written and synchronized to disk. The writing of metadata is performed after all data is successfully written. This is to support wait-free read by other processes to an object with the same name. In this way, processes will retrieve the ID of the object from existing metadata, which point to an existing object, while the new object is being written to another location with another ID. We also write a HDF5 virtual dataset which links all the chunks together and present a unified view of the entire object. HDF5 virtual datasets can be opened as a single dataset with existing HDF5 dataset APIs as if it is one single file. How our emulator writes an object chunk and correspondent metadata is shown in Fig. 3.

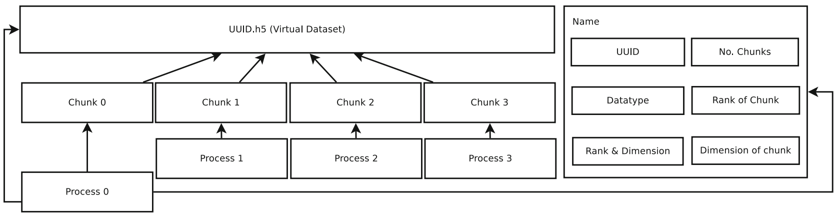

Fig. 3. Illustration of how the emulator uses object chunks and metadata.

Objects Creation. A new object is created with a PUT operation. A PUT operation receives object name, data and metadata which describes the object, and performs operations. We support multidimensional data in a variety of datatypes. Additionally, we represent descriptive information such as size and dimension as metadata. When a PUT request is received, a new UUID is generated to uniquely represent the object. We use a hash function to determine in which object storage device the object will the be placed. A new HDF5 dataset will be created to store the structured input data and stored in an HDF5 file with the UUID as filename.

Object Chunking. During object chunking, the object is divided into equal sized portions and stored individually with many HDF5 files. Multidimensional chunks are supported. A chunk ID is appended to the filename for reconstruction. Immediately before the metadata is being written, a HDF5 Virtual Dataset (VDS) is created to provide a high level overview of the object being written [7,25]. Thus, the object chunks can be retrieved as a single object thereafter. Figure 4 shows our object store design, where individual object chunks are represented as individual HDF5 files and are linked via VDS according to part number.

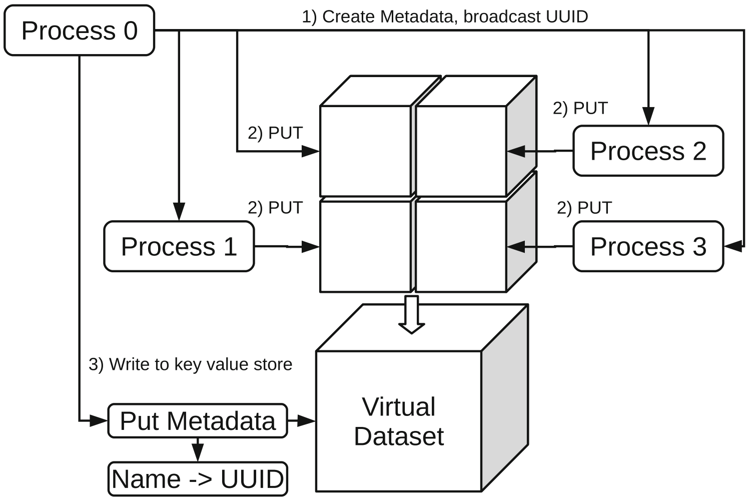

Fig. 4. In our emulated object storage, object chunks are stored as HDF5 files and metadata as Virtual Data Sets (VDS).

Metadata Management. We represent object metadata using protobuf, which is a serialization mechanism developed by Google. Our emulated object storage supports data in multi-dimensional tensors. So, in our particular case, we store the rank of the tensor, the size for each tensor dimension and the UUID. If the object is chunked, then the chunk size, dimension and chunk count will be also stored. The protobuf object will be serialized and written to a temporary file on disk. After the file is synchronized to disk, we perform a POSIX rename() and synchronization to rename the temporary file to the user defined object name. This ensures that a third party client who is accessing the metadata file with the same name will either get a new or an old copy of the metadata. A client who is holding a file descriptor to the old metadata will still be able to read the old copy of the metadata. In the case where multiple clients are creating different chunks of the same object, the master process must initiate a PUT process by obtaining a copy of the metadata and UUID from the library. The UUID will be broadcast to other processes and they can perform their own chunked PUT by supplying the data, UUID and part number. When all processes have completed their respective chunked PUT, the master process performs a commit to write the metadata to disk so that the new object will be visible. It is the application's responsibility to ensure that all chunked PUT operations by different processes are complete. When two processes perform PUTs with the same object name concurrently, the most up-to-date version of the object is the object of the last process writing the metadata.

4 Experimental Environment

Our experiments are performed on the Beskow supercomputer at KTH. Beskow is a Cray XC40 system, consisting of 2,060 compute nodes, equipped with two Xeon E5-2698v3 Haswell 2.3 GHz CPUs (16 cores per CPU) per node and high speed network Cray Aries. The storage employs a Lustre parallel file system (client v2.5.2) with 165 OST servers. Beskow OS is SUSE LINUX (Release 11). We use GCC version 4.9.1, Cray-MPICH v7.0.4 and HDF5 v1.10.1.

To measure the I/O performance, we use the Darshan profiler [8]. Darshan is a low-overhead tool to investigate the I/O performance of parallel applications. Darshan provides bandwidth and measured time spent on I/O. Measurement is done at MPI I/O and POSIX level. For this reason, the Darshan tool is capable of profiling both our emulated library and parallel HDF5 as our library is implemented through HDF5 which is based on POSIX I/O; and parallel HDF5 which is based on MPI I/O.

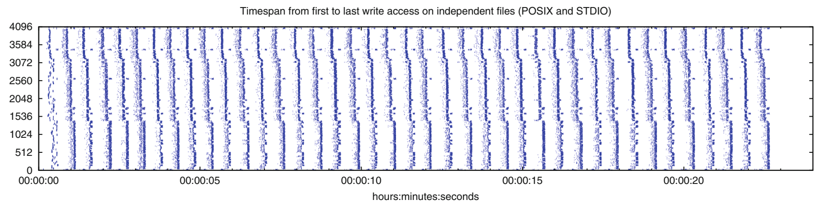

Fig. 5. Different processes writing to individual object chunks.

We implemented a skeleton application iterating over 200 computational cycles. Every five cycles, we perform parallel I/O. Therefore, the whole execution cycle consists of 40 I/O phases. The computation and I/O phases take approximately 75% and 25% of the total execution time respectively. We perform weak scaling test, keeping the amount of data to be written by each process constant while varying number of processes. For every I/O phase, we output a two dimensional integer array where the size is a multiple of chunk size with the number of processes. A broadcast of UUID is performed by rank 0, processes write their chunk and sign-in to a barrier. Rank 0 commits metadata to disk after all processes are signed-in. To compare the I/O performance with parallel HDF5, a separate experiment that utilizes HDF5 was created. The set-up consists of the same skeleton application performing calculations at each iteration and I/O with parallel HDF5 to write to a shared file every five cycles. MPI hints are provided to utilize Lustre striping with a stripe count equal to the number of MPI processes in use divided by the number of processes per stripe.

We created test configuration with different chunk sizes. We tested 16×4096, 32×4096 and 64×4096. For parallel HDF5, we additionally test for different number of processes per stripe. We tested 16, 32 and 64, where 32 is the number of processors on one computing node of Beskow. For each configuration we repeat the tests 5 times and report the median, minimum and maximum bandwidth in MiB/s. For time spent on I/O operation we report the average value in seconds.

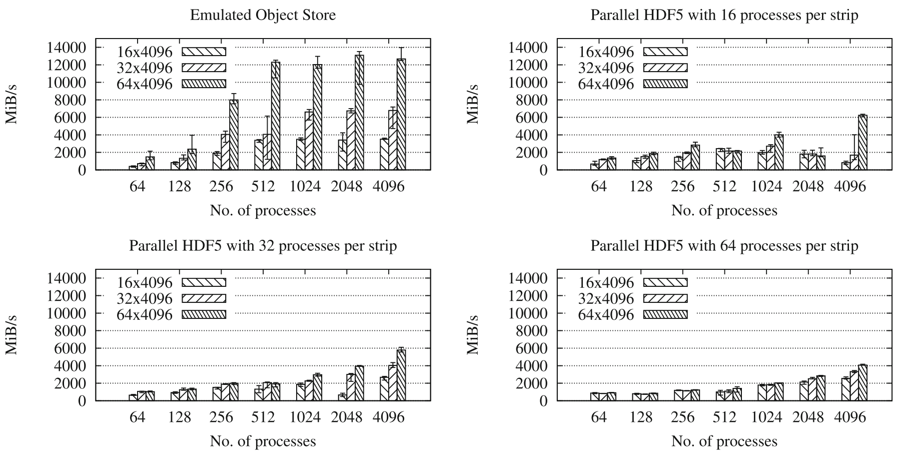

Fig. 6. Median bandwidth in MiB/s measured by Darshan for different configurations. Maximum and minimum bandwidth recorded are represented by error bars. Scaling of workload results in increase of bandwidth in both methods but on a different scale.

5 Evaluation

We perform scaling tests up to 4,096 processes. We measure both time spent on I/O operations and bandwidth of write operations. Figure 5 shows I/O patterns of different processes of a particular configuration using our emulator where processes perform putting of their data chunk. Figure 6 shows the bandwidth under different chunk sizes and stripe configuration by the application with the two I/O methods: emulated object storage and parallel HDF5 to a shared file. Our emulator outperforms parallel HDF5 in terms of bandwidth. Comparing to the emulator, parallel HDF5 only provides moderate scaling in bandwidth.

We also noticed that the performance of parallel HDF5 is extremely sensitive to configurations. Figure 7 shows the maximal and minimal bandwidth measured from writing operations among all configurations.

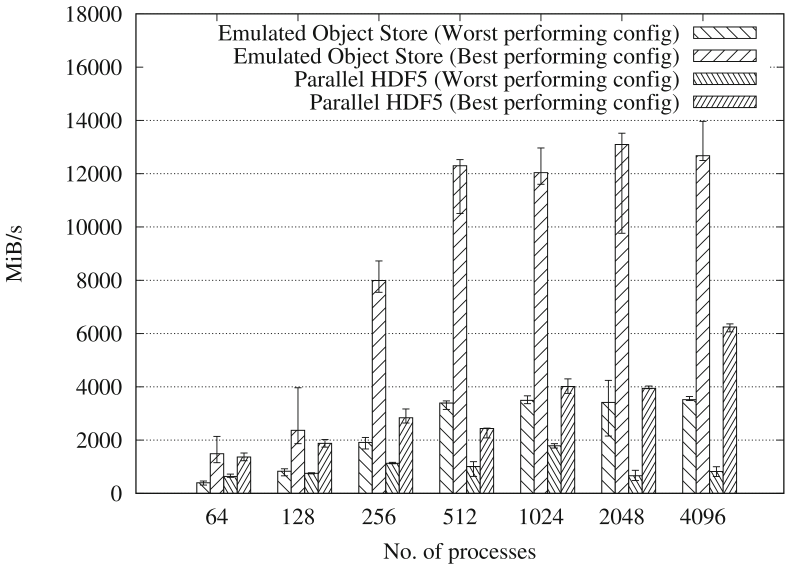

Fig. 7. Configurations with maximal and minimal bandwidth in MiB/s measured by Darshan among different configurations. Both methods provide comparable bandwidth with small number of processes.

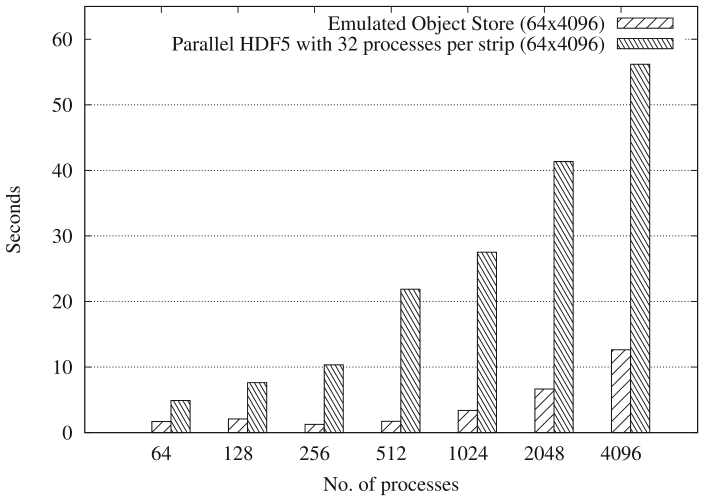

Fig. 8. Average total time spent on I/O in seconds with chunk size 64 × 4096. For parallel HDF5 32 processes per strip is set.

We find that our emulated object store implementation provides better scalability than writing to shared file. Although still outperforming parallel HDF5, after more than 2,048 processes are used, we observe saturation and slight decline in bandwidth. We also observed that the bandwidth scales together with increasing chunk size with our emulator. The same is observed with parallel HDF5. Yet the scaling comparing with the emulator is only moderate. Figure 8 shows the total time spent on I/O during application execution. We observe a large increase of time spent on I/O relative to number of processes used for the implementation with parallel HDF5. On the other hand, the implementation with our emulated object store shows relatively little change in terms of time spent.

6 Conclusion

One of the performance and scalability bottlenecks in large scientific applications is parallel I/O to file systems. In fact, the usage of parallel file systems and consistency requirements of POSIX (that all the traditional HPC parallel I/O interfaces adhere to) poses limitations to scientific applications. Object storage is a promising technology that could address the parallel I/O scalability issues at extreme scale. In this work, we designed and implemented a library to emulate the object storage operation semantics with the goal of understanding the scalability benefit of scientific HPC applications, using object storage on large scale supercomputers. We showed that scientific applications can benefit from the usage of object storage on large scales.

In the future, we would like to apply our library to HPC applications with heavy I/O workload patterns and investigate further for potential improvements. In particular, we would like to implement a submodule with IOR [22], an I/O benchmarking software such that we can perform qualitative studies of how object store I/O semantics can contribute to scalability. Through these studies, we also hope to identify the requirements for HPC oriented object stores and how they can contribute to future highly parallel systems.

Acknowledgments. Funding for the work is received from the European Commission H2020 program, Grant Agreement No. 671500 (SAGE).

References

1. Introduction to Librados. http://docs.ceph.com/docs/master/rados/api/librados-intro/
2. Lustre: A scalable, high-performance file system. Cluster File Systems Inc., White Paper (2002)
3. IEEE standard for information technology-portable operating system interface (POSIX(R)) base specifications, issue 7. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (2018)
4. Bergman, K., et al.: Exascale computing study: technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Technical Report 15 (2008)

5. Borrill, J., Oliker, L., Shalf, J., Shan, H.: Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 1–12 (2007)
6. Breitenfeld, M.S., et al.: DAOS for extreme-scale systems in scientific applications. arXiv preprint arXiv:1712.00423 (2017)
7. Byna, S., Chaarawi, M., Koziol, Q., Mainzer, J., Willmore, F.: Tuning HDF5 subfiling performance on parallel file systems (2017)
8. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale I/O workloads. In: IEEE International Conference on Cluster Computing and Workshops, CLUSTER 2009, pp. 1–10. IEEE (2009)
9. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: a parallel file system for Linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, vol. 4. p. 28 (2000). https://www.usenix.org/legacy/publications/library/proceedings/als00/2000papers/papers/full_papers/carns/carns_html/#foot9
10. Data Direct Networks: WOS: Object storage. https://www.ddn.com/products/object-storage-web-object-scaler-wos/
11. Factor, M., Meth, K., Naor, D., Rodeh, O., Satran, J.: Object storage: the future building block for storage systems. In: Local to Global Data Interoperability-Challenges and Technologies, pp. 119–123. IEEE (2005)
12. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the HDF5 technology suite and its applications. In: Proceedings of the EDBT/ICDT2011 Workshop on Array Databases, pp. 36–47. ACM (2011)
13. Li, J., et al.: Parallel netCDF: a high-performance scientific I/O interface. In: 2003 ACM/IEEE Conference on Supercomputing, p. 39 (2003)
14. Mohindra, A., Devarakonda, M.: Distributed token management in Calypso file system. In: Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing, pp. 290–297 (1994)
15. Narasimhamurthy, S., et al.: The SAGE project: a storage centric approach for exascale computing. In: Proceedings of Computing Frontiers. ACM (2018)
16. Narasimhamurthy, S., et al.: SAGE: percipient storage for exascale data centric computing. Parallel Comput. (2018). https://doi.org/10.1016/j.parco.2018.03.002
17. Peng, I.B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., Markidis, S.: Exploring the performance benefit of hybrid memory system on HPC environments. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 683–692. IEEE (2017)
18. Peng, I.B., Markidis, S., Laure, E., Kestor, G., Gioiosa, R.: Exploring application performance on emerging hybrid-memory supercomputers. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications, IEEE 14th International Conferenceon Smart City, IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 473–480. IEEE (2016)
19. Rivas-Gomez, S., et al.: MPI windows on storage for HPC applications. In: Proceedings of the 24th European MPI Users' Group Meeting, p. 15. ACM (2017)
20. Schmuck, F.B., Haskin, R.L.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the Conference on File and Storage Technologies, FAST 2002, pp. 231–244 (2002)
21. Schwan, P., et al.: Lustre: building a file system for 1000-node clusters. In: Proceedings of the 2003 Linux Symposium, vol. 2003, pp. 380–386 (2003)

22. Shan, H., Antypas, K., Shalf, J.: Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, p. 42. IEEE Press (2008)
23. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and withhigh performance. In: Proceedings of the Sixth Workshop on I/O in Paralleland Distributed Systems, pp. 23–32 (1999)
24. The HDF Group: Hierarchical Data Format, version 5 (1997). http://www.hdfgroup.org/HDF5/
25. The HDF Group: HDF5 Virtual Dataset (2014). https://support.hdfgroup.org/HDF5/Tutor/vds.html
26. Wang, F., et al.: File system workload analysis for large scale scientific computing applications. Technical report (2004)
27. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th symposium on Operating systems design and implementation, pp. 307–320. USENIX Association (2006)
28. Xu, C., et al.: LIOProf: exposing Lustre file system behavior for I/O middleware (2016). https://cug.org/proceedings/cug2016_proceedings/at_a_glance.html
29. Yu, W., Vetter, J., Canon, R.S., Jiang, S.: Exploiting Lustre file joining for effective collective I/O. In: Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), pp. 267–274 (2007)