LibCOS: Enabling Converged HPC and Cloud Data Stores with MPI

Daniel Medeiros
KTH Royal Institute of Technology
Sweden
dadm@kth.se

Stefano Markidis
KTH Royal Institute of Technology
Sweden
markidis@kth.se

Ivy Peng
KTH Royal Institute of Technology
Sweden
bopeng@kth.se

ABSTRACT

Recently, federated HPC and cloud resources are becoming increasingly strategic for providing diversified and geographically available computing resources. However, accessing data stores across HPC and cloud storage systems is challenging. Many cloud providers use object storage systems to support their clients in storing and retrieving data over the internet. One popular method is REST APIs atop the HTTP protocol, with Amazon's S3 APIs being supported by most vendors. In contrast, HPC systems are contained within their networks and tend to use parallel file systems with POSIX-like interfaces. This work addresses the challenge of diverse data stores on HPC and cloud systems by providing native object storage support through the unified MPI I/O interface in HPC applications. In particular, we provide a prototype library called LibCOS that transparently enables MPI applications running on HPC systems to access object storage on remote cloud systems. We evaluated LibCOS on a Ceph object storage system and a traditional HPC system. In addition, we conducted performance characterization of core S3 operations that enable individual and collective MPI I/O. Our evaluation in HACC, IOR, and BigSort shows that enabling diverse data stores on HPC and Cloud storage is feasible and can be transparently achieved through the widely adopted MPI I/O. Also, we show that a native object storage system like Ceph could improve the scalability of I/O operations in parallel applications.

KEYWORDS

object storage, S3, Ceph, MPI, parallel computing

ACM Reference Format:
Daniel Medeiros, Stefano Markidis, and Ivy Peng. 2023. LibCOS: Enabling Converged HPC and Cloud Data Stores with MPI. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2023), February 27-March 2, 2023, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3578178.3578236

1 INTRODUCTION

Recently, federated high-performance computing (HPC) and cloud systems have been considered viable solutions for providing geographically available computing resources. However, HPC and cloud systems have evolved to meet different application needs. HPC systems have reached the exascale milestone, and users seek new ways to make their workflows leverage all available processing, storage, and network resources, usually consisting of top-end hardware to reduce time to scientific discoveries. On the other hand, warehouse-scale computers (i.e., data centers) base themselves on low to mid-end hardware, and developers are usually concerned with using the resources as efficiently as possible, since a longer response time of a certain service may violate the contract with customers on the quality of service (QoS) while the uncontrolled usage of resources may lead to increased cost and reduced profit margin [2, 11, 19].

The idea of object storage has been widely embraced in cloud computing as it supports cost-effective application scaling and data resilience. Major cloud companies such as Amazon, Google, and Microsoft provide their own object storage services to consumers. As most users of these services are not located within the same network as the storage server, communication between client and server is usually handled through the internet over the HTTP protocol. In the HPC community, CERN is a major player that has already embraced object storage as a solution for storing and retrieving large amounts of data through HTTP [17]. In that sense, the Amazon Simple Storage Server (S3) API [22] is currently one industry standard to access and manipulate object storage data. Even if the S3 API standard is fully proprietary, it is widely supported by many open-source object storage systems, including Ceph [27] and MinIO [18]. Also, popular frameworks like OpenStack Swift [1] provide similar S3-compatible interfaces.

HPC systems are often equipped with high-performance parallel systems to meet the massive I/O needs of large-scale parallel applications. Today, as applications run on pre- and exascale machines, they also generate tremendous data sets that stress the capacity and throughput of parallel file systems. Moreover, the extreme parallelism in these applications results in concurrent file accesses that have exacerbated scalability issues due to the strong consistency in the file systems. The Lustre Parallel File System (LustreFS) [25] is one widely adopted file system on HPC systems. For instance, the latest Top500 in 2022 shows that #1, #2, and #3 - Frontier (USA), Fugaku (Japan), and LUMI (Finland) supercomputers - all provide LustreFS. Therefore, most existing HPC applications have been ported to use the interface of file systems for their I/O. When employing object storage on HPC systems, in order to ease the programming burden for application developers, object storage servers in HPC environments tend to expose a POSIX-like file system interface, which is easily compatible with MPI I/O or libraries such as HDF5 and NetCDF. The drawback of this approach is that some consistency guarantees from POSIX must be ensured, which is usually achieved through imposing serial locks. As the applications continue to scale, so do their I/O needs, and thus, the serialization and consistency semantics increasingly become the bottleneck on

HPC ASIA 2023, February 27-March 2, 2023, Singapore, Singapore

ACM ISBN 978-1-4503-9805-3/23/02.
https://doi.org/10.1145/3578178.3578236

HPC systems [21, 23, 24]. Also, different from the use scenarios in cloud computing, an HPC system has both clients and its object storage server located within the same network.

In this work, we explore using native object storage to support common parallel I/O patterns in HPC applications. The goal of this work has twofold. First, on the transition towards federated computing resources, how to provide converged support to HPC applications to access diverse data stores on geographically distributed cloud storage systems? Second, as shown in existing studies [24], some consistency semantics become unnecessary in HPC I/O use scenarios but are still imposed due to POSIX compliance. Therefore, different from previous works that employ POSIX-like interface atop object storage, e.g., CephFS [15, 29], we leverage MPI I/O as the unified interface in applications and provide a scalable mapping between data objects and I/O operations to bypass the serialization points in file systems. Hence, this work focuses on designing and mapping between representative parallel I/O operations and object storage operations.

To assess the feasibility of our approach, we provide a proof-of-concept implementation that leverages the portability of the S3 APIs and the native object storage on Ceph. Our design principle is to minimize code modifications in existing HPC workloads. We achieve this by providing a library called LibCOS (COS: Converged Object Storage) that intercepts MPI I/O calls as in the existing MPI applications. LibCOS then transparently converts parallel reads and writes into a set of organized data objects through S3 operations. Our research contributions in this paper are summarized in the following:

- We propose a design that uses object storage natively in support of representative parallel I/O operations in HPC workloads.
- Our design enables converged data access from HPC applications to data stores on cloud storage systems in federated computing and data resources.
- We provide a prototype implementation called LibCOS that leverages the portability of the S3 APIs and transparently transforms MPI I/O operations in HPC applications.
- We evaluate LibCOS on a Ceph storage system and an HPC system in three applications, including HACC, BigSort, and IOR.
- We also provide a performance characterization of core S3 operations for enabling parallel I/O operations and their performance tradeoffs.

2 STORAGE INTERFACES

2.1 POSIX I/O

The Portable Operating System Interface (POSIX) is the most commonly adopted I/O interface on HPC systems. The success of POSIX stems from the fact that it provides a set of standards to maintain compatibility between operating systems. The POSIX I/O provides a small set of interfaces that define how a file must be accessed within the operating system – a `open()` command returns the file descriptor, the `read()` fetches data from files into the memory, the `seek()` changes the position of the file descriptor where you start `write()` or `read()` data and then the `close()` operation finally closes the file.

As file descriptors must be managed by the operating system, an overhead penalty is introduced as the system scales [16]. Furthermore, POSIX I/O has strong consistency guarantees for writes. For instance, it may have to block an application's subsequent execution until the system can guarantee that all reads following the write operation will see the update from the last write. While this might not be an issue for small-scale computer systems, it can become a scalability bottleneck for large-scale parallel applications that consist of a large number of processes writing multiple to shared files concurrently.

The other scalability issue in POSIX is associated with metadata management. The filesystem needs to track all of the associated metadata to each stored file, such as the access permissions and last modified date, regardless of whether they are identical in the same directory. When dealing with a large number of files as seen in either HPC or cloud systems, the developers offload the complexity of managing file directories to the operating system. They do not need to track where the files are located (or even its name), and this abstraction becomes one more factor of the overhead in POSIX-like systems.

Parallel File Systems, such as Lustre [4, 25], OrangeFS [3], and Cray's Datawarp [12], either bypass the page cache (used by the operating systems to mitigate the latency penalty for I/O) or relax POSIX consistencies for some I/O patterns or use locking mechanisms to ensure that a file cannot be read by a process while being modified by another one. This comes with tradeoffs, consisting of latency penalties or dirty cache pages.

2.2 MPI I/O

The MPI standard introduced a set of I/O operations in the MPI version 2.0 to accommodate the increasing I/O needs in HPC applications. MPI I/O was created as a solution to address the portable parallel I/O problem that existed at that time with the POSIX programming interface. One advantage of using MPI I/O is to express data partitioning, including non-contiguous data layout, using MPI-derived data types to express common I/O patterns for accessing a shared file to avoid intermediate steps like packing and unpacking data buffers. Using MPI-derived data types to describe well-defined collective I/O operations also enables parallelism and portability. The specification also defines all the interfaces for operations that should be done with the files - from opening to writing, either individually or collectively - and also the consistency semantics. OpenMPI and MPICH are two widely available open-source MPI implementations on most HPC systems. There are also vendor-specific implementations such as IBM's Spectrum MPI, used by the Summit and Sierra supercomputers.

Still, MPI I/O implementations, such as ROMIO, use POSIX I/O, and therefore follow POSIX I/O's consistency semantics. Because MPI I/O is part of the MPI standard, developers with legacy code bases can leverage its portability when migrating their codes to different HPC systems or filesystems that support MPI. In this work, we will concern mainly with the functions `MPI_File_Write`, `MPI_File_Read`, `MPI_File_Write_at` and `MPI_File_Read_at` from the MPI standard. The first two functions are usually associated with operating in a file in its entirety, while the last two are required when one desires to write or read in a specific offset.

2.3 Object Storage

Given the need for scalability for serving multiple concurrent requests - both on reads and writes - at the same time that storing a large number of files is necessary, major cloud players adopted object storage as a solution: as of 2021, Amazon announced that it holds over 100 trillion objects in its servers [1]. However, the original idea of object storage dates from the early 90s by Carnegie Mellon University [7], and the standard is currently maintained by the T10 committee of the International Committee for Information Technology Standards (INCITS).

An object storage system is a way of offloading work from the host kernel while maintaining or improving performance and security [2]. This offloading works by abstracting the user from the necessities of managing a typical local filesystem, as the object storage device will handle all those needs under the hood. The major tradeoff is that it is assumed that the user does not need a hierarchical structure for the organization of the files (called "objects" in this setting), such as folders or filenames, but instead knows directly what he/she wants to access.

As some non-relational databases[6, 14], object storage systems typically work through the establishment of a key-value pair. The "key" is a globally unique identifier (i.e., "tag") for the stored object while the "value" refers to the own binary data. In addition, it is possible to add metadata information to the object such as date, content type and content length.

Figure 1 gives an overview of how some object storage systems work in practice. An object is usually whatever data the user wants it to be, as the data is not interpreted in any way, and it is stored in a flat namespace in memory. An object storage device (OSD) might contain multiple buckets, which can be analogue to root folders, and each of them contains one or more objects. Many object store systems replicate objects in multiple OSDs in order to ensure redundancy and fault tolerance.

2.4 Ceph

Ceph is a distributed filesystem which supports object storage through the RADOS backend [26]. A usual Ceph cluster consists of at least three working nodes with monitors, managers, and object storage devices. The monitors are responsible for ensuring fault tolerance and redundancy whenever one of the nodes is down. The managers are the ones to receive the incoming requests and, through the usage of the CRUSH algorithm [28], determine where the object can be retrieved or put as fast as possible. CRUSH takes into consideration the layout of the cluster (including the data centre, room, row, and rack) to pseudo-randomly map the data to the OSDs, which are the servers whose only purpose is to store data. The OSDs use Ceph's own backend, named Bluestore, to manage data within disks, directly consuming raw block devices or partitions to avoid additional layers of complexity.

Ceph provides access to its system either through a mounted POSIX-like filesystem (named CephFS), through the S3/Swift APIs or through its own RADOS API. In order to be able to expose the first option to its users, which can be done through mounting, Ceph also needs to be provided with a metadata server (MDS) in addition to the three working nodes described above. The metadata server manages the filesystem namespace, coordinating access to the shared OSD cluster.

The major advantage of using the exposed filesystem is the fact that one can use the MPI I/O code without any major changes, but it comes with two drawbacks as well. First, there is the need to provision hardware that could be used for other matters - an MDS for a large cluster will use at least 64 GB of cache memory and two to three CPU cores. Furthermore, at least two MDS must coexist in order to maintain the reliability of the system [3]. The second issue is that, by exposing the filesystem, the user becomes bound to some POSIX-like consistencies as described in Section 2.1. In order to overcome such issues, Ceph also provides the `LAZY_IO` option that relaxes some of these consistency guarantees, in particular by allowing buffered reads/writes when an object is opened by multiple applications on multiple clients.

In high-performance computing systems, a case usage for Ceph can be seen at CERN: the European institute uses it both in hyperconverged infrastructure via CephFS (i.e., storage and processing units at the same system) and for cloud access through HTTP protocol, amounting to a few petabytes of total storage [17]. In industry, Digital Ocean, also a major provider of cloud services, uses Ceph as well albeit mainly for its block storage services, while Red Hat and Canonical provide enterprise-level services for companies that might be interested in adopting or are using this technology.

Another system that is currently being used to leverage object storage for HPC is the Distributed Asynchronous Object Storage (DAOS), currently developed by Intel. As an object storage system, DAOS is built to offload the I/O operations from the kernel to the user space. It uses the Intel Optane technologies - both in memory and SSDs - to build a tiering system where meta- and low-latency data are stored on memory while the bulk data is stored on NANDs and NVMe disks - unlike Ceph, it does not support slow disks such as HDDs. Objects can be accessed either through the native DAOS API (named `libdaos`) or through middlewares such as Hadoop, Spark, MPI I/O and HDF5. A POSIX-like emulation layer is provided by DAOS also under relaxed constraints, in a similar fashion as the one provided by Ceph. The Aurora exascale supercomputer at Argonne National Laboratory (USA) will use DAOS for I/O scalability.

3 REST AND S3

REST is an acronym for "Representational State Transfer" [8], a machine-to-machine (usually, a client-server) architectural interface which includes constraints for statelessness, cacheability, uniform interface, layered system, and code-on-demand. As REST delimits the client and server, it allows those parts to evolve independently while also improving the portability across multiple platforms due to its uniformity. Nonetheless, REST also diminishes the access overhead as the statelessness is a guarantee that a server session will not be stored.

REST web APIs are commonly used atop of HTTP methods (such as PUT, GET, POST and DELETE) to access resources via

[1] Liam Tung. AWS: S3 storage now holds over 100 trillion objects. https://www.zdnet.com/article/aws-s3-storage-now-holds-over-100-trillion-objects/

[2] J. Corbet. Linux and object storage devices. https://lwn.net/Articles/305740

[3] Deploying Metadata servers. https://docs.ceph.com/en/quincy/cephfs/add-remove-mds/

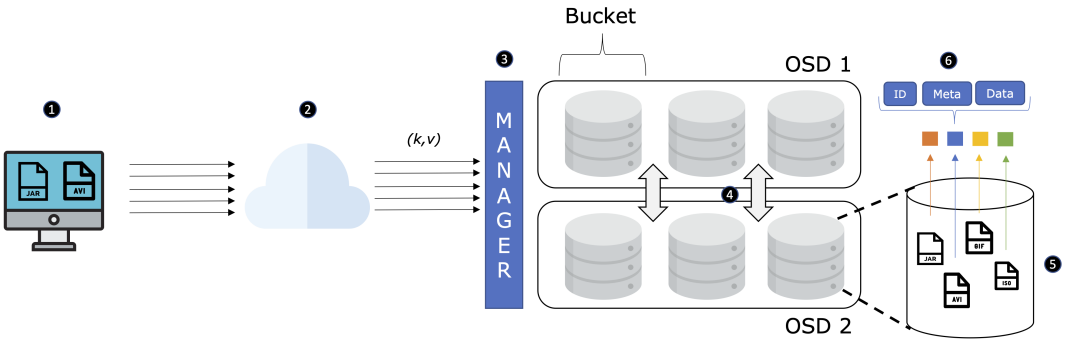

Figure 1: Image displaying the entire path that an object goes through to be stored. (1) An object is sent from the client through the (2) cloud, associated with a (key, value) pair. (3) The request will reach a managing front-end who will distribute the files among the available object storage devices. (4) After some time, the objects will be replicated in different buckets for redundancy. (5) A bucket consists of multiple different objects in a flat namespace, whereas (6) an object consists of an ID, some metadata and the data itself.

URL-encoded parameters and the use of JSON or XML to transmit or retrieve data, which is then parsed at some point. Among cloud services that provide REST APIs are Prometheus (for logging), Jenkins (for continuous integration) and Elasticsearch (for search requests). Social networks such as Instagram and Twitter also provide access to the massive amount of data they have through REST APIs.

Amazon S3 is a REST API that communicates through HTTP and HTTPS. It is not an open specification, but rather defined by the Amazon Web Services company itself, suiting the features implemented by its storage service. Even so, this has not prevented other players to provide compatibility with such interface in order to ease migration and portability between different services.

The S3 specification defines how authentications and request signing occur, as well as the request and response HTTP headers. PUT and GET are among the most relevant S3 operations, respectively related to the upload and download of a certain object. One can also create buckets, list the contents, delete objects and copy or add metadata into those objects as well.

A relevant supported feature by the specification is the multipart upload (MPU), which allows the object storage server to merge different objects into a single one, which works similarly to HTTP/1.1's Multipart Request. For this, one must initiate a request with the server for MPU to retrieve a request ID, upload all relevant parts with this request ID attached along with an associated part number and, finally, send a request to close the uploading process, where the object storage server will then proceed to merge those files in the specified order. While this is an interesting feature that can be used together with asynchronous uploads, a large file that is to be uploaded by different ranks might have some overhead due to the need of sharing the request ID and also the uploading status between them.

4 METHODOLOGY

This paper implements a library that intercepts MPI I/O calls and transforms such calls into S3 operations for object storage servers, uploading or retrieving data directly from/to the memory buffer. Aside from the obvious benefit of the easiness that allows developers to shift the storage paradigm of their application, this approach pursues a strong case for portability: it enables the applications to interact with many diverse object store systems (OSS), some of which may be located over the internet, and also be executed in different computer systems as the developer does not need to concern about the storage whereabouts anymore as the generated maps will take care of it. The second significant obvious benefit is that, given that the developer knows which files he wants to access (either for "read" or "write" operations), a connection made directly to the OSS through HTTP offloads the operating system from the complexity of mounting and handling POSIX-like interfaces, decreasing the overall I/O overhead.

Our read/write approach for object storage servers can be thought of as "one IO call deals with one or more objects", where the object can be either the whole memory buffer or based on how much the current MPI rank intends to write or read on disk. As a consequence of this, the third benefit to be considered is that our approach vastly simplifies all I/O operations under the hood as they will be handled in the same way regardless whether the buffer is being written/read by one or by more ranks in an individual or shared file. Alas, other operations such as overwriting and writing in-between offsets on a file are nothing more than generating a new object and correctly mapping in the metadata.

The remainder of this section will dive into the technical details of how our approach works and how it was implemented.

4.1 S3 Client

Many libraries provide implementations for the S3 APIs in C/C++, which can be further linked to other codes in the C/C++ language or interfaced with Fortran or other programming languages. In particular, during this work, we found that the `libs3` and the `aws4c` libraries are used in the standard I/O micro-benchmarks called IOR. However, the `libs3` library has not been relevantly updated for over four years (i.e., does not support parallel transfers as `libcurl`

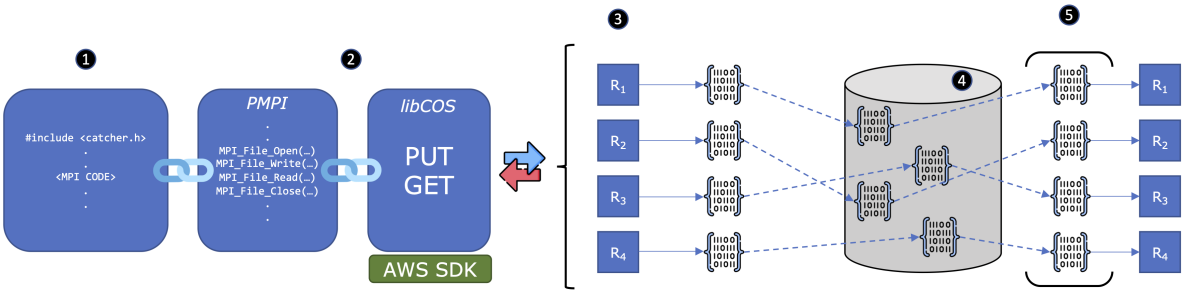

Figure 2: The architecture of the LibCOS library and its main workflow. (1) To make use of the PMPI interface, the original code should only concern with inserting the header for our PMPI API, which contains all the `MPI_File` replacement functions for the original code. (2) Our PMPI API is linked to LibCOS, which does not contain any MPI-related code, and is the only point of communication with the AWS SDK. LibCOS is responsible for all the communication with the object storage system, issuing PUT and GET operations. (3) In an example of individual writes, each rank is writing a different buffer to the object storage system as an individual object, as seen in (4). (5) In the case of a collective read, assuming that the large original file was split into different objects, each rank will also retrieve the object (or parts of it) according to the provided offset and count.

only implemented it in 2019), and the `aws4c` library was lastly updated in 2011.

To obtain the modern functionalities, in this paper, we use the Amazon Web Services Software Development Kit (AWS SDK) as a means to access an object storage server through the S3 API interface. There are a few issues with this approach, however. First, Amazon recently introduced a new S3 Client based on its own Amazon Common Runtime Environment (CRT), in which the HTTP/1.1 and HTTP/2 standards were re-implemented using the C99 standard to make more usage of certain asynchronous features. However, as of October 2022, it is not possible to use the new S3-CRT client due to the lack of features related to changing connection ports [4]. So we decided to use the traditional S3 client based on `libcurl`, which also supports important features from the SDK, such as multiplexing, multipart uploads and asynchronous operations. For the client to access the Ceph storage system, we had to disable the SSL connection (as our Ceph testbed was using an HTTP port instead of HTTPS) alongside the certificate signing while overriding the connection IP to our host on the testbed.

Traditional HTTP methods for object storage are based on PUT and GET operations, which can be used to support MPI IO's write and read operations. Non-blocking MPI I/O operations can also be achieved through asynchronous requests. In the same way, Collective I/O operations can be thought of as multipart uploads, as discussed in the previous section, where multiple parts can be written concurrently by different processes and only merged later on, i.e., deferred synchronization off the critical path. The AWS SDK eases most of the programming burden for those operations, while additionally providing an interface for ranged reads, where you can select the range of bytes you want to read from a certain object. We believe that this ranged operation can be used for emulating `MPI_File_SetView` calls.

[4] "Host cannot be resolved when port is specified in endpoint override for S3 CRT client". https://github.com/aws/aws-sdk-cpp/issues/1844

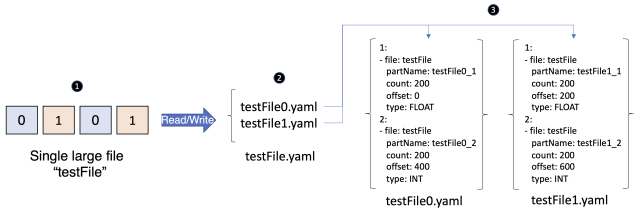

Figure 3: An image that briefly displays how object mapping is performed by LibCOS. (1) From a large buffer in which ranks 0 and 1 are attempting to write different parts of it, (2) they initially check if a global map for the file exists and, if it doesn't, create it and add their own local map file name there. (3) This local map file is appended with metadata every time a rank writes a new object. For read, the rank iterates over each local map listed at the global one and searches for the offset it desires. When both the offset and filename desired match, the associated `partName` is retrieved through a GET operation.

4.2 Implementation

LibCOS is implemented through the usage of the Profiling MPI Interface (PMPI). Fig. 2 provides a diagram with a high-level implementation description. The original MPI-based code (i.e., the application) requires only the additional insertion of the external headers with our code which will overwrite the original MPI calls. One of our two headers consists of only MPI-related code, which is partly responsible for managing the data structures that will be passed to an independent `LibCOS`, related to the second header and which is responsible for handling all the communication between the object storage server and the client as well as interfacing with the AWS SDK for all S3-related operations.

In order to correctly translate the calls between the MPI I/O domain and the object storage server, LibCOS maintain a set of YAML files with associated metadata. There are two types of YAML files to be accessed during a request: a local map generated per rank

during a write phase and contains information such as filename, count, offset of the part and datatype, and the global map, which only consists of the location of all local map files for that file. Fig. 3 shows the structures of such maps as well as how they are accessed during the write or read stages. Additionally, an auxiliary map containing only the number of parts that a rank has generated is also created and used, but it serves only to help the library to know how many parts it should iterate.

Based on the mapping algorithm described above, we implemented the MPI-IO functions. In the `Open` stage, LibCOS will check for the existence of a global map for the associated filename and, if not, will create it while also storing the metadata associated with that initial call. The `Close` function is implemented by LibCOS but does not perform any major role as the idea of ending access to a local file does not exist in this context. Other calls, such as `MPI_File_Get_Size`, can be later implemented by checking the associated local metadata instead of querying the entire object at the server.

When dealing with either `MPI_File_Write` or `MPI_File_Write_at`, LibCOS will write all the metadata into a local map file. For simplification purposes, a write is considered a write_at without offset. Based on the datatype provided by the call, the memory buffer is then converted into a string stream and then an S3 PUT operation is issued for that string stream. When the outcome is listed as successful, LibCOS returns success and the application can resume.

During the execution of a `MPI_File_Read` or `MPI_File_Read_at`, LibCOS iterates over the global map file (which has the same name as the provided filename from `open`) and, by iterating in all listed map files, it searches for the desired offset and retrieves the associated part name. A read is considered a read_at for a part name with an offset equal to zero. Henceforth, a GET operation is issued at this point and, upon conclusion, the obtained string stream is converted to the datatype written in the YAML file. The obtained memory buffer is transferred to the buffer address provided by the original call. If everything succeeds, the application resumes normally.

The connection to the object storage server is exclusively done during the write or read stages. During such stages, an additional YAML file - `server.yaml` - is verified to retrieve data such as server IP, the bucket in which data should be accessed and the credentials (access key and secret key). If the file does not exist in the same place as the application, the write or read operation will not be able to proceed.

As the usage of LibCOS requires only the addition of the header files and also the library linking during the compilation stage, there are some limitations introduced by this model. First, the AWS SDK imposes restrictions on how it can be used and, due to this, the SDK must be set up, started and stopped for every write or read call. Establishing a handshake with the object storage server every time, as opposed to keeping a persistent connection, brings an overhead that is later examined in this paper. Another limitation is that, in our model, all PUT and GET operations use a string stream, which means that the data must be converted back and forth between different datatypes. Although the convert stage increases as much as the data, our analysis has shown that the total time is mostly dominated by the transfer time.

The need to convert data brings another important limitation: while there is an explicit conversion between some of MPI Types (e.g., `MPI_SHORT`, `MPI_FLOAT`, `MPI_CHAR`) and C datatypes (`short`, `float`, `char`), there is not a specific correspondence for (`MPI_BYTE`). This imposes that the developer must create an appropriate type conversion to accurately represent the data it is writing or reading. This is further discussed in Section 5.2.

5 EXPERIMENTAL SETUP

5.1 Hardware Infrastructure

In this study, we use a supercomputer (named "Dardel"), located at KTH Royal Institute of Technology, and a single-node computer system (called "cloud testbed"). Dardel is an HPE Cray EX machine running a custom version of SUSE Linux 15.1 with 554 nodes, containing a dual AMD EPYC 2 (Rome) with 128 processors in total per node. The memory within nodes ranges from 256 GB to 2 TB DDR4, but this work used only nodes with 256 GB ones. The supercomputer uses the Lustre Parallel File System for a total of 15 petabytes of storage. Network-wise, it has all nodes interconnected with an HPE Slingshot network using a Dragonfly topology, with up to 100 Gbps of connection internally and practical measurements reveal that its connection to the internet is about 3 Gbps for downloads and 600 Mbps for uploads.

The cloud testbed has a single Intel i7-7820X processor, with 16 cores in total, and 32 GB of DD4 memory at 2133 MHz. In terms of storage, the system contains an Intel Optane SSD 900p with 480 GB, a Kingston UV400 SSD, 2x Seagate Barracuda with 2 TB each and a Samsung EVO NVMe driver with 1 TB, which the operating system (Ubuntu 22.04) is running. This cloud testbed also includes an Intel I219-V single-port 1 gigabyte Ethernet controller. The system is connected via a Gigabit Ethernet connection to the HPC testbed.

5.2 Applications

This work will present results based on two applications derived from the CORAL-2 suite, provided by the Lawrence Livermore National Laboratory (USA), and the IOR benchmark. It is important to note that, as the AWS SDK recommends the usage of CMake to link with applications (due to the large number of files involved), we created a CMakeFile for each of them which seamlessly built and linked them to LibCOS. Since applications like HACC and IOR are complex and span thousands of lines, we opted to use stripped-down versions of them while conserving the I/O patterns.

The Hardware/Hybrid Accelerated Cosmology Code (HACC) is an application used to "simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe" [10]. Here, we use a stripped-down version of HACC named "HACC-IO"[5]. This stripped version uses the pattern of each rank generating a full independent particle file, and each particle file will be written (or read) 10 times at different offsets. The particle size is the only parameter that one can set when starting the application. The data written into the file consists of arrays with different datatypes and the header, which is written as `MPI_BYTE` but is expressed as a C++ `int64_t` type (and thus, the writing/reading operation for that had to be particularly implemented in LibCOS).

The BigSort benchmark is intended to sort a large number of 64-bit integers and it contains multiple mini-applications within it

[5] "HACC-IO": https://github.com/glennklockwood/hacc-io

that perform different stages of the overall process. One of those applications is named Genseq, which writes to a large, shared file a specified sequence of numbers, in sequential order. It also takes the DRAM allocation as a parameter to manage how much the application can write at each time in the file. Even though it is generating numerical sequences, the data is ultimately written as `char` type.

IOR is a widely used I/O benchmark tool and allows a high degree of customization on how one desires the I/O operations to be emulated. IOR supports a large number of APIs - POSIX, MPI I/O, MMAP, HDFS, and others - through its AIORI backend, and allows a high degree of customization when starting the application: for example, one can choose the number of blocks and segments, the transfer size, the API to be used, or if the I/O operations should be executed in a sequential or random pattern. In this work, we used IOR to write a shared single file in a random pattern (as opposed to the sequential one of Genseq) while keeping constant the number of segments and block size.

We stripped IOR to the bare minimum of its compilation requirements to run the MPI I/O backend within our CMakeFile. However, since the AWS SDK is written in C++ and IOR is in C, we had to do some minor changes to the original code to be able to compile the application with a C++ compiler and link it with our library. These modifications were essentially solving issues related to code that is conforming with GCC but not with g++ and removing the `MPI_File_Get_Size` call as this was not implemented in LibCOS. For HACC and BigSort, the only changes were the addition of the LibCOS headers into the code.

5.3 Configuration of Object Storage Server

Our object storage server consists of a Ceph cluster running on the cloud testbed. As the object storage server requires a cluster with at least three worker nodes and one coordinator node to properly run, this setting was emulated through Minikube, which implements a local Kubernetes cluster. The emulation of such nodes can be done either within a virtual machine (e.g., KVM2, QEMU) or within a container (e.g., Docker or Podman). Each pod has its own IP within the cluster. However, this approach does not allow one to access those nodes externally (i.e., from the internet) and, in order to bypass this issue, Minikube also provides the bare-metal option and, by exposing a port of Ceph's manager node, it is possible to access from anywhere through the cloud testbed's IP. The tradeoff is that Minikube supports only 1 node in the bare-metal configuration, so each pod is effectively emulating an entire node. Fig. 4 shows the architecture of this configuration.

Ceph was installed with Rook, a storage operator for Kubernetes. In our configuration of Ceph, there's a single manager, a single monitor and a single object storage daemon, for a total of three Kubernetes pods. Each pod has effectively reserved 2 CPU cores and 256 MB of RAM. This effectively removes any redundancy from our system. Additionally, no metadata servers are used as we did not intend to deal with the CephFS interface. Furthermore, as Ceph does not enforce any types of disk to use (i.e., it will accept either SSDs or traditional hard disks), the object storage consisted only of the Intel Optane 900p SSD (480 GB) to enforce the highest possible I/O throughput.

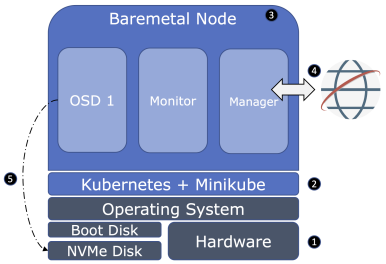

Figure 4: A conceptual illustration of the configuration of our cloud testbed. (1) The hardware includes a booting disk, used by the operating system, and an empty NVMe disk, without any filesystem within. (2) On the user space layer, Minikube deploys Kubernetes in a single node (multinode is not supported on bare-metal) and (3) within the node, multiple pods with different applications for Ceph are started. Each pod has an internal IP that can be used for communication between pods, but only (4) the manager pod is exposed to the internet and receives requests. (5) When a request arrives, the manager redirects to the OSD pod, which has direct access to the NVMe disk.

6 RESULTS

The overall idea of all the experiments, which will use the applications described earlier, is to transfer or receive directly from/to the memory buffer to/from an object storage server. While both host and server are very physically proximal, they do not share the same network, but rather connect through the internet, which puts an enormous delay when compared in performance with local MPI I/O operations.

That said, the results presented here should be seen through the lens of i) viability, (i.e., the approach works, even if it is not in an ideal setting for high performance), ii) scalability and iii) portability (as the storage is located at a different server, which makes a case for Federated HPC storage). The standard deviation was very small in the plots shown in this section, hence it is not displayed.

6.1 Performance Characterization of S3 Operations

As it was stated in Section 4.2, there is an implicit overhead of having to establish a connection with the object storage server at every operation. In this context, we want to measure the overhead caused by having to set up the initial connection to the cloud testbed instead of using a persistent one. The experiment carried out for investigation is setting a synthetic application for PUT/GET directly at LibCOS and uploading the same data over and over without closing the connection (but changing the associated part name to avoid possible effects from overwriting in the server). Each operation is performed ten times before closing the connection. The execution times shown were collected through the usage of `ctime` library and the timers were in place right before and after the PUT/GET operations. This experiment was executed using only one rank.

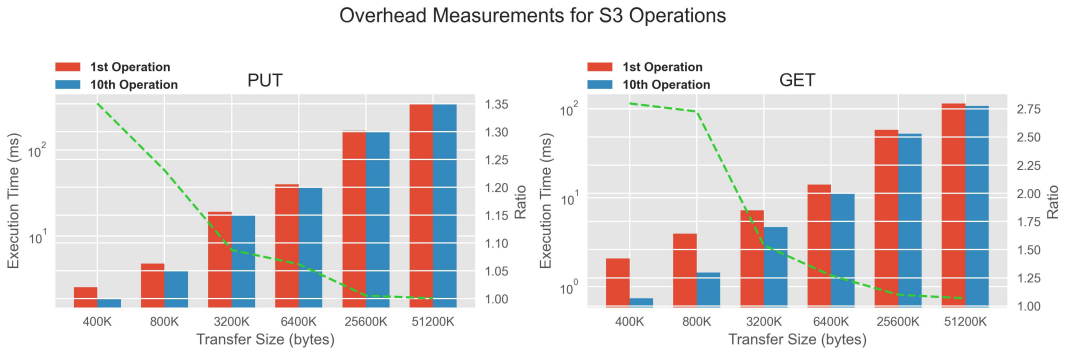

Figure 5: Plots measuring the overhead for (left) PUT and (right) GET operations with varying transfer sizes. The dashed green line is the ratio between the execution time of the 1st and the 10th operations.

As shown by Fig. 5, there is a clear difference between the first and the other connections - mainly because there is some overhead due to setting this initial connection is avoided later on. However, it is possible to verify that, as the size of the buffer increases, the overhead becomes very minor in relation to the total execution time as the latter starts to be increasingly dominated by the transfer time.

6.2 Independent Parallel I/O Patterns

An independent parallel I/O pattern means that each rank will take care of its own file, without any intervention from other ranks. That is what happens at HACC. The times used here are reported by HACC-IO's own timer and concern the total execution time - which includes either 10 reads or 10 writes regardless of the chosen particle size. We performed a weak scaling analysis and compared the results among multiple ranks, as can be seen in Fig. 6.

While it is clear that execution times are worse when you increase the number of ranks, there are two things to consider: first, that HACC's code includes some non-MPI I/O operations such as broadcasting and scattering, which significantly affect execution times as you increase the number of ranks. Second, as the results are related to weak scaling, one should consider the trade-off between having a ten to twenty percent slower execution time to the benefit of having eight times more data written or read from the server.

6.3 Collective I/O Patterns

In this pattern, the applications write or read to/from a single file that is accessed by multiple ranks. Genseq solely focuses on writing operations and splits the workload between the available ranks. The left side of Fig. 7 shows the obtained results for weak scaling and strong scaling, and also for the scale of PUT with a varying memory buffer size.

The major points to notice are that the S3 curve in Fig. 7.a is essentially straight, which means that the time stays mostly stable (with small variations, but the scale does not allow us to observe it) when both the number of ranks and the write buffer increase. While timescales are very different in relation to the MPI's, the MPI's curve behaviour follows a very different pattern. Fig. 7.b follows the pattern of a logarithmic curve with the distribution of work in Genseq, which means that the workload can be equally distributed among multiple ranks and this will perform a roughly similar decay in execution time, and this is because ranks are mostly independent and do not need to wait for each other to keep performing operations (unless it is necessary to resume the application). Fig. 7.c shows that, as it was seen in HACC, the execution time decreases when you increase the number of ranks, while also reinforcing the analysis of Fig. 7.b.

Also within the same Collective I/O pattern, the results obtained by using LibCOS with IOR's MPI I/O API are shown on the right side of Fig. 7. While Genseq writes the content in sequential offsets (i.e., rank 1 will write from offset 0 to 200, then 200 to 400 until it reaches all supposed to write), IOR can be set up for writing or reading in random offsets and that is the major difference in the pattern of the two applications. We use IOR to write and read files up to 1 gigabyte in size (as this is also the network speed between the HPC testbed and the cloud testbed) and perform multiple scaling results for up to 16 ranks and different file sizes. Fig. 7.d displays the weak scaling in a better scale than the one by Genseq. Ranks are writing 16 MB at a time in multiple chunks with different offsets, which means that, for every chunk, a new connection to the S3 server needs to be established and the file sent. In this case, we attribute the heavy penalty of this delay to our object storage server which is not able to handle a large number of concurrent connections at the same time - as seen in Section 5.3, our object storage manager and daemon consist only of 2 CPU cores each, which may be insufficient to process the number of concurrent connections IOR is creating.

Figs. 7.e and 7.f display the already expected behaviour based on the previous experiments and shows that the ordering of the writing does not matter in our implementation.

7 RELATED WORK

There are several works integrating HPC and object storage, or HPC and REST. The FirecREST API [5] is an attempt to externally access and manage resources in an HPC system, with a focus on i) submitting processing jobs, and ii) downloading and uploading large files through the Swift/S3 API, but there aren't any performance benefits claimed on the paper aside the portability and easiness of use. Similarly, since 2020, SLURM has been providing a REST API (although not web-enabled as FirecREST) that allows clients to

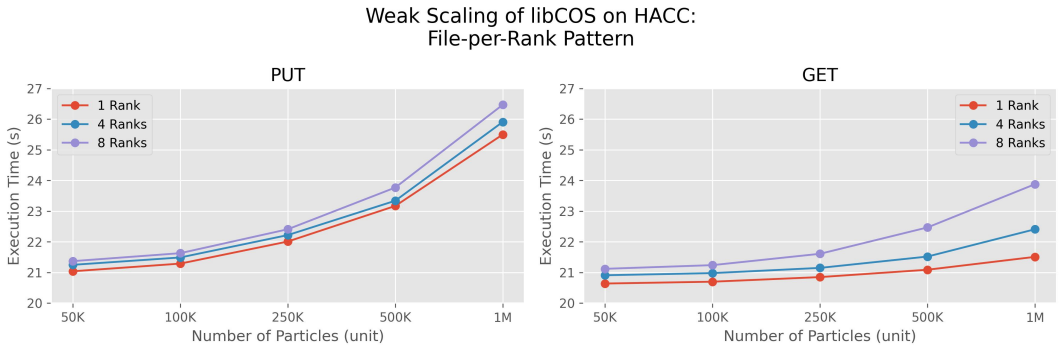

Figure 6: Plots that show the weak scaling results for (left) PUT and (right) GET operations on HACC using libCOS. Eight ranks are effectively writing or reading eight times more data in relation to the first.

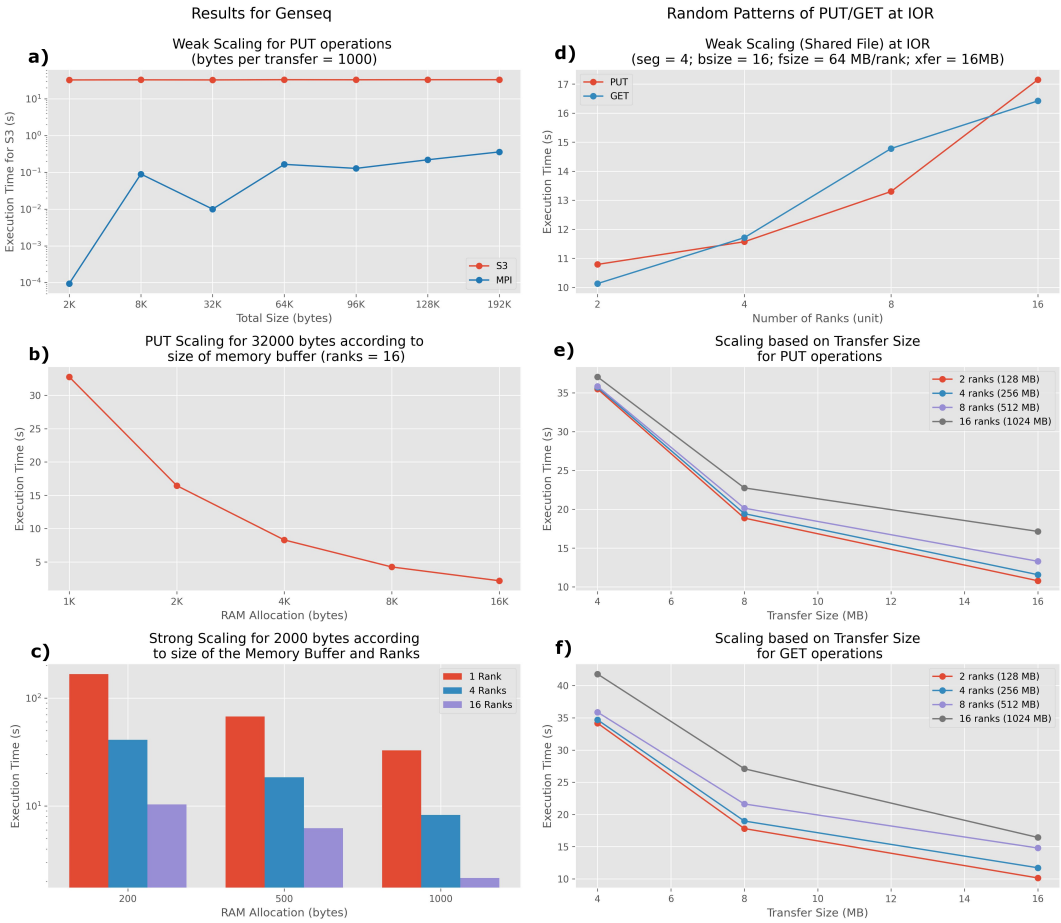

Figure 7: A panel displaying obtained results for Genseq and IOR applications. (Left) Results for Genseq, where a) does a brief comparison of execution times between MPI and MPI over S3 through LibCOS, b) displays the scaling behaviour for PUT operations with 16 ranks and varying sizes of memory buffer, and c) illustrates multiple barplots that allows one see the strong scaling according to the RAM allocation of Genseq. (Right) Plots for results obtained from LibCOS and IOR, with random ordering in a shared, single file. d) Weak scaling for multiple ranks and increasing file size, e) a scaling with varying transfer sizes for PUT, and f) same as previously but for GET operations.

communicate with the daemon for operations such as submitting or cancelling jobs and reading data (i.e., partitions, users) from a cluster.

A paper in 2018 [15] compares the performance of RADOS (through Ceph), DAOS and OpenStack Swift for HPC in different contexts - namely, "sequential and large writes", "random and small writes", and "sequential and large reads". This was done through the usage of HDF5's Virtual Object Layer (VOL) plugin, which allowed the authors to interface existing applications with such systems. It was shown that i) object stores have better scalability than POSIX filesystems, ii) that RADOS has the best performance for partial reads and writes, and iii) that DAOS has "outstanding performance" for the presented benchmarks. The major issue with the authors' approach in this paper is that, in the case of Swift (an HTTP-based REST API), they resorted to building the VOL plugin using Python over a C interface, as the native C version had over five years without significant updates - and this means that significant newer features of the HTTP protocol were not evaluated in this work. Also, given that Python is an interpreted language, interfacing it with C would cause unacceptable overhead in HPC environments. Finally, the Swift testbed had also a low network limit of 1 Gbps in comparison with the DAOS/Ceph testbeds and hence would also suffer performance degradation when evaluated.

A more recent paper [20] evaluates DAOS with applications that interface with the HDF5 library, also using the VOL plugin. The biggest advantage of the presented approach is the fact that only a few lines had to be modified or included for the applications to be able to use HDF5 at DAOS. It takes advantage of DAOS-specific features such as asynchronous I/O, independent object creation and maps to show performance gains.

There are also a 2022 evaluation in relation of the viability of S3 at Sandia National Laboratories[13]. The reached conclusion is that the slowness of S3 in comparison to traditional HPC storage is a trade-off for accessibility, especially when dealing with historical data (i.e., archiving and exploiting).

It is also worth mentioning that the HDFGroup also developed an HDF5 REST API specification [9] which is integrated into some network-based services, such as HSDS and H5serv.

8 DISCUSSIONS AND CONCLUSIONS

In this work, we proposed a native object storage support in MPI I/O to address diverse data stores across converged HPC and cloud systems. In particular, we presented an approach to leverage the native Ceph object storage without imposing POSIX constraints and the portability of S3 API operations in MPI I/O interfaces. Our evaluation of common I/O patterns through different applications on a converged HPC and cloud testbed shows that an object storage native implementation of MPI I/O can have improved scalability at increased concurrency in I/O operations. Furthermore, while there is also a latency associated with accessing cloud storage outside the HPC system, our characterization study shows that the latency is amortized at a relatively large transfer size or when the connection between the HPC and cloud systems is persistent - a model that can be adopted when using object storage directly from the application itself.

The most important conclusion derived from the results we presented here is that the mapping algorithm effectively equalizes the collective and independent operations — working with smaller chunks is as effective as working with a single, large, and complex file. Still, the former has the benefit of reduced complexity. Leaving such mapping to the object storage API eases the burden for the developers and allows them to think in "MPI terms" while writing to object storage systems under the hood.

Furthermore, as there is no enforcement of POSIX consistency nor the handling of file descriptors by the operating system, the PUT and GET operations can be asynchronous when each process just writes or reads an independent chunk. The offloading of the I/O operations to the object storage server transfers the possible bottlenecks from the local storage disk to the network bandwidth, and in cases where the network is not saturated, allows a good scaling as shown in Section 6.

Lastly, the results show the feasibility of accessing data outside a local HPC cluster. Even though there is a heavy delay associated with the connection to the internet, we assume that in peta/exascale settings, this delay is very minor in comparison to the total processing time of the application, but has the benefit of enabling whole new possibilities of data usage by scientists and developers.

As a prototype implementation, and due to limited hardware capacity, e.g., network bandwidth from the HPC testbed to our cloud testbed. However, the limited connection bandwidth also represents realistic converged HPC and cloud systems in federated resources. Moreover, we focused on the high-level design that is generally applicable to MPI applications and object storage that supports S3 interfaces. Our future works will aim to integrate even more with the full MPI I/O standard, make use of multiple threads to write data from the same buffer and support retrieval or insertion of data from/to multiple federated object storage servers.

Nevertheless, we believe this work is a step towards the native adoption of object storage in HPC workloads and also easing the transition towards emerging federated HPC and cloud resources. Also, as the adoption of object storage systems increases in the HPC domain, as motivated by several existing efforts like Intel's DAOS, it is expected that more applications will be motivated to leverage the scalability that the object storage systems might provide.

ACKNOWLEDGEMENTS AND REPRODUCIBILITY

Financial support was provided by the European Commission under the agreement No. 955811 ("I/O Software for Exascale Architecture"). The authors would like to thank the PDC Center for High Performance Computing for providing the Dardel testbed used in this work.

REFERENCES

[1] Joe Arnold. 2014. *Openstack swift: Using, administering, and developing for swift object storage.* " O'Reilly Media, Inc.".

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.

[3] Michael Moore David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles Sam Sampson, Shuangyang Yang, and Boyd Wilson. 2011. OrangeFS: Advancing PVFS. In *USENIX Conference on File and Storage Technologies (FAST)*.

[4] Peter Braam. 2019. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[5] Felipe A Cruz, Alejandro J Dabin, Juan Pablo Dorsch, Eirini Koutsaniti, Nelson F Lezcano, Maxime Martinasso, and Dario Petrusic. 2020. FirecREST: a RESTful API to HPC systems. In *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*. IEEE, 21–26.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[7] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. 2005. Object storage: The future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*. IEEE, 119–123.

[8] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

[9] Gerd Heber. 2013. RESTful HDF5-Interface Specification-Version 0.1. (2013).

[10] Katrin Heitmann, Thomas D. Uram, Hal Finkel, Nicholas Frontiere, Salman Habib, Adrian Pope, Esteban Rangel, Joseph Hollowed, Danila Korytov, Patricia Larsen, Benjamin S. Allen, Kyle Chard, and Ian Foster. 2019. HACC Cosmological Simulations: First Data Release. *The Astrophysical Journal Supplement Series* 244, 1 (sep 2019), 17. https://doi.org/10.3847/1538-4365/ab3724

[11] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[12] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. 2016. Architecture and design of cray datawarp. *Cray User Group CUG* (2016).

[13] Todd Kordenbrock, Gary Templet, Craig Ulmer, and Patrick Widener. 2022. Viability of S3 Object Storage for the ASC Program at Sandia. *Sandia National Laboratories* (2022).

[14] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[15] Jialin Liu, Quincey Koziol, Gregory F Butler, Neil Fortner, Mohamad Chaarawi, Houjun Tang, Suren Byna, Glenn K Lockwood, Ravi Cheema, Kristy A Kallback-Rose, et al. 2018. Evaluation of HPC application I/O on object storage systems. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, 24–34.

[16] Glenn Lockwood. 2017. What's So Bad About POSIX I/O? https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/.

[17] Luca Mascetti, Maria Arsuaga Rios, Enrico Bocchi, Joao Calado Vicente, Belinda Chan Kwok Cheong, Diogo Castro, Julien Collet, Cristian Contescu, Hugo Gonzalez Labrador, Jan Iven, et al. 2020. Cern disk storage services: report from last data taking, evolution and future outlook towards exabyte-scale storage. In *EPJ Web of Conferences*, Vol. 245. EDP Sciences, 04038.

[18] Inc. MinIO. 2021. MinIO. High Performance, Kubernetes Native Object Storage. *MinIO*. (2021).

[19] Eric Schurman and Jake Brutlag. 2009. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. oreilly.

[20] Jerome Soumagne, Jordan Henderson, Mohamad Chaarawi, Neil Fortner, Scot Breitenfeld, Songyu Lu, Dana Robinson, Elena Pourmal, and Johann Lombardi. 2021. Accelerating hdf5 i/o for exascale using daos. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 903–914.

[21] Houjun Tang, Suren Byna, François Tessier, Teng Wang, Bin Dong, Jingqing Mu, Quincey Koziol, Jerome Soumagne, Venkatram Vishwanath, Jialin Liu, et al. 2018. Toward scalable and asynchronous object-centric data management for HPC. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 113–122.

[22] Jinesh Varia, Sajee Mathew, et al. 2014. Overview of amazon web services. *Amazon Web Services* 105 (2014).

[23] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. 2005. Providing Tunable Consistency for a Parallel File Store.. In *FAST*, Vol. 5. 2–2.

[24] Chen Wang, Kathryn Mohror, and Marc Snir. 2021. File system semantics requirements of HPC applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 19–30.

[25] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. 2009. Understanding Lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep* 120 (2009).

[26] Sage Weil, Andrew Leung, Scott Brandt, and Carlos Maltzahn. 2007. RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. 35–44. https://doi.org/10.1145/1374596.1374606

[27] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 307–320.

[28] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 31–31.

[29] Patrick Widener and Matthew Curry. 2020. *CephFS experiments on stria. sandia. gov*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).