Accelerating HDF5 I/O for Exascale using DAOS

Jerome Soumagne, *Member, IEEE,* Jordan Henderson, Mohamad Chaarawi, Neil Fortner, Scot Breitenfeld, Songyu Lu, Dana Robinson, *Member, IEEE,* Elena Pourmal, and Johann Lombardi

Abstract—The Hierarchical Data Format 5 (HDF5) has long been defined as one of the most prominent data models, binary file formats and I/O libraries for storing and managing scientific data. Introduced in the late 90s when POSIX I/O was the standard, the library has since then been continuously improved to respond and adapt to the ever-growing demands of high-performance computing (HPC) software and hardware. Given the limitations of POSIX I/O and with the emergence of new technologies such as object stores, non-volatile memory, and SSDs, the need for an interface that can efficiently store and access data at scale through new paradigms has become more and more pressing. The Distributed Asynchronous Object Storage (DAOS) file system is an emerging file system that aims at responding to those demands by taking disk-based storage out of the loop.
We present in this paper the research efforts that have been taking place to prepare the HDF5 library for Exascale using DAOS. By enabling and defining a new storage file format, we focus on the benefits that it delivers to the applications in terms of features and performance.

Index Terms—Parallel I/O, Distributed File Systems, Data Storage Representations, Object Representation.

1 Introduction

DATA management and storage has always been a central point of scientific workflows. Over the years, scientific simulations and experiments have largely been focused on increasing the resolution of models and precision of sensors, resulting in an increased demand for I/O bandwidth as the amount of data generated has exponentially grown larger. In addition, the increased complexity of applications, where multi-physics and code coupling has become a norm, has exacerbated that demand, making I/O a critical part of the various steps of the scientific workflow.

The HDF5 library [1], [2] is one of the most commonly used I/O libraries and binary file formats in the HPC community. The native HDF5 file format was created in the late 90s and was designed to handle data stored on traditional POSIX file systems as efficiently as possible, given the constraints that were imposed by the technology of the time. This file format has been extended and improved over the years, however the constraints of POSIX I/O semantics and disk-based storage fundamentally limit the I/O potential of the HDF5 library, similar to other monolithic file systems that have occupied the high-performance computing space for many years [3], [4].

DAOS, the Distributed Asynchronous Object Storage [5], is an open-source software-defined scale-out object store that provides high bandwidth, low latency, and high IOPS storage containers to HPC applications. It enables next-generation data-centric workflows combining simulation, data analytics and A.I. Unlike earlier storage software stacks that were primarily designed for rotating media, DAOS is built from the ground up to exploit new non-volatile memory (NVM) technologies. DAOS offers a shift away from the traditional POSIX storage model designed for block-based, high latency storage to one that inherently supports fine-grained data access and unlocks the performance of the next generation storage technologies.

From the early prototype evaluation that was presented in [6], several components have now matured and evolved. This paper makes the following contributions: (1) presents the new paradigms that have now been put into place to shift the HDF5 library from a POSIX-oriented model to an object store model that relies on DAOS (2) highlights a set of new features that were so far not possible to implement efficiently on top of POSIX I/O (3) evaluates each of these features individually on HPC hardware and emphasizes their necessity through a concrete use case. The features presented in this paper are essential for applications to maximize I/O throughput at extreme scale and are geared towards applications with a high degree of parallelism, effectively removing the I/O contention that can be seen on today's POSIX types of storage—this is most notably seen on metadata intensive applications.

In Section 2, we first discuss related work as well as object store models and I/O technologies that have also taken a step away from the standard POSIX file I/O and block-based model. In Section 3, we focus on the original design of the so called native HDF5 file format and on the reasons why it is no longer fit for use at Exascale, highlighting the limitations that were directly inherited from POSIX block-based I/O. We also focus more in depth on DAOS and on the new semantics that it introduces to circumvent these limitations. In Section 4, we present our HDF5 connector solution as well as its benefits and novel features for application users. Section 5 evaluates standalone performance while Section 6 illustrates features and performance in a real world application. We present conclusions and future work in Section 7.

- *J. Soumagne, J. Henderson, N. Fortner, S. Breitenfeld, S. Lu, D. Robinson and E. Pourmal are with The HDF Group, Champaign, IL, 61820.*
- *M. Chaarawi and J. Lombardi are with Intel Corporation, Extreme Storage Architecture & Development, Santa Clara, CA, 95054. .*

Manuscript received 24 Feb. 2021; revised 15 Jun. 2021; accepted 13 Jul. 2021.

2 Related Work

Common HPC storage systems have relied for several years on object-based semantics [3], [7], though they have never been exposed or intended to be used by storage middleware such as HDF5. They are instead usually hidden behind the Linux file system framework and only accessed through POSIX I/O.

To improve scientific workflows and work around non-suitable storage layouts, solutions such as PLFS [8] have been proposed, effectively inserting themselves as an intermediate storage layer between the application and the underlying file system, thereby exposing new types of I/O semantics to the application (in this case optimizing small non-aligned writes). This type of solution has been directly integrated into HDF5 [9] and proposed as a new storage format for the application.

Full-fledged object store solutions have also emerged as a replacement for POSIX file systems, offering a different approach as to how data can be accessed. For instance, RADOS [10], Amazon S3 [11], and OpenStack Swift [12] all offer object-centric semantics though they primarily target cloud-based solutions and are not directly suitable for HPC use. Amazon S3, for instance, does not allow concurrent writes to shared objects, partial overwrites, or native non-http access. RADOS, OpenStack Swift and Amazon S3 have, however, also been interfaced and evaluated with HDF5 [13], [14].

As a middle ground to both of these approaches, Proactive Data Containers (PDC) [15] proposes, with an object-centric approach, to insert itself as an intermediate layer of abstraction between the application and the underlying file system, effectively exposing object-centric semantics. By doing so, it is capable of taking advantage of the underlying object stores when available or conventional file systems such as Lustre to propose unified access methods that are based on the notion of objects. This approach has also been integrated with the HDF5 library [16].

Finally, to facilitate data management and to introduce new types of workflows that are directly adapted to the applications, solutions such as Mochi [17] and Faodel [18] have been proposed to create a collection of composable services that can be easily deployed and tailored to the application needs, effectively exposing new types of I/O semantics for the application that can either rely on K/V objects or other forms of data representation. DAOS also relies on some of Mochi's components for its implementation.

3 Background

This Section focuses on both the HDF5 library and DAOS to provide sufficient background and understanding on our approach and why it is essential for achieving I/O at extreme scale.

3.1 HDF5

As mentioned in introduction, the HDF5 library was initially designed around POSIX I/O and block-based storage. However, the HDF5 data model itself was designed as an abstract object-based model, which defines a few primary entities: files, groups, datasets, datatypes, and attributes that can be used to build complex data representations and associations. Based on this model, the library encompasses a set of high-level I/O primitives aimed at facilitating data management and organization of those objects within a file or a collection of files, creating links between them.

3.1.1 Block-Based File Format and Limitations

While the HDF5 API is high-level and object-based, the resulting POSIX-based storage is only viewed as a stream of bytes that are sequentially accessed; and the only primitives that POSIX offers are block-based (e.g., `pwrite()`). In that context, to be able to store a hierarchy of objects, the library has to define a file format that represents how these objects are mapped to disk as a block sequence. A simplified representation of that file format is depicted in Figure 1. HDF5 files are comprised of both internal file metadata (i.e., object headers, heaps, object tables, etc.) and raw data. Headers are used to locate objects within the file as well as embedding the extra metadata that is attached to those objects, such as dataset dimensions and object creation order.

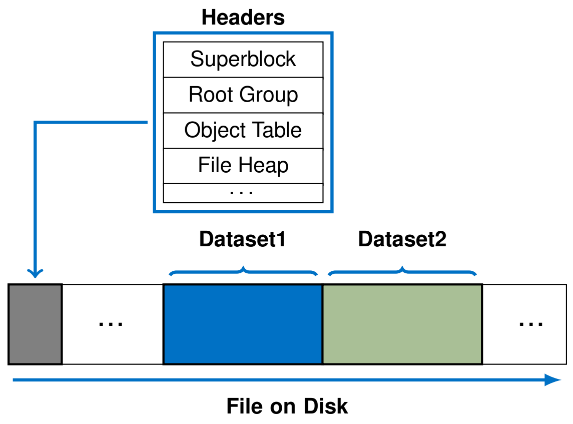


Fig. 1: Simplified representation of an HDF5 file with two datasets on block-based storage using POSIX I/O.

This format on disk is a depiction of the linear address space that is imposed by POSIX I/O and directly inherited from disk-based storage addressing. Consequently, when writing multiple objects to a file, the required space that is necessary to store these objects must be determined sequentially. This issue is exacerbated when writing in parallel from multiple processes to a single shared file. All processes must first coordinate their writing by exchanging information on the block's address offset that they are going to write before the actual write can happen. This limitation has historically reflected on the semantics of HDF5 when doing parallel I/O: all object operations that affect the metadata of the file (e.g., object creation, deletion) must be done *collectively*, whereas nothing from an API and object-based perspective would in principle prevent users from doing so.

As a workaround, a frequent technique has been to instead do file-per-process I/O or subfiling, effectively removing synchronization constraints. However, this also increases metadata I/O to the file system, which may be put under heavy stress, particularly when a large number of processes are performing I/O on a similarly large number of files. This shift of I/O responsibilities to layers underneath HDF5 ultimately does little to fundamentally solve an application's data throughput problems and the application is commonly forced to adopt I/O strategies that do not necessarily reflect its data model.

3.1.2 *Virtual Object Layer*

In HDF5 1.12.0, the HDF5 library was restructured to use a new Virtual Object Layer (VOL), which separates the abstract HDF5 data model from the underlying storage system. The VOL maps HDF5 API calls that pertain to storage (e.g., `H5Fcreate()`, `H5Dread()`) to callbacks in a user-specified VOL connector that implements a particular storage scheme, one of which is the *native* HDF5 connector that implements the legacy HDF5 file format. Crucially for HPC applications, this architecture allows portably mapping non-POSIX storage systems to the HDF5 API, often with few, if any, code changes to existing applications.

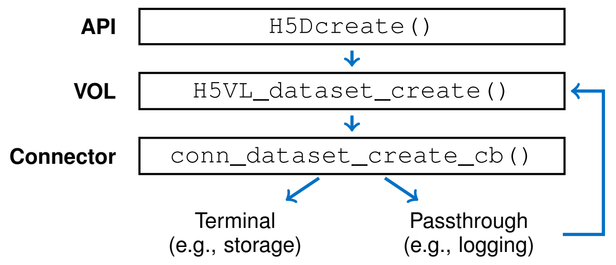


Fig. 2: HDF5 API calls are redirected through the Virtual Object Layer to the connector implementation.

As represented in Figure 2, calls can be redirected to two types of VOL connectors: terminal or pass-through. A number of VOL connectors have already been implemented [9], [13], [16] following that logic. We focus in this paper on the terminal type of connectors that directly interact with the storage. One of the main and obvious advantages of this layer is that it enables a direct mapping of the fundamentally object-based HDF5 API to an object store, carrying the metadata of the HDF5 objects down to the storage layer. Each VOL connector is responsible for storing the data in the manner it wishes, defining new formats that are optimized and dedicated to these types of storage.

3.2 DAOS

DAOS is an open-source object store designed for massively distributed NVM devices, taking advantage of next-generation NVM technology, such as Storage Class Memory (SCM) and NVM Express (NVMe). It presents a key-value storage interface on top of commodity hardware that provides features such as transactional non-blocking I/O, advanced data protection with self-healing, end-to-end data integrity, fine-grained data control, and elastic storage, optimizing performance and cost. Unlike traditional burst buffers [19], DAOS is a high-performant independent and fault-tolerant storage tier that does not rely on a third-party tier to manage metadata and data resilience.

DAOS moves away from an I/O model that is designed for block-based, high-latency storage to one that inherently supports fine-grained data access and unlocks the performance of the next generation storage technologies. Figure 3 presents an overview of the DAOS architecture.

DAOS servers maintain their metadata on persistent memory, with bulk data going straight to NVMe SSDs. In addition, small I/O operations are absorbed on the persistent memory before being aggregated and then migrated to

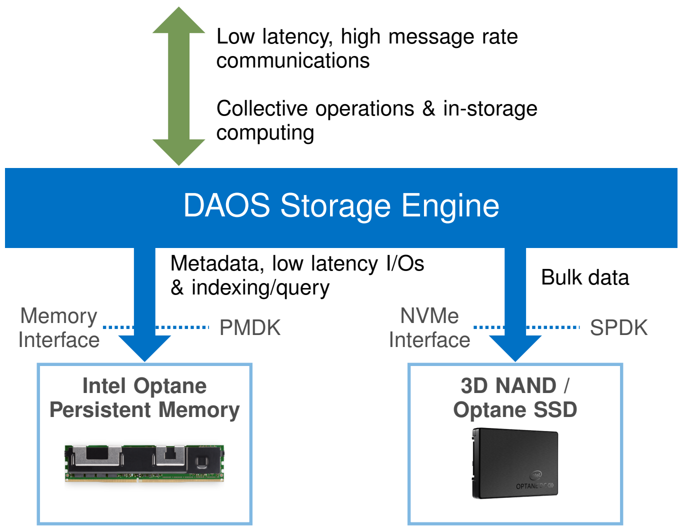

Low latency, high message rate communications
Collective operations & in-storage computing
DAOS Storage Engine
Metadata, low latency I/Os & indexing/query
Bulk data
Memory Interface
PMDK
NVMe Interface
SPDK
Intel Optane Persistent Memory
3D NAND / Optane SSD

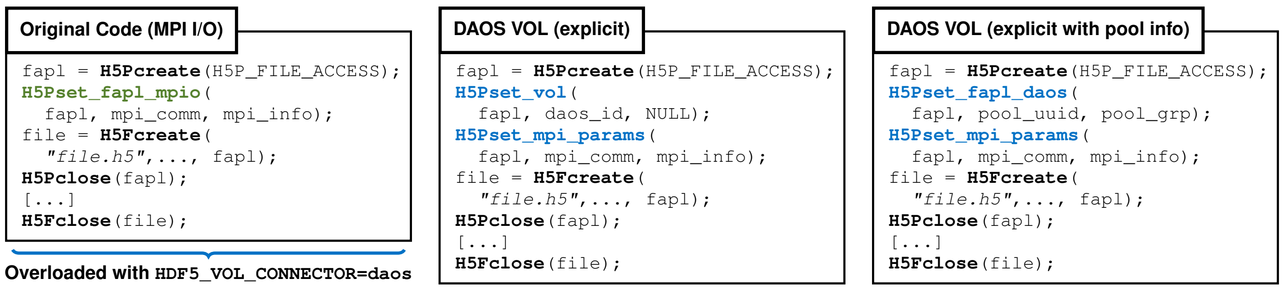
Fig. 3: Overview of DAOS storage engine. Metadata is stored on persistent memory, while bulk data is stored on NVMe SSDs.

the larger capacity flash storage. DAOS uses the Persistent Memory Development Kit (PMDK) [20] to provide transactional access to the persistent memory and the Storage Performance Development Kit (SPDK) [21] for user space I/O to NVMe devices. DAOS is extremely lightweight since it operates end-to-end in user space with full OS bypass. This architecture allows for data access times several orders of magnitude faster than in existing storage systems (in the order of μs vs ms).

In this new storage paradigm, POSIX is no longer the foundation for new data models. Instead, the POSIX interface is built as a library on top of the DAOS backend API, similar to any other I/O middleware. A POSIX namespace can be encapsulated within a container and mounted by an application into its file system tree. While most HPC I/O middleware could run transparently over a DAOS backend via the POSIX emulation layer, migrating I/O middleware libraries to support the DAOS API natively adds the benefit of DAOS's rich API and advanced features.

3.3 Summary

In essence, HDF5 could support DAOS using either MPI I/O or POSIX I/O via the HDF5 native file format and mapping DAOS into a block-based representation, though its benefits would be limited by the same POSIX constraints seen with legacy file systems. Contiguous raw data bandwidth would still be improved due to the nature of DAOS and high-performing hardware, nevertheless I/O operations would remain blocking operations and metadata writes would remain collective, preventing the number of I/O operations from scaling. Taking advantage of the full potential of DAOS's functionality requires a VOL connector that makes direct use of the DAOS API and an optimized storage format. The VOL allows applications to take advantage of DAOS's advanced features, replacing a POSIX HDF5 file with a DAOS container, by making few, if any, code changes to an existing application, while also exposing non-blocking I/O and fine-grained data control semantics.


Original Code (MPI I/O)

```
fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(
  fapl, mpi_comm, mpi_info);
file = H5Fcreate(
  "file.h5",..., fapl);
H5Pclose(fapl);
[...]
H5Fclose(file);
```

Overloaded with `HDF5_VOL_CONNECTOR=daos`

DAOS VOL (explicit)

```
fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_vol(
  fapl, daos_id, NULL);
H5Pset_mpi_params(
  fapl, mpi_comm, mpi_info);
file = H5Fcreate(
  "file.h5",..., fapl);
H5Pclose(fapl);
[...]
H5Fclose(file);
```

DAOS VOL (explicit with pool info)

```
fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_daos(
  fapl, pool_uuid, pool_grp);
H5Pset_mpi_params(
  fapl, mpi_comm, mpi_info);
file = H5Fcreate(
  "file.h5",..., fapl);
H5Pclose(fapl);
[...]
H5Fclose(file);
```

Fig. 4: Three usage options of the DAOS VOL within an existing application code.

4 HDF5 VOL Connector

This section describes the architecture of the DAOS VOL connector, the HDF5 storage format on top of DAOS objects, and the new capabilities that it enables.

4.1 VOL Connector

As represented in Figure 5, the DAOS VOL connector sits directly under the HDF5 API, exposing callbacks to API routines that may be accessing the storage (e.g., H5F files, H5G groups, H5D datasets, etc.). By doing so, it is able to completely bypass the native connector implementation, which is built on top of POSIX and MPI I/O and offers only a byte-stream representation. It is worth noting that DAOS can also be accessed through the DAOS File System (DFS) API, integrated within an MPI I/O ROMIO backend driver [22], which interfaces directly with MPI I/O. While the HDF5 library can use the native method and this back-end in order to access DAOS containers, it is tied to the POSIX restrictions that are mentioned in Section 3.

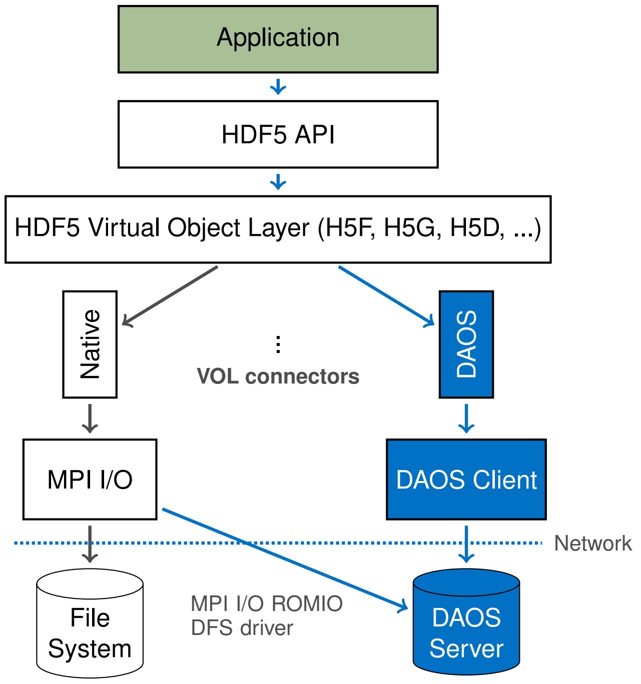


Fig. 5: DAOS within Virtual Object Layer. All of the HDF5 I/O related calls are routed to the VOL connector.

This connector redefines the base set of HDF5 APIs to store data in DAOS. However, to expose advanced features, extensions to the HDF5 library are necessary and we describe them in Section 4.3.

4.1.1 *Application Usage and Portability*

DAOS stores containers in pools that are referenced by a UUID and follow user/group ownership and permission rules. To start creating DAOS containers through the HDF5 API, an application must first inform the library of that UUID and the specific connector that it wants to use in order to create HDF5 files (even if DAOS is present on a system, applications still have the choice to fallback to a native representation). To select and use the DAOS VOL connector, the application is offered several options that are highlighted in Figure 4. Which option to choose depends on the ability to modify the application's source code, its desired portability, etc.

The first option uses environment variables to overload an application configuration. `HDF5_VOL_CONNECTOR` specifies the connector to use (`daos`), `DAOS_POOL` the pool's UUID, and `HDF5_PLUGIN_PATH` the path to the VOL connector library. With that information, the DAOS connector is dynamically loaded and supplied the pool UUID. This option, however, restricts an application to a single VOL connector when creating or opening HDF5 files. It is useful when the application cannot be modified or is experimenting with its use of DAOS storage. Nonetheless, it remains possible in this configuration to access existing files from multiple DAOS pools in a single application and this can be done through a specific DAOS feature called, Unified Namespace (UNS), described in Section 4.1.2. The second option no longer uses the `HDF5_VOL_CONNECTOR` environment variable but the application explicitly expresses its intent of using the DAOS VOL with a particular file by making a call to `H5Pset_vol()`. Finally, the third option does not use environment variables and uses explicit linking. It consists of calling the following routine with explicit pool and pool group name information, allowing for multiple files to be created with different pools:

```
herr_t H5Pset_fapl_daos(hid_t fapl_id, const
    uuid_t pool_uuid, const char *pool_grp);
```

While these two last options require modification of the application's source code, they allow applications to use multiple VOL connectors at the same time, which is useful if an application has files that were created with the native connector and files that are newly created through DAOS. Furthermore, they both decouple the use of MPI communicator information, which is made optional and passed through `H5Pset_mpi_params()`. This allows non-MPI applications to use the DAOS VOL, while MPI-dependent applications can maintain collective semantics if desired (e.g., for compatibility purposes).

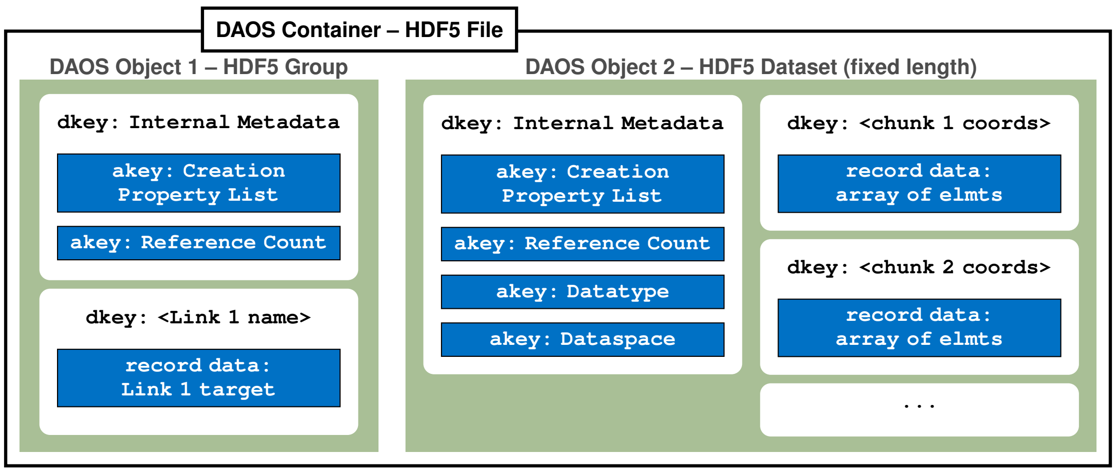

DAOS Container – HDF5 File
DAOS Object 1 – HDF5 Group
dkey: Internal Metadata
akey: Creation Property List
akey: Reference Count
dkey: <Link 1 name>
record data: Link 1 target
DAOS Object 2 – HDF5 Dataset (fixed length)
dkey: Internal Metadata
akey: Creation Property List
akey: Reference Count
akey: Datatype
akey: Dataspace
dkey: <chunk 1 coords>
record data: array of elmts
dkey: <chunk 2 coords>
record data: array of elmts
. . .

Fig. 6: Simplified representation of a file composed of a single group and a dataset.

4.1.2 *File Accessibility and Unified Namespace*

While DAOS makes heavy use of UUIDs, HPC application users are used to interacting with a system namespace and not UUIDs for data storage. The DAOS Unified Namespace (UNS) is a feature that enables exposing DAOS pools and containers in a system namespace as files and directories. This feature works in conjunction with any file system that supports extended attributes (e.g., Lustre, Spectrum Scale, etc.) and is therefore integrated within the DAOS VOL so that creation, opening and deletion of HDF5 DAOS files can be routed through that layer. As described in Figure 7, when opening an existing HDF5 DAOS file with the UNS, the DAOS VOL connector queries the extended attributes on a UNS file stub (created at file create time). These attributes then inform the connector of both the DAOS pool and container UUIDs that were used to create the file, facilitating the handling of that extra metadata for the application.

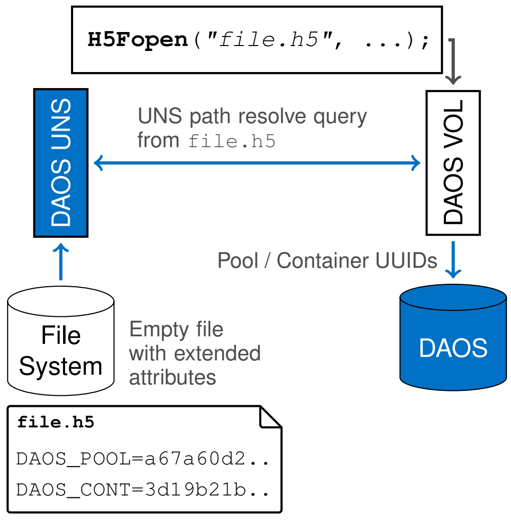


Fig. 7: Example of accessing DAOS file through UNS.

Furthermore, as files may reside either on a POSIX filesystem (created through the native connector) or in a DAOS container, the HDF5 library has an added ability to automatically detect which VOL connector must be used to access a particular file. This feature is currently implemented by attempting to open a file using a predefined list of VOL connectors to use.

4.2 DAOS VOL File Format

With this object-based architecture in place, we can define how HDF5 objects are stored to DAOS. This process, by definition, describes a new HDF5 file format. Implementing a VOL connector requires all operations for each object type to be redefined. Our goal in that context is to ensure that the newly defined objects can be created and accessed efficiently without any contention, while allowing for DAOS features to be natively supported and exposed to the user.

4.2.1 *DAOS Model*

A DAOS object can be accessed through different APIs but the *multi-level key-array* API, which is the DAOS native object interface with locality feature, is of special interest. The key is split into a distribution key (*dkey*) and an attribute key (*akey*). Both the dkey and akey can be of variable length and type (i.e., a string, an integer or even a complex data structure). All entries under the same dkey are guaranteed to be colocated on the same storage target—this is a very important point for the rest of the discussion. By being able to control locality, keys for a given object can not only be accessed with minimum latency but they can also be spread over multiple storage targets, increasing overall bandwidth. The value associated with an akey can either be a single variable-length value that cannot be partially overwritten or an array of fixed-length values. Both the akeys and dkeys support enumeration.

4.2.2 *HDF5 Representation*

In that context, the central paradigm for mapping HDF5 to DAOS is that HDF5 files are stored as DAOS containers, and HDF5 objects (groups, datasets, etc.) are stored as DAOS objects, both using a one-to-one mapping in order to maintain DAOS object properties and provide maximum scalability as the number of HDF5 objects increases. To preserve HDF5's hierarchy, each container for an HDF5 file must contain at least two objects: the global metadata object and the root group. The global metadata object stores metadata that describes HDF5 properties that apply to the entire file. The root group is always the start of an HDF5 group structure and uses the same format as all other groups. From the root, all metadata associated with underlying HDF5 objects can be stored as entries in the key-value store associated with that object and provided by DAOS. This allows us to keep the HDF5 object's metadata along with its data, thereby preserving HDF5's object-based model.

Raw data, mainly stored in datasets, is divided into *chunks*, which are regularly spaced blocks of data elements. As shown in Figure 6, each data chunk is stored in its own dkey, as an array of records, where each record corresponds to an element in the dataset. This representation allows for datasets to be distributed among multiple DAOS targets, maximizing bandwidth.

4.3 DAOS VOL Specific Features

This new representation and API enable new features for the application user that were not available with a POSIX representation: fine-grained data control and placement, independent HDF5 metadata operations including object creation and deletion, a new type of HDF5 object, *Maps*, and asynchronous I/O.

4.3.1 Data Control and Placement

An application's output can usually be comprised of different data structures and types, resulting in HDF5 datasets of very disparate sizes. Additionally, it is common, within an output file, that only part of the data varies while the other remains fixed (e.g., mesh used to represent fixed objects in a domain). In that context, there is a need for an application to express different storage properties, depending on the object that is stored. DAOS uses the concept of *object classes* to describe the data distribution and protection scheme for an object. These object classes allow applications to be more flexible in the way they utilize underlying storage by informing DAOS about how an individual object is intended to be used. To provide this flexibility to HDF5 applications, the DAOS VOL connector introduces the following API that allows setting of a particular DAOS object class on a file access property list or object creation property list.

```
herr_t H5daos_set_object_class(
    hid_t plist_id, char *object_class);
```

By default, the connector uses the `S1` (unstriped) object class for all objects except datasets, which use the `SX` (fixed stripe) object class to maintain data distribution across the available DAOS storage targets. The usage of these particular object classes means that data protection is not enabled by default for any objects, but it may also be enabled by using the connector API to set an appropriate DAOS object class for that object (e.g., `RP_2G1` for 2-way data replication).

In complement, the `H5Pset_chunk()` routine can control the size of each dataset's chunk, using the representation described in Section 4.2, allowing for chunks to be spread among storage targets in order to increase throughput.

4.3.2 Independent Object Creation

As discussed in previous sections, parallel native HDF5 has historically kept its semantics collective for metadata write operations (e.g., object create operations), though they are independent for metadata read operations (e.g., object access operations). The HDF5 API routine `H5Pset_all_coll_metadata_ops()` can in effect be used to change metadata *reads* to use a collective interface, but POSIX imposes a hard requirement that all HDF5 metadata write operations must remain collective. One consequence of that design limitation is that applications have been structured around these more constraining semantics. Therefore, for application compatibility, the DAOS VOL connector continues to expose collective semantics by default and `H5Pset_all_coll_metadata_ops()` can similarly be used to change metadata reads to be collective. However, the DAOS VOL connector also introduces an additional API, which can be used to enable independent metadata *write* operations on a file or object (group, dataset, etc.):

```
herr_t H5daos_set_all_ind_metadata_ops(
    hid_t accpl_id, hbool_t is_independent);
```

This feature allows application developers more flexibility in structuring their code while preserving locality within a given process, as collective synchronization is no longer needed between ranks in order to create new objects. A direct usage example is to allow each independent process rank to create its own local HDF5 group and underlying set of objects, without needing to collectively coordinate with other ranks.

4.3.3 Maps

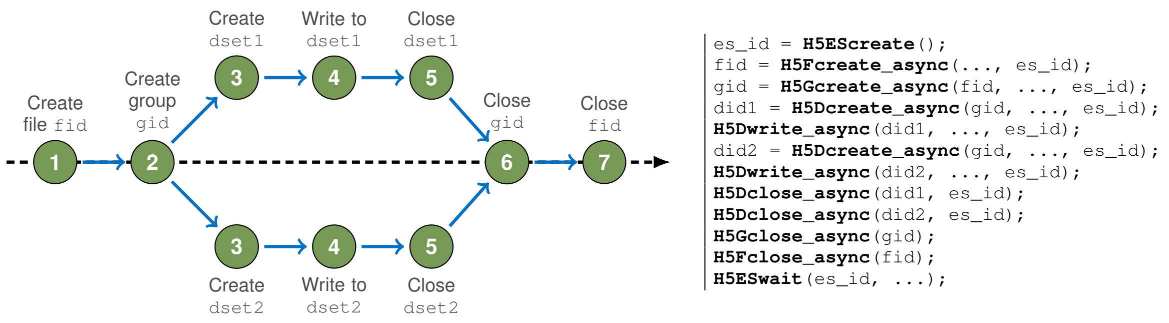
While the HDF5 data model is a flexible way to store data that is widely used in HPC, some applications require a more general or unstructured storage model to store information. To facilitate this type of data model, HDF5 introduces with this VOL connector a new application-defined key-value store object, called a *Map* object, that provides put and get semantics, e.g.:

```
herr_t H5Mput(hid_t map_id,
    hid_t key_mem_type_id, const void *key,
    hid_t val_mem_type_id, const void *value,
    hid_t dxpl_id);
```

Translating HDF5 Map objects to DAOS follows similarly to other HDF5 objects. When adding a key-value pair to a Map object, the value is stored under its own DAOS dkey, hence key-value pairs can be spread in the same fashion as dataset chunks. However, as opposed to the more general array type of storage that is offered through the dataset type of object, these APIs offer a way for applications to directly insert and append new data records without the need to describe the total space that is needed to insert those records. This feature is very relevant to applications where it is either difficult to aggregate data to create an array of those records (as memory space is limited) or have a need for accessing and identifying records individually.

4.3.4 Asynchronous I/O

In the past, asynchronous I/O, while existing through POSIX `aio`, has never been really encouraged due to its overhead and performance issues over Lustre file systems. DAOS, however, exposes a fully non-blocking API that allows for multiple operations to be posted and later progressed, with an explicit progress call. Rather than being implicit, explicit progress allows for more flexibility and control of the progress loop and overall progress of the operations. Asynchronous I/O enables applications to either post multiple I/O operations at the same time for improved throughput or to overlap computation and I/O.



```
es_id = H5EScreate();
fid = H5Fcreate_async(..., es_id);
gid = H5Gcreate_async(fid, ..., es_id);
did1 = H5Dcreate_async(gid, ..., es_id);
H5Dwrite_async(did1, ..., es_id);
did2 = H5Dcreate_async(gid, ..., es_id);
H5Dwrite_async(did2, ..., es_id);
H5Dclose_async(did1, es_id);
H5Dclose_async(did2, es_id);
H5Gclose_async(gid);
H5Fclose_async(fid);
H5ESwait(es_id, ...);
```

Fig. 8: Example of concurrency and dependency when using asynchronous I/O routines.

4.3.4.1 Implementation: As depicted in Figure 8, to expose non-blocking I/O semantics, current HDF5 operations are augmented with a parameter that allows for tracking of these operations so that completion can be determined at a later time (non-blocking HDF5 routines have an `_async` suffix that distinguishes them from their blocking counterparts). Rather than tracking operations individually, HDF5 uses the notion of event sets that instead allow for bulk tracking of operations and an event set can be waited on with a call to `H5ESwait()`. This call has the double functionality of testing for completion but also forcing explicit progress of these operations for a given timeout.

Asynchronous I/O in the DAOS VOL connector is currently implemented using a thread-less task progress engine, which checks for completion of in-flight operations any time it is entered. This means that there is no background thread making progress, which guarantees that I/O will not interfere with computation (hence preventing use of an additional CPU core). However, as depicted in Figure 8, HDF5 operations being hierarchical, there is a strong likelihood that they may depend on one another (e.g., `H5Dwrite_async()` cannot start before `H5Dcreate_async()` completes) and in those cases, this means that the HDF5 library may either need to be entered occasionally in order to make progress or calls to `H5ESwait()` with a zero timeout may be made in order to force progress and posting of dependencies. This is obviously a generic example and it is up to applications to ensure that asynchronously posted operations remain independent as much as possible.

4.3.4.2 Completion and Synchronization: Similar to other asynchronous I/O libraries, in order to prevent memory copies, the application must be careful not to use, modify or free any buffers in use by asynchronous tasks until those tasks are complete. This applies to all read and query operations, as well as raw data and attribute write operations. For small (non-attribute) metadata write operations, the connector will make a temporary copy of any buffers passed in. The application must also be careful not to assume write operations are visible in the file until it has verified that the operation has been completed (e.g., if a dataset is written to asynchronously, the application must wait for the write to complete before reading that data).

It is possible in some cases, however, to issue operations before prerequisites have been complete. Any identifier (ID) returned from the HDF5 API can be passed back in through the API even if the open operation for the object that that ID refers to has not been completed. This allows applications to, for example, create a file, a group in the file, a dataset in the group, write to the dataset, and close all IDs, all in a non-blocking manner without waiting (until the dataset write buffer needs to be modified or freed). As already mentioned, asynchronous operations issued to the DAOS VOL connector generally execute concurrently without any ordering enforced between them, though the connector enforces ordering between object open operations and operations that use that object, in order to facilitate the use of incompletely opened object IDs. The application can also manually enforce ordering (in a non-blocking fashion) using the following API calls: `H5Oflush_async()` and `H5Fflush_async()`. These operations only complete when all previously issued operations for the object or file complete, and all subsequently issued operations only begin after the flush is complete.

Finally, it is also worth noting that parallel collective operations (when used) add another constraint and implicit synchronization point on asynchronous operations. Because asynchronous MPI operations must be strictly ordered, all collective HDF5 operations are strictly ordered with respect to each other when executed with more than one rank. Therefore, care must particularly be taken when using these.

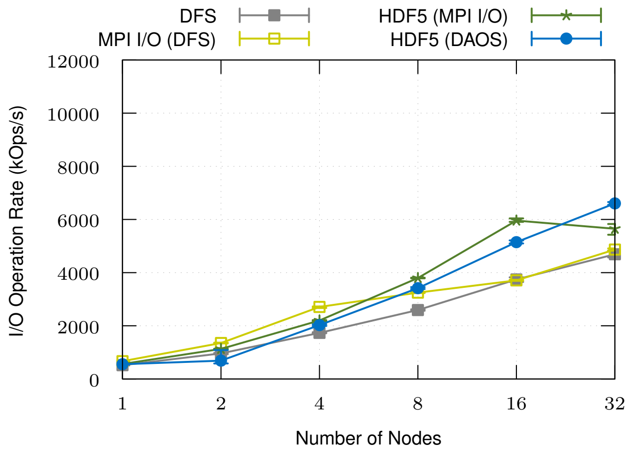
4.3.4.3 Summary: While asynchronous I/O in HPC has so far remained difficult to use efficiently, the asynchronous I/O feature that we expose is expected to improve performance by leveraging DAOS's non-blocking I/O, allowing for multiple I/O operations in flight. Its API and semantics, by being *explicit* as opposed to a more implicit mode of operation, are designed so that applications are entirely in control of the I/O that is made, preventing unwanted use of resources. The ultimate goal that is to allow for overlap of computation and I/O does not come without well thought I/O pipelines and this new API offers an opportunity for application developers to move further into that direction.

5 EVALUATION

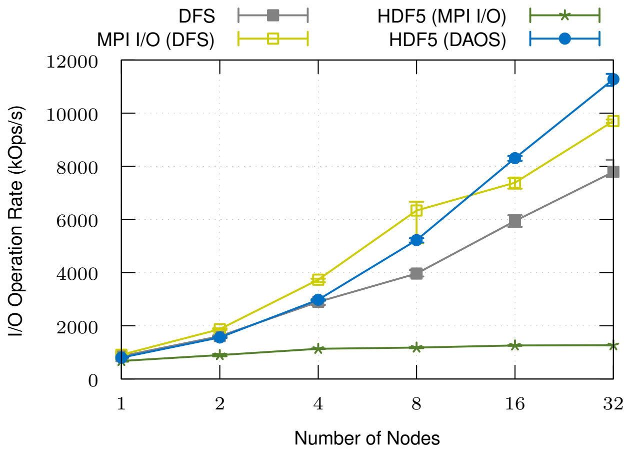
This section focuses on the individual evaluation of the DAOS VOL and its most prominent features, giving an overall idea of its performance in various cases. While we cannot cover all cases in this paper, we attempt to provide the most meaningful representation before focusing on an application use case.

5.1 Configuration

The DAOS configuration used for this evaluation is deployed on the Frontera system, hosted at the Texas Ad-

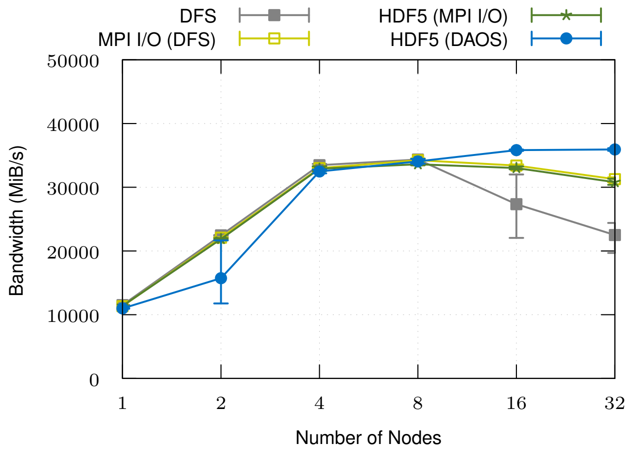


(a) Write.

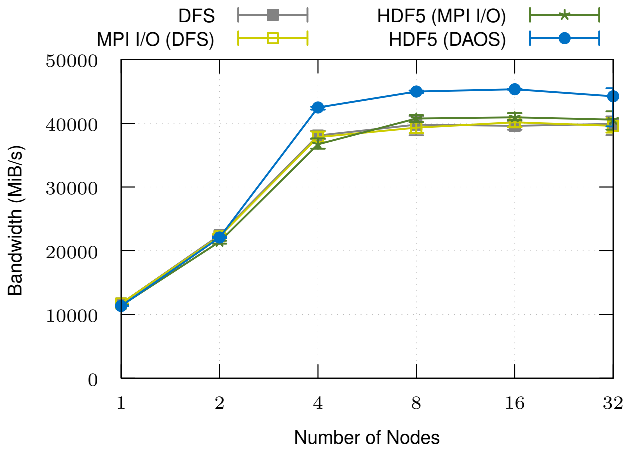


(b) Read.

Fig. 9: IOR operation rate (1KB transfer size, 28 processes per node).



(a) Write.



(b) Read.

Fig. 10: IOR bandwidth (1MB transfer size, 28 processes per node).

vanced Computing Center (TACC). The DAOS system installed is a small configuration composed of only 4 storage nodes that serves only evaluation purposes and is not used in production. Storage nodes (Dell EMC® PowerEdge R840) are composed of 4 sockets, each hosting an Intel® Xeon® Platinum 8280 CPU with 28 cores. Each node is also accompanied of 24 Intel® Optane™ persistent memory DIMMs of 256GB each.The interconnect used on this system is based on Mellanox InfiniBand HDR with full HDR (200 Gb/s) connectivity between the switches and HDR100 (100 Gb/s) connectivity to the compute nodes. Computes nodes used for this experiment are composed of the same CPUs but are only dual sockets with 192GB of RAM DDR4-2933.

For the purpose of those experiments, we make use of only 28 cores located within the same NUMA node as the InfiniBand card. We also use a 2 TB DAOS pool without NVMe backend to make exclusive use of persistent memory. These two settings ensure that there is no performance discrepancies. The DAOS version used is 1.1.2.1, HDF5 version is 1.13.0rc5 and DAOS VOL version is 1.1.0rc3 (pre-release).

5.2 Raw Data and Metadata Performance

We focus on measuring both the raw data and metadata performance of the DAOS VOL.

5.2.1 *Raw Data I/O*

For raw data performance, we make use of the IOR benchmark [23] to measure both small I/O and large I/O performance, while scaling up to 32 nodes (896 cores). While this may seem small for that system, this is already more than enough to evaluate saturation of the storage servers. We make use of a modified version 2.3.5 of MVAPICH2 to enable support of the MPI I/O DFS ROMIO backend while being able to take advantage of the InfiniBand interconnect. The DAOS client itself uses mercury RPC [24] and libfabric [25] over InfiniBand. In that configuration, the maximum theoretical network bandwidth for a 4-node configuration is 50 GB/s. We enhanced the IOR HDF5 backend to support the DAOS VOL and run 10 iterations with small and large I/O (collective I/O is disabled), .e.g.:

```
$ ibrun ior -g -a HDF5 -o "testFile.h5" \
  -b 10m -t 1k -i 10 -w -r              \
  --hdf5.noFill=1 --hdf5.chunkSize=16k
```

Performance is presented in Figures 9 and 10 respectively for IOR using either DAOS's DFS interface, MPI I/O over DFS (POSIX semantics), native HDF5 over MPI I/O with DFS, and finally HDF5 over the DAOS VOL. For small I/O, Figure 9 shows that the performance for both writes and reads is very good in terms of IOPS, achieving more than 10 million IOPS for reads in the 32 node configuration. In terms of bandwidth and for large I/O, Figure 10 shows that DAOS and the DAOS VOL reach close to the theoretical bandwidth that is offered by the hardware. For reads, the network becomes the limiting factor. It is also worth noting that in those two sets of results, MPI I/O and HDF5 with MPI I/O follow a similar performance. IOR does not really expose any limitation from the native HDF5 format for this type of access, which is expected for contiguous I/O, though for small reads and as shown in Figure 9b, native HDF5 chunk cache becomes actually a limiting factor at scale.

5.2.2 *Asynchronous I/O*

In a second experiment, we modified IOR to support measurements of asynchronous I/O. In that case, we transfer a total block size of 1GB of data from one single process to DAOS using the DAOS VOL and various transfer sizes, decomposing the I/O in multiple operations (i.e., 1GB block size with 1MB transfer size equals to 1024 I/O operations posted). The purpose of this experiment is to demonstrate the benefits of asynchronous I/O for applications that usually have to write multiple HDF5 datasets and HDF5 objects, resulting in smaller I/O requests.

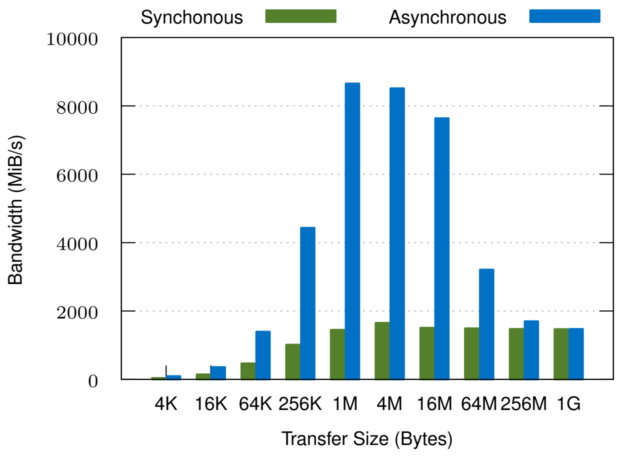


Fig. 11: Effect of Asynchronous I/O with multiple transfer sizes (write). Performance is equal to synchronous I/O for 1GB transfer size when the number of operations is 1.

Results presented in Figure 11 show that in that case, for 1MB transfer size, using the same fixed stripe, the benefits of asynchronous I/O are quite substantial. As opposed to blocking I/O, asynchronous I/O allows in that case for better concurrency and pipelining of independent I/O requests by posting and keeping several operations in flight before waiting for completion. In all cases where the number of operations is greater than one, using asynchronous I/O results in better performance compared to synchronous I/O.

5.2.3 *Independent Object Creation*

In this third experiment, we focus on the effects of enabling independent object creation and metadata I/O. We measure performance for two separate types of operations: creation of groups and datasets, which are the most prominent types of HDF5 objects. We measure performance using mdtest over DFS and using a custom HDF5 benchmark for both native HDF5 over MPI I/O with DFS and HDF5 over the DAOS VOL. The reason for using mdtest for measurements is that it provides us with a direct comparison between our HDF5 implementation and DFS (which is also implemented on top of the DAOS API). Directory create operation over DFS are similar to group create operations over the DAOS VOL and file create operations are equivalent to dataset create operations. It is worth noting though that HDF5 object creation (for groups and datasets) requires two DAOS operations (RPC requests to the DAOS servers) whereas mdtest create operations (for files and directories) are a single DAOS operation. Furthermore, DFS directory creation is not an exact comparison to HDF5 group creation as the object class of the parent directory in mdtest is not widely striped (mdtest does not allow setting the parent and leaf directory to different layouts while the HDF5 metadata benchmark does).

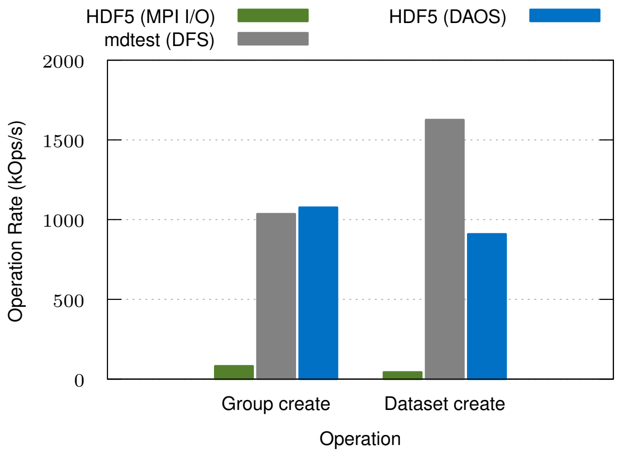


Fig. 12: Metadata operation rate on 4-node configuration (28 processes per node).

Results are presented in Figure 12. On a 4-node configuration, performance of the DAOS VOL is on par with DFS and allows us to create more objects than the native POSIX HDF5 implementation by one order of magnitude higher. This is where the DAOS VOL is able to outperform the native implementation by removing the need for collective synchronization of metadata writes, induced by a POSIX block-based format.

6 Application

Vector Particle-In-Cell (VPIC) [26] is a one, two, and three spatial dimensions particle-in-cell kinetic plasma simulation framework. We study in this section the current challenges that present a code such as VPIC that already uses HDF5 and highlight the new possibilities that are now offered.

6.1 Description and Challenges

VPIC follows a particle-in-cell method and, as such, divides the simulated space into cells while distributing ownership and management of each cell among the processes in the simulation. Within each cell, a process manages a set of moving particles, which often move between cells as the simulation progresses. Every few time steps, VPIC outputs the state of all particles to storage. One common problem that is highlighted in [27] is that scientists are often interested in analyzing the trajectories of a tiny subset of particles that end a simulation with an unusually high energy.

More broadly, the difficulties VPIC scientists experience is a common challenge in data analytics and as mentioned in Section 3, POSIX has forced applications to output data in a way that does not necessarily match the underlying data model, most often optimizing writes by aggregating small data into larger arrays without considering the efficiency of the reads or queries that will be subsequently needed to analyze the data. Solutions such as data indexing have been demonstrated in [28] though building an index has an added cost both in terms of time and space required to store the index. In that context, there is a need for accessing and storing data in a manner that is more efficient in the first place and that preserves the application's data model.

6.2 Independent Object Creation

As described in Section 4, by enabling independent metadata writes, we are now able to independently create HDF5 objects in parallel within a given file. This feature, only present in the DAOS VOL, has two advantages for an application such as VPIC. First, it removes any need for a file per process I/O method that would create several hundreds of files that are difficult to handle. Second, it allows for new organizations of objects within a given file that used to be very costly. As an example we take the extreme case of creating one group per particle, using the particle ID as the group name and representing the particle as an object.

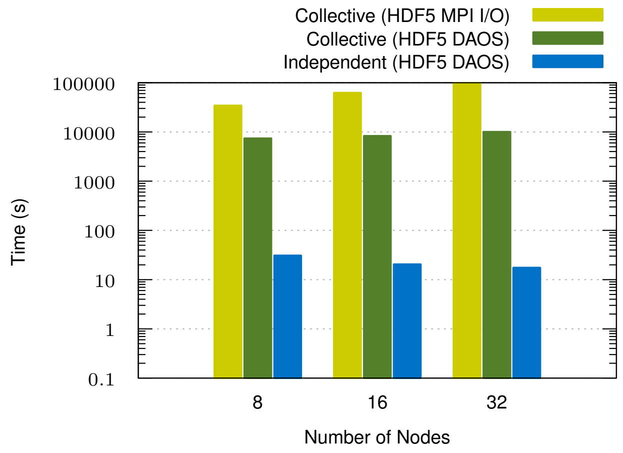


Fig. 13: VPIC I/O performance using collective and independent group creation (28 processes per node, total of ≈ 25.7 M particles is kept constant, one group per particle).

As shown in Figure 13, doing this type of operation with collective metadata is two to three orders of magnitude slower as all processes have to be involved in all group creations, effectively looping over all ranks to create as many groups as ranks, equal in that case to the total number of particles. Independent metadata, however, only requires the local number of particles, which in that case decreases as as the number of processes grows. This feature is therefore essential for scalabilty, preserving process locality and the original intent of the underlying data model.

6.3 Maps

To go further in that direction, Map objects add an additional optimization and are an ideal alternative for dumping particles that are not meant to be kept in arrays. Array objects are difficult to handle, particularly so when their size is not constant, which is the case where particles can migrate between processes. It is also common for simulation codes to make extra copies or allocations to transform data and be able to write the data out as arrays. Using Maps, VPIC can for instance dump particles into key-values, with keys identifying particles, without the extra consideration of the total number of particles or space that is needed. There is an added cost, however, as opposed to array writing as this results in smaller I/O. In its current form, the DAOS VOL requires DAOS's support for transactions, which will be available in its next release, to enable aggregation of small I/O operations, preventing separate I/O operations to be issued per map write operation.

6.4 Asynchronous I/O

One obvious type of scenario for enabling asynchronous I/O in VPIC consists of blindly switching the entire I/O routines to use HDF5 asynchronous I/O. However, as we already discussed in Section 4.3.4, codes are usually thought of as sequential and structured with dependencies between HDF5 operations (i.e., a create operation is followed by a write, which is followed by a close, etc)—this type of ordering prevents performance from being improved on a per-operation basis since they need to be executed sequentially, which in itself defeats the purpose of asynchronous I/O that relies on parallelism of operations. Additionally, legacy codes that relied on native HDF5 were built around collective semantics and this type of semantics adds extra synchronization and overheads.

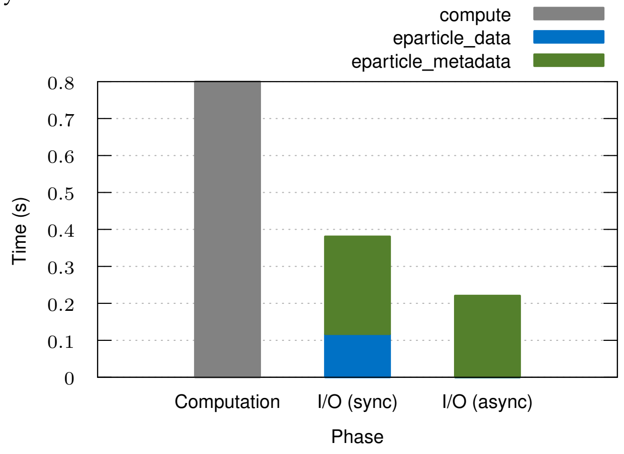


Fig. 14: VPIC I/O performance using asynchronous I/O on electron particle data (≈ 35 M particles). Computation time is given as a comparison and remains constant between synchronous and asynchronous I/O phases.

To illustrate potential benefits of asynchronous I/O, we modified VPIC to add initial support for it, disabling collective metadata I/O in order to prevent potential contentions with the computation phase and keeping a copy of the particle data to evaluate pipelining capabilities. Results are presented in Figure 14 from a single process point of view to highlight advantages and aforementioned drawbacks (note that metadata includes VPIC's processing time for preparing the data output). When outputting particle data, VPIC by default creates a new file, group, dataset, writes the particles to the dataset and then closes all the objects. Overlapping metadata with computation is hence limited by this sequential object creation (see Section 4.3.4). We therefore call `H5ESwait()` right before the dataset write operation to ensure that the write operation gets posted and previous metadata dependencies have been executed. Since there is no background thread (meaning no contention with the computation phase), progress must be made either when the library is entered or when `H5ESwait()` is called. No further wait call is made until the next output in order to overlap transfers. As shown in this figure, raw data, representing about half of the I/O time for this workload, is overlapped with computation, while metadata I/O is slightly improved due to metadata I/O operations being simultaneously posted without waiting between them.

To go further, the I/O pipelines of an application such as VPIC would need to be significantly reworked to take full advantage of asynchronous I/O without dependencies and avoid memory copies. This is out of scope of this paper.

7 Conclusion

Using the HDF5 VOL interface, the HDF5 library has been able to retain its data model and interface while moving from block-based storage to DAOS. This switch to a new object-based format removes previous limitations that were imposed by POSIX I/O and adds new features such as asynchronous I/O, independent metadata writes, and granular control of object placement and object maps, all while benefiting from DAOS's performance and architecture.

Some of the new features that we have described require application rework and care in use (e.g., asynchronous I/O) to utilize effectively, yet they also open new possibilities for re-engineering I/O pipelines around these alternative semantics for improved performance. Even without reworking an application, however, keeping most of the HDF5 API unchanged means that minimal code changes are required for existing applications to benefit from the high performance of DAOS and removal of block-based I/O semantics.

Source Code

Software presented in this paper is open-source and available on GitHub, source code archives can be found at: "HDF5 1.13.0rc5" [Online]. Available: https://github.com/HDFGroup/hdf5/archive/hdf5-1_13_0-rc5.tar.gz.
"HDF5 DAOS VOL 1.1.0rc3" [Online]. Available: https://github.com/HDFGroup/vol-daos/archive/v1.1.0rc3.tar.gz.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy and Argonne National Laboratory and its Leadership Computing Facility, including under Contract DE-AC02-06CH11357 and Award Number 8F-30005. This work was generated with financial support from the U.S. Government through said Contract and Award Number(s), and as such the U.S. Government retains a paid-up, nonexclusive, irrevocable, world-wide license to reproduce, prepare derivative works, distribute copies to the public, and display publicly, by or on behalf of the Government, this work in whole or in part, or otherwise use the work for Federal purposes.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. The authors would like to thank Quincey Koziol (NERSC) for his help in the design of the HDF5 asynchronous interface, as well as Rob Ross, Phil Carns and Kevin Harms (ANL) for their valuable feedback. The authors would also like to thank Sylvia Chan (Intel) for her help in running the experiments presented in this paper. The authors would finally like to thank Neelam Bagha (THG) and Ira Lewis (Intel), project managers, who contributed to the success of this work.

References

[1] The HDF Group. (1997-) Hierarchical Data Format, version 5. Http://www.hdfgroup.org/HDF5.
[2] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *EDBT/ICDT*, 2011, pp. 36–47.
[3] P. J. Braam, "The Lustre storage architecture," *White Paper*, 2004.
[4] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.
[5] Intel Corporation, "Distributed Asynchronous Object Storage." [Online]. Available: https://daos.io
[6] M. S. Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol, "DAOS for Extreme-scale Systems in Scientific Applications," *CoRR*, vol. abs/1712.00423, 2017. [Online]. Available: http://arxiv.org/abs/1712.00423
[7] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, 2000.
[8] J. Bent *et al.*, "Plfs: a checkpoint filesystem for parallel applications," in *Supercomputing*. IEEE, 2009, pp. 1–12.
[9] K. Mehta, J. Bent, A. Torres, G. Grider, and E. Gabriel, "A plugin for HDF5 using PLFS for improved I/O performance and semantic analysis," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 11 2012, pp. 746–752.
[10] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters," in *PDSW*, 2007, pp. 35–44.
[11] Amazon. Amazon Web Services. Http://s3.amazonaws.com.
[12] J. Arnold, *OpenStack Swift: Using, administering, and developing for swift object storage*. O'Reilly Media, Inc., 2014.
[13] J. Liu, Q. Koziol, G. F. Butler, N. Fortner, M. Chaarawi, H. Tang, S. Byna, G. K. Lockwood, R. Cheema, K. A. Kallback-Rose, D. Hazen, and M. Prabhat, "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)*, Nov 2018, pp. 24–34.
[14] The HDF Group, "HDF5 REST VOL plugin." [Online]. Available: https://github.com/HDFGroup/vol-rest
[15] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu, and R. Warren, "Toward Scalable and Asynchronous Object-centric Data Management for HPC," in *CCGRID*, 2018.
[16] J. Mu, J. Soumagne, H. Tang, S. Byna, Q. Koziol, and R. Warren, "A Transparent Server-Managed Object Storage System for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2018, pp. 477–481.
[17] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, Jan 2020. [Online]. Available: https://doi.org/10.1007/s11390-020-9802-0
[18] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, "Faodel: Data management for next-generation application workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*. ACM, 2018, p. 8.

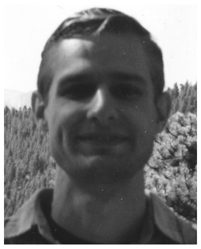
[19] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *MSST*, April 2012, pp. 1–11.
[20] Intel Corporation, "Persistent Memory Development Kit." [Online]. Available: https://pmem.io/
[21] ——, "Storage Performance Development Kit." [Online]. Available: https://spdk.io/
[22] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, 1996, pp. 180–187.
[23] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms," in *In: Cray User Group Conference (CUG'07*, 2007.
[24] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling Remote Procedure Call for High-Performance Computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.
[25] Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/
[26] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.
[27] Q. Zheng, C. D. Cranor, A. Jain, G. R. Ganger, G. A. Gibson, G. Amvrosiadis, B. W. Settlemyer, and G. Grider, "Streaming Data Reorganization at Scale with DeltaFS Indexed Massive Directories," *ACM Trans. Storage*, vol. 16, no. 4, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3415581
[28] S. Byna, J. Chou, O. Rubel, Prabhat, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–12.

Jerome Soumagne received his Master's degree in Computer Science in 2008 from ENSEIRB Bordeaux and his Ph.D. degree in Computer Science in 2012 from the University of Bordeaux in France while being an HPC software engineer at the Swiss National Supercomputing Centre in Switzerland. He is now a lead HPC software engineer at The HDF Group. His research interests include large scale I/O problems, parallel coupling and data transfer between HPC applications and in-transit/in-situ analysis and visualization. He is responsible for the development of the Mercury RPC interface and an active developer of the HDF5 library.

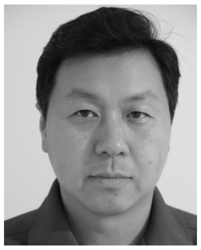
Jordan Henderson received his B.S. degree in Computer Science in 2016 from Eastern Illinois University. He currently works at The HDF Group as a developer of the HDF5 library.

Mohamad Chaarawi is a senior software engineer in the Extreme Scale Architecture & Development Division at Intel. He is responsible for the DAOS client software stack, the POSIX emulation over the DAOS API, and other middleware libraries, and works closely with application and I/O middleware developers to migrate to the new DAOS storage model. Before that, Mohamad was a lead developer at The HDF Group. Mohamad earned his doctoral degree in computer science from the University of Houston; his research focused on Parallel I/O.

Neil Fortner received his Bachelor's degree in Aerospace Engineering in 2004 from the University of Maryland at College Park. He is currently a software engineer at The HDF Group, and was principal investigator for The HDF Group's effort to develop the HDF5 DAOS VOL connector.



Scot Breitenfeld received his Master's degree in 1998 and a Ph.D. degree in 2014, both in Aerospace Engineering from the University of Illinois at Urbana-Champaign. He currently works for The HDF Group in parallel computing and HPC software development. His research interests are in scalable I/O performance in HPC and scientific data management.



Raymond Lu received his bachelor of Science degree in architecture from Tianjin University, China in 1992 and his Master's degree in computer science from the University of Illinois at Urbana-Champaign in 1999. He has worked in The HDF Group for more than fifteen years as a software engineer.

Elena Pourmal is one of the founders of The HDF Group, a not-for-profit company with the mission to develop and sustain the HDF technology, and to provide free and open access to data stored in HDF. She has been with The HDF Group since 1997 and for more than 20 years led HDF software maintenance, quality assurance and user support efforts. Ms. Pourmal currently serves as The HDF Group Engineering Director leading HDF5 engineering effort and is also a member of The HDF Group Board of Directors. Elena received her MS in Mathematics from Moscow State University and MS in Theoretical and Applied Mechanics from University of Illinois at Urbana-Champaign.

Dana Robinson received his B.S. degree in biochemistry in 2003 and his Ph.D. in chemistry in 2010, both from the University of Illinois at Urbana-Champaign. He has been with The HDF Group since 2009, where he currently works as a lead developer on the HDF5 library and as a scientific data consultant.

Johann Lombardi is a Principal Architect in the High Performance Data Division at Intel. He started to work on Lustre back in 2003 and led the sustaining team in charge of the Lustre filesystem worldwide support for more than 5 years. He then transitioned to research programs for Exascale storage in 2012 and now leads the effort at Intel to develop a storage stack for Exascale HPC and Data Analytics.