Extreme Scale Storage and I/O RND

Final Report – Milestone 8.1

High Performance Data Division

June 2017

Government Purpose Rights: Prime Contract No.: DE-AC52-07NA27344
LLNL Subcontract No.: B613306
Subcontractor Name: Intel Federal LLC, on behalf of itself, its parent and Affiliates
Subcontractor Address: 4100 Monument Corner Dr, Ste 540, Fairfax, VA 22030

The Government's rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by the above agreement.

Limited Rights: Prime Contract No.: DE-AC52-07NA27344
LLNL Subcontract No.: B613306
Subcontractor Name: Intel Federal LLC, on behalf of itself, its parent and Affiliates
Subcontractor Address: 4100 Monument Corner Dr, Ste 540, Fairfax, VA 22030

The Government's rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by the above agreement.

Acknowledgment: This material is based upon work supported by Lawrence Livermore National Laboratory subcontract B613306.

Disclosure Notice: This presentation is bound by Non-Disclosure Agreements between Intel Corporation, the Department of Energy, and DOE National Labs, and is therefore for Internal Use Only and not for distribution outside these organizations or publication outside this Subcontract.

USG Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Intel Disclaimer: Intel makes available this document and the information contained herein in furtherance of extreme-scale storage and I/O research and development. None of the information contained therein is, or should be construed, as advice. While Intel makes every effort to present accurate and reliable information, Intel does not guarantee the accuracy, completeness, efficacy, or timeliness of such information. Use of such information is voluntary, and reliance on it should only be undertaken after an independent review by qualified experts.

Access to this document is with the understanding that Intel is not engaged in rendering advice or other professional services. Information in this document may be changed or updated without notice by Intel.

This document contains copyright information, the terms of which must be observed and followed.

Reference herein to any specific commercial product, process or service does not constitute or imply endorsement, recommendation, or favoring by Intel or the US Government.

Intel makes no representations whatsoever about this document or the information contained herein.

IN NO EVENT SHALL INTEL BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY

USE OF THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS INTERRUPTION, OR OTHERWISE, EVEN IF INTEL

IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Document Revision History

Revision Number	Date	Comments
0.9	June 2017	Draft of Extreme Scale Storage and I/O RND final report, MS8.1.
1.0	July 2017	Updated report from feedback

Contents

Figures

1 Executive Summary

Data-intensive applications in science and other domains are already stretching limits of existing I/O architectures. The increasing size and complexity of both structured and unstructured datasets and the I/O workloads featuring increasing metadata and fragmented data are not suitable for the conventional I/O storage stack. New HPC workloads are becoming less bulk synchronous and feature more streaming I/O workloads, data centric applications, and data analytics components.

Current HPC storage systems perform poorly with random, unaligned, small I/O sizes. Data models are limited by POSIX requirements and the corresponding performance implications. Additionally, traditional parallel file systems will face issues scaling to exascale architectures as the number of compute nodes and storage servers increase. Non Volatile Memory and NAND/SSDs are bridging the big hardware gap between main memory and disk drives, but the software stack has not evolved to exploit high performance features required for exascale architectures, such as concurrency control mechanisms, consistency and recovery from failures.

Distributed Asynchronous Object Storage (DAOS) is a new storage stack designed from the ground up to address the challenges of existing storage stacks and provide novel features that enable more complex data models and workflows. DAOS provides fine grained I/O, and high IOPs at arbitrary alignment and size by using a persistent memory storage model for byte granular data. The DAOS API is asynchronous to allow I/O overlap with application computation. Additional features include replication and erasure coding, lockless consistency model at arbitrary alignments and scalable transactions with guaranteed consistency and automated recovery. DAOS will provide configuration and monitoring API similar to software-defined storage services. Finally, DAOS will support multiple storage tiers with implicit or explicit data movement among various tiers. Existing applications, programming models, and I/O middleware libraries, like HDF5, will utilize DAOS to store and organize their data.

Five applications were ported to DAOS for performance evaluation, testing correctness, and identifying DAOS and HDF5 feature gaps. The applications (section 6), were HACC, CLAMR, Legion, NetCDF and PIO. Substantial improvements to the DAOS HDF5 library (section 6.1), showed improvement regarding ease of application migration and productivity when compared with the initial I/O stack developed in the Fast Forward Project with IOD/DAOS [17] (section 6.1). Four out of the five applications (CLAMR, Legion, NetCDF and PIO) were ported from the initial DAOS/IOD implementation to DAOS.

DAOS relies on several software libraries developed with support from DOE such as Mercury [14] and Argobots [13]. Section 4.2.1 lists other DAOS dependencies.

2 Terminology

The following terms and acronyms are found in this document:

Term/Acronym	Meaning
DAOS	Distributed Asynchronous Object Storage
DAOS-M/DSM	DAOS Persistent Memory Storage layer
DAOS-SR/DSR	DAOS Sharding and Resilience layer
DAOS-P/DSP	DAOS POSIX layer
HDD	Hard Disk Drive
IOD	IO Dispatcher (software running on the ION)
ION	IO Node
KV	Key - Value
MTBF	Mean Time Between Failures
NVML	Non-Volatile Memory Libraries
OFI	Open Fabrics Interface
PM/PMEM	Persistent Memory
RAS	Reliability, Availability & Serviceability
RDMA/RMA	Remote (Direct) Memory Access
SPDK	Storage Performance Development Kit
SSD	Solid State Drive
UUID	Universally Unique IDentifier

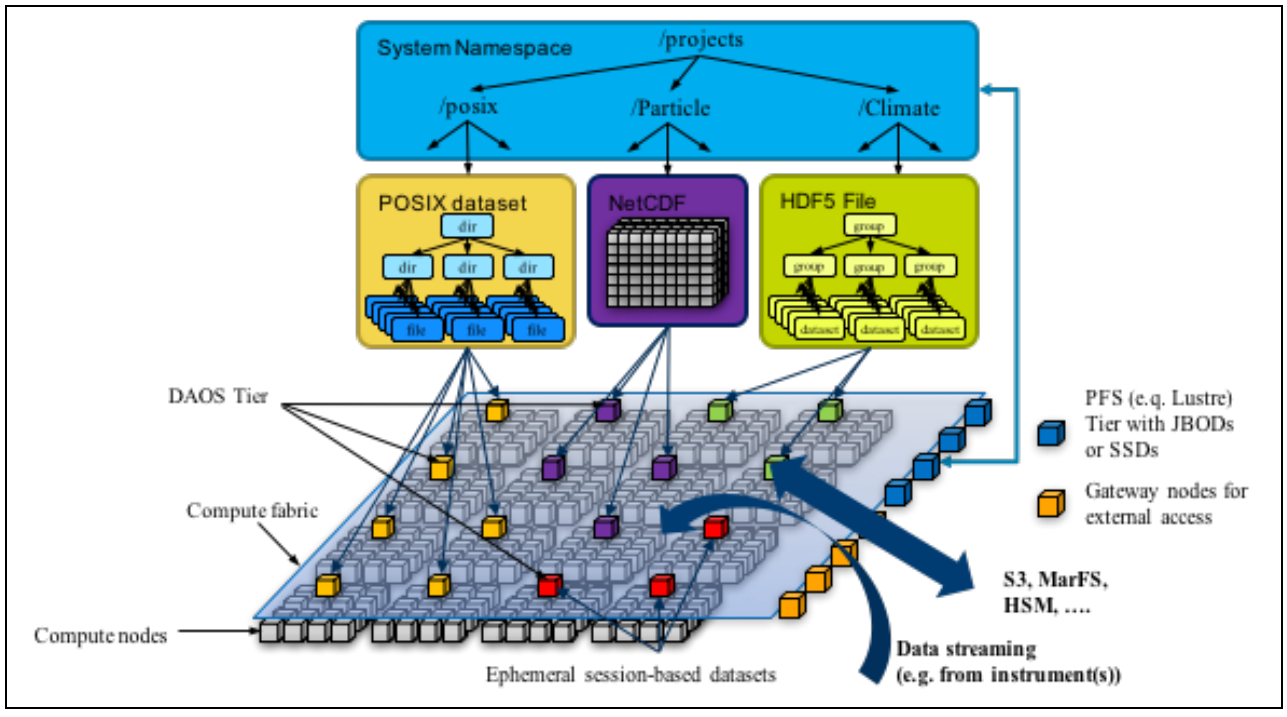
3 Design

3.1 DAOS: Distributed Asynchronous Object Storage

DAOS is an open-source storage stack designed from the ground up to exploit new NVM and integrated fabric technologies. Unlike traditional storage stacks primarily designed for rotating media, the DAOS stack is extremely lightweight since it is designed to work end-to-end in user-space with full OS bypass. I/O operations are handled in a client library linked directly with the application or the middleware I/O library, transferred through the fabric using Mercury, an RPC library designed for HPC systems, to storage services running in user-space. Data is transferred directly into NVRAM with no memory copy or context switch, bypassing the in-kernel filesystem and block layer.

Figure 3-1 depicts what we envision for a future HPC system architecture:

Figure 3-1. Exascale Storage Vision

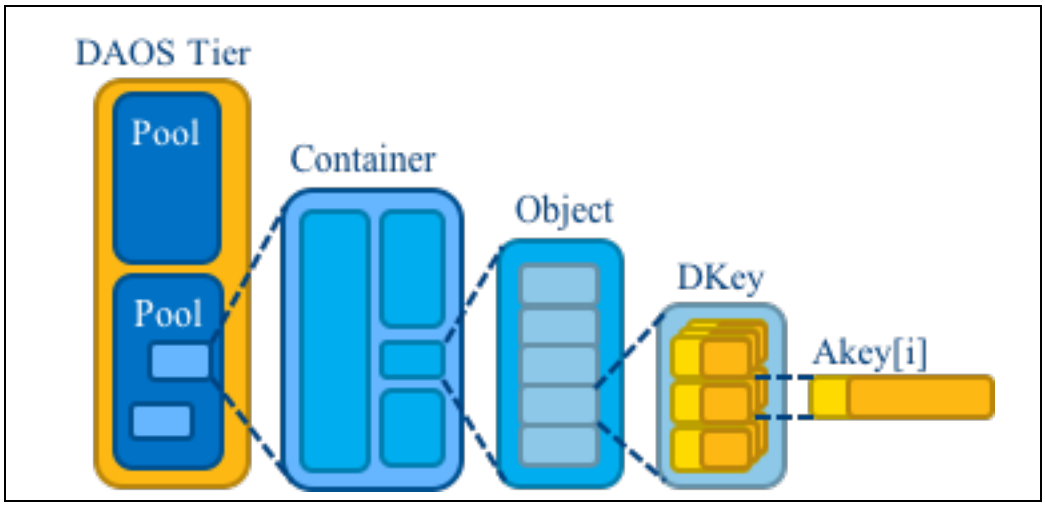


We consider an HPC cluster with hundreds of thousands of compute nodes interconnected with a scalable, high-speed, low-latency fabric where all (or a subset) of the nodes, called storage nodes, have direct access to byte-addressable persistent memory and optionally block-based NVMe storage. A storage node can export, over the network, one or more DAOS targets, each of which corresponds to a fixed-size partition of its directly accessible storage. A storage node can host multiple targets within the limits of the available storage capacity. In the near future, DAOS would support additional I/O nodes to host a parallel file system, like Lustre, and provide a smooth migration path to DAOS. This parallel file system would host a

system namespace with links to persistent DAOS containers, since DAOS does not support a system namespace. In addition, gateway nodes may be used for devices streaming data to the DAOS namespace.

Figure 3-2 below represents the fundamental abstractions of the DAOS storage model. A DAOS pool is a collection of targets distributed across multiple storage nodes. A pool is responsible for both data and metadata resilience. A UUID identifies each pool and maintains target membership persistently. A pool can host multiple transactional object stores called DAOS containers, identified by UUIDs. Each container is a private object address space, which can be atomically modified and snapshotted independent of other containers sharing the same pool. DAOS objects in a container are identified by a unique object address and are Key-Array objects, with an indexed array of fixed sized elements under a particular key. A traditional Key-Value interface is also supported for the same object for unstructured data. Objects can be distributed across any target of the pool for both performance and resilience.

Figure 3-2. DAOS Storage Model



3.1.1 DAOS Feature Set

The DAOS API provides a rich feature set upon which high performance IO middleware and applications can be built to provide advanced IO capabilities to the application developers.

A list of DAOS features is provided below and includes features currently not implemented:

- Non-blocking list I/O to allow I/O-computation overlap
- Scalable distributed transactions are exported to I/O middleware with guaranteed data consistency and automated recovery. This allows the data model to move atomically from one consistent state to another one with automated rollback of uncommitted changes on application failure.
- Loosely coupled execution to support more than the bulk synchronous parallel (BSP) model. The DAOS API will support more data centric models, in-situ workflows, and data analytics frameworks.

- Software-defined storage management API to allow integration with resource and workflow management subsystems.
- Collective and independent access.
- Native producer/consumer pipeline. Producers are not blocked by concurrent consumers and consumers receive notification on complete atomic updates and see a consistent snapshot of data.
- Distributed consistent snapshots.
- End-to-end data integrity. I/O middleware or application can optionally provide a checksum (checksum computed internally by DAOS if not provided) that will be stored and verified on access and background scrubbing. Data corruptions can be not only detected, but also corrected, by reading from a replica or reconstructing from erasure code.
- Integrated multi-tier management with explicit data movement and transparent cache fetch on miss.
- Infrastructure to ship and execute code on the server side to support advanced analytics and new programming models.

This rich feature set is expected to be extended as needs arise to further assist scientific application and middleware developers. Section 4 has more information on features implemented for the ESSIO program.

3.1.2 DAOS Target

A target is the basic unit of storage allocation and space management. It is associated with a reservation of persistent memory optionally combined with block-based storage. It is the unit of both performance and concurrency. A target has a fixed capacity expressed in bytes and fails operations when full. A target is a single point of failure and is responsible for data integrity. Checksums will be managed internally to detect and report corruption. Checksum management support is not in scope of ESSIO though.

3.1.3 DAOS Pool

A pool is a set of targets spread across different storage nodes over which data and metadata are distributed to achieve horizontal scalability and replicated or erasure-coded (the latter is not implemented in the ESSIO program) to ensure durability and availability. Each target is associated with a unique pool, which maintains the membership by storing persistently in the pool map the list of participants. At any point in time, new targets can be added to the pool map and failed ones can be excluded. Moreover, the pool map is fully versioned which effectively assigns a unique sequence to each modification of the map.

Upon target failure and exclusion from the pool map, data redundancy inside the pool is automatically restored online. This process is known as rebuild. Rebuilding progress is recorded regularly in special logs in the pool to address cascading failures. When new targets are added, data will be automatically migrated to the newly added targets to redistribute space usage equally among all the members. This process is known as space rebalancing and uses dedicated logs to support interruption and restart.

A pool will only be accessible to authenticated and authorized applications. Multiple security frameworks could be supported from simple POSIX access control lists to third party authentication mechanisms such as Kerberos. Security will be enforced when connecting to the pool. No security model is implemented yet as it was not in scope of ESSIO.

3.1.4 DAOS Container

A container represents an object address space inside a pool. A container is the basic unit of atomicity and versioning. Any time a container is opened, a handle for that container is returned to the user. All object operations are explicitly tagged by the caller with both the container handle and a transaction identifier called an epoch. Operations submitted against the same epoch and container handle are applied atomically to a container on a successful commit.

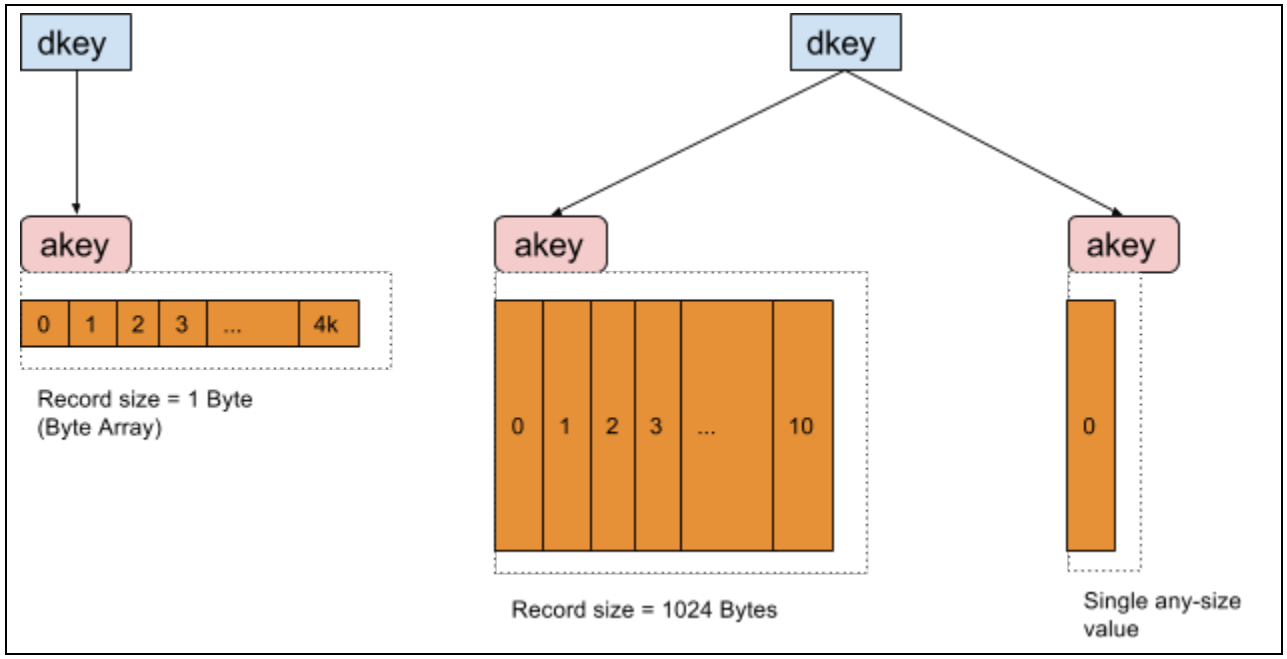
3.1.5 DAOS Object

The essence of the DAOS storage model is a key-array object providing efficient storage for both structured (fixed-size array element addressed by index) and unstructured (variable length data stored in first array index) data. Figure 3-3 below represents the DAOS object model. The object key is a 2 level key:

1. The Distribution Key (dkey) which determines placement. A user groups data under a single dkey to hint locality and colocation of that data on a single target.
2. The Attribute Key (akey) which identifies an array of values.

An array value is a blob with an arbitrary size (from 1-byte to multiple GB).

Figure 3-3. DAOS Object Model



To achieve high availability and horizontal scalability, many object schemas (replication/erasure code, static/dynamic striping and others) are provided. The schema framework is flexible and easily expandable to allow for new custom schema types in the future. The object schema is determined from the object class which is extracted from the object identifier that is used to open the object.

3.1.6 Transactional Model

The primary goal of the DAOS transaction model is to guarantee data model consistency with highly concurrent workloads. When using DAOS, applications would be able to safely update the dataset in-place and rollback to a known consistent state on failure.

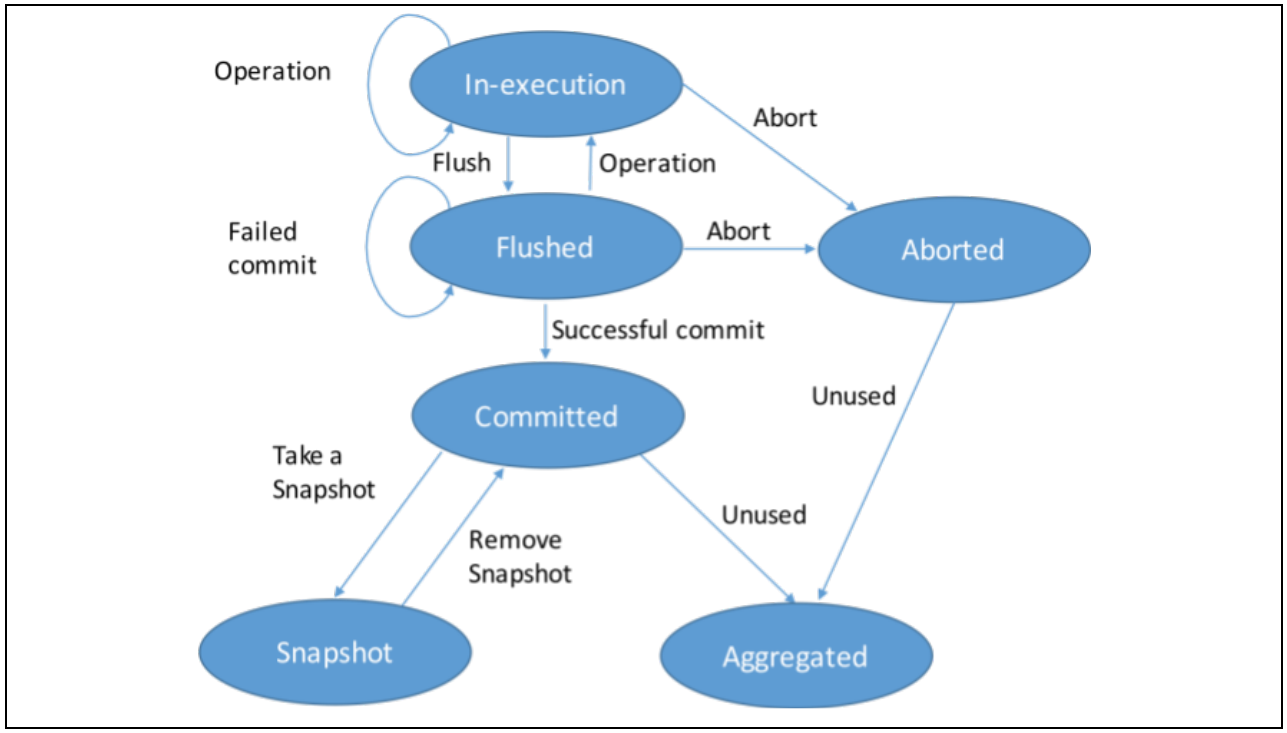
The DAOS transactional model allows applications to concurrently update a DAOS container through different transactional contexts by utilizing different container handles. All operations submitted with the same epoch and container handle are guaranteed to be atomically committed or aborted. Several applications or processes/threads within an application may independently open and access a container through different handles. DAOS tracks all I/O submitted with both the epoch state and the container handle.

This “all or nothing” semantic eliminates the possibility of partially integrated updates on a container handle. On a successful commit, an epoch is guaranteed to be immutable, durable and consistent. Unused committed and aborted epochs for a container may from time to time be aggregated to reclaim space utilized by overlapping writes and reduce metadata complexity. A snapshot is a permanent reference that can be placed on a committed epoch to prevent this aggregation. The user is responsible for conflicts (for example updating overlapping extents of an array in the same epoch), as DAOS will not have information at aggregation time to resolve such conflicts. The rest of this section provides more details on the transactional implementation in DAOS.

Each DAOS operation is assigned a unique identifier called an epoch. The lifecycle of an epoch is composed of four main phases represented in Figure 3-4:

- **In-execution:** the DAOS targets process the operations associated with the epoch and log the resulting distributed updates.
- **Flushed:** the updates are safely stored on persistent storage.
- **Committed or discarded:** the updates are integrated and made durable (the epoch is committed) or thrown away (the epoch is discarded).
- **Aggregated or snapshot:** if no persistent reference (i.e. a snapshot) is taken on an epoch, its updates may be merged with those from other epochs. Updates of an aggregated epoch cannot be dissociated any more.

Figure 3-4. Epoch Lifecycle



The following variables are associated with each container handle and tracked persistently by the storage system:

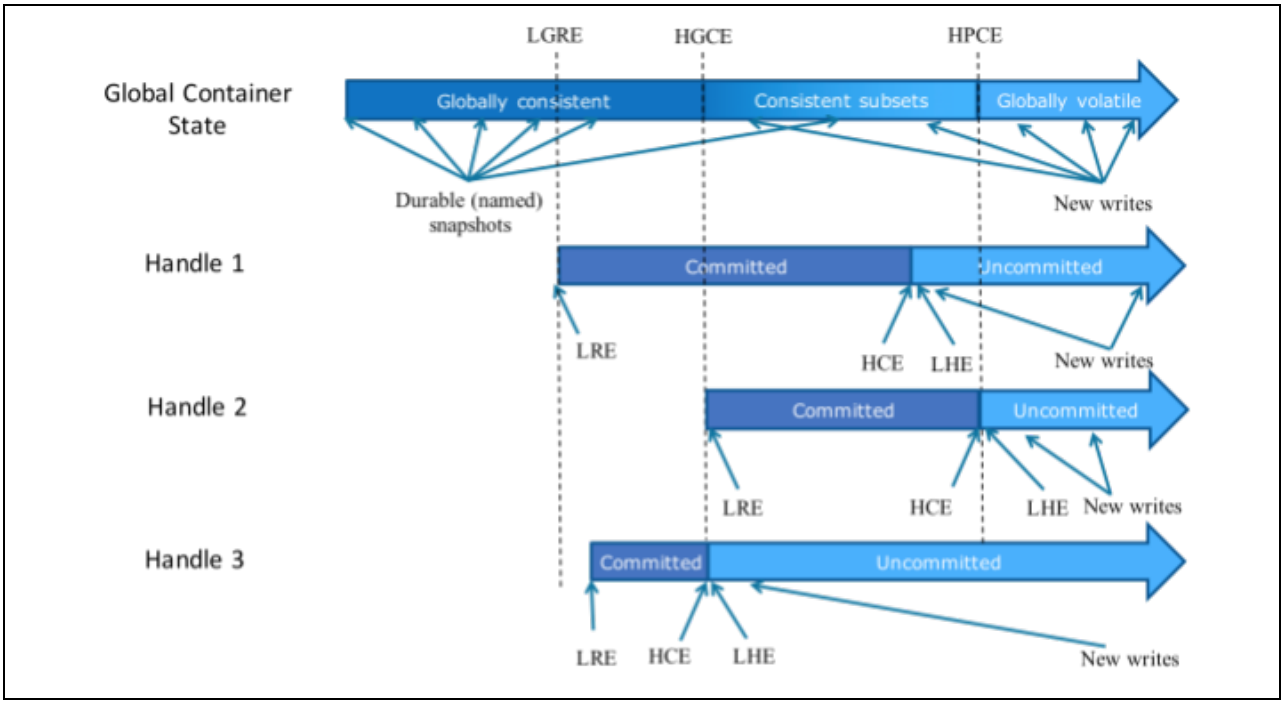
- *LHE*: lowest epoch held by this handle. The handle is expected to submit I/O operations for and to commit all epochs >= LHE. The LHE is increased automatically on commit to HCE+1 and can be obtained/released at any time with the hold operation.
- *HCE*: highest epoch committed by this handle. Any changes submitted by this container handle with an epoch <= HCE are guaranteed to be durable. On the other hand, any updates submitted with an epoch > HCE are automatically rolled back on failure of the container handle. The HCE is increased on successful commit.
- *LRE*: lowest epoch referenced by this handle. Any epoch >= LRE can be read. The LRE is moved forward with the slip operation.

Furthermore, the following global variables are maintained by the storage system and may be queried at any time:

- HPCE: the highest partially committed epoch, equal to max($\{HCE\}_h$)
- HGCE: the highest globally committed epoch, equal to min({min($\{LHE\}_h$) - 1, HPCE). This epoch is guaranteed to have immutable data.
- LGRE: the lower globally referenced epoch, equal to min($\{LRE\}_h$, epochs below the LGRE with no named snapshots can be aggregated.

Figure 3-5 represents the container with three active handles.

Figure 3-5. Transactional Model



3.1.7 Concurrency Control

Uncoordinated access from applications to the data in a DAOS container could present a conflict. A conflict in this context means an incompatibility or a clash that cannot be resolved without some information from the user. Updating a value for the same key from two handles in the same epoch is an example of a conflict. While DAOS epochs can be used to support Atomicity, Consistency and Durability guarantees, the Isolation property is considered beyond the scope of the DAOS transaction model. No mechanism to detect and resolve conflicts among different transactions as well as within a transaction is provided or imposed. Imposing such constraints could result in false sharing and it would be better to delegate that to the layer that has more information. The top-level API is thus responsible for implementing its own concurrency control strategy (e.g., two-phase locking, timestamp ordering, etc.) depending on its own needs. This means that concurrency control can be enabled only when required. DAOS provides some functionality to facilitate the development of conflict detection and resolution mechanisms on top of its transaction model:

- Middleware layered over DAOS can execute code on the DAOS target. This provides a way to run the concurrency control mechanism where the data is located.
- Changes submitted against any epoch can be enumerated, if the epoch has not been aggregated. This allows the development of an optimistic approach where conflict detection is delayed until its end, without blocking any operations. The transaction can then be discarded if it does not meet the serialization and recoverability rules.

DAOS still provides some basic I/O ordering guarantees. I/O operations submitted with different epoch numbers from the same or different process groups are guaranteed to be applied in epoch order, regardless of execution order. Concurrency control mechanisms that might be implemented on top of DAOS are strongly encouraged to serialize conflicting updates by using different epoch numbers in order to guarantee proper ordering. Therefore, conflicting I/O operations submitted by two different container handles to the same epoch is considered a programmatic error and will fail at I/O execution time. As for conflicting I/O operations inside the same epoch submitted with the same container handle, the only guarantee is that an I/O started after the successful completion of another one will not be reordered. This means that concurrent overlapping I/O operations are not guaranteed to be properly serialized and will generate non-deterministic results.

3.1.8 Fault Model

Unlike other parallel file system solutions, DAOS operates in a share-nothing model and does not require dual-ported storage devices for failover capability. This means that each DAOS target is effectively a single point of failure. DAOS achieves availability and durability of both data and metadata by providing redundancy across targets in different fault domains. A fault domain is a set of servers sharing the same point of failure and are thus likely to fail simultaneously. DAOS assumes that fault domains are hierarchical and do not overlap. The actual hierarchy and fault domain membership must be supplied by an external database used by DAOS to generate the pool map.

Pool metadata is replicated on several nodes from different high level fault domains for high availability, whereas object data is replicated or erasure-coded over a variable number of fault domains depending on the application needs. DAOS internal metadata (pool map, container list, epoch information, etc.) are replicated on a few nodes from distinct high-level fault domains. For very large configurations with hundreds of thousands of storage nodes, only a very small fraction of those nodes (in the order of tens) runs the metadata service.

DAOS delegates node aliveness monitoring to a dedicated service called RAS (Reliability, Availability and Serviceability). The RAS system collects information from multiple sources: baseboard management controllers (BMC), fabric, distributed software running on the cluster, etc. All this data is carefully analyzed and correlated by the RAS system, which makes authoritative and unilateral decision on node eviction. A subset of the storage nodes (typically the ones running the pool metadata service) is thus notified of node evictions in a timely manner and in consistent order via a collective communication. RAS must deliver notifications reliably and in consistent order to all processes registered to receive them. With a limited number of storage nodes, DAOS can afford to rely on a consensus algorithm (RAFT [18]) to reach agreement and to guarantee consistency in the presence of faults. Members of the metadata service elect a leader, which is responsible for processing new updates and servicing reads. Updates are validated once they have been written to a quorum of replicas. The leader sends periodic heartbeats to inform the other replicas that it is still alive. A new leader election is triggered if replicas do not receive any heartbeat from the current leader after a certain timeout.

3.1.8.1 RAFT

To provide high availability, the pool-map and other container services are replicated on a subset of storage nodes by RAFT. A Raft consensus based algorithm offers replicated service that tolerates failure of any minority of its replicas. By spreading replicas of each service across the fault domains, pool connections and container handles can therefore tolerate a reasonable number of target failures. A replicated service is built around a Raft replicated log. All service state updates are committed to the replicated log before being applied by any of the service replicas. Since Raft guarantees consistency among log replicas, the service replicas end up applying the same set of state updates in the same order. If the state updates are deterministic, service replicas go through identical state histories.

Among all replicas of a replicated service, only the current leader can handle service RPCs. The leader of a service is the current Raft leader (i.e., a Raft leader with the highest term number at the moment). Non-leader replicas reject all service RPCs and try to redirect the clients to the current leader to the best of their knowledge. Clients cache the addresses of the replicas as well as who current leader is. Occasionally, a client may not get any meaningful redirection hints and can find current leader by broadcasting to all potential replicas. Similar addressing problems also apply to the failover approach, for clients need to find out which machine the service has failed over to.

3.1.8.2 Isolation and Recovery

Once the RAS system has notified the pool metadata service of a failure, the faulty node must be excluded from the pool map. This is done automatically and silently if the DAOS pool has enough internal redundancy to cope with the failure. Otherwise, I/O errors will be reported to the application since data has been lost. Upon exclusion, the new version of the pool map is eagerly pushed to all storage targets. At this point, the pool enters in degraded mode that might require extra processing on access (e.g. computing data out of erasure code). Consequently, DAOS client and storage nodes retry RPC indefinitely until they are notified of target exclusion from the pool map. At this point, all outstanding communications with the evicted target are aborted and no further messages should be sent to the target until it is explicitly reintegrated.

All storage targets are promptly notified of pool map changes by the pool metadata service. This is not the case for client nodes, which are lazily informed of pool map invalidation each time they communicate with servers. To do so, clients pack in every RPC their current pool map version. Servers reply not only with the current pool map version if different, but also with the actual changes between the client and server version if the log size is reasonable. Consequently, when a DAOS client experiences RPC timeout, it regularly communicates with the other DAOS target to guarantee that its pool map is always up-to-date. Clients will then eventually be informed of the target exclusion and enter into degraded mode. This mechanism guarantees global node eviction and that all nodes eventually share the same view of target aliveness.

When a target fails and some objects were lost, provided these objects are using schema with data protection, DAOS can rebuild data for them on other surviving targets. Upon exclusion from the pool map, each target starts the rebuild process automatically to restore data redundancy. Each target scans its local object table to identify objects impacted by the target exclusion. Then for each impacted object, the location of the new object shard must be determined and redundancy of the object restored for the whole history (i.e. committed epochs) as well as for future transactions (i.e. uncommitted epochs). Once all impacted objects have been rebuilt, the pool map is updated a second time to report the target as failed out. Rebuild progress is recorded regularly in special logs in the pool to address cascading failures. The global log is stored in the pool metadata and tracks the overall rebuild status for this fault. It records the current status of each target and is used to inform the system administrator of global recovery progress across targets and to determine when all targets have completed resilvering. Moreover, each target maintains locally a more fine-grained log recording the list of objects to be rebuilt, what objects have been rebuilt already and which ones are in progress (potentially at which epoch and offset/hash).

This rebuild process is executed while applications continue accessing and updating objects. Reads will be in “degraded” mode until the failed target is effectively excluded. This means that any reads to the failed target will fail, but could be redirected to replica target or reconstructed from erasure code. As for writes to the failed target, they will be retried until the notification is received and they can be redirected to the failover target.

3.2 HDF5

HDF5 is a file format for array data that can be stored in a directory like structure. The data arrays themselves are called datasets and the directory objects are called groups. While the HDF5 library normally stores its data in a binary file using the native HDF5 file format, a new prototype feature called the Virtual Object Layer (VOL) allows HDF5 to interface with any arbitrary storage system, provided an appropriate plugin. A VOL plugin defines its own way of storing HDF5 data that does not need to use the HDF5 file format or even a traditional file system. HDF5 API calls that would normally cause I/O to the file are instead forwarded to the VOL plugin, which executes the requested operation using the storage scheme it implements on top of the storage system it uses.

DAOS, being an object storage system with unique capabilities that is very different from a traditional (POSIX-like) file system, is a prime candidate for a VOL plugin. In FastForward 1, we implemented a VOL plugin to interface with IOD and the old version of DAOS. For FastForward 2, we again decided to implement a VOL plugin to allow HDF5 to interface with the new version of DAOS, whose interface is very different from the previous version's.

3.2.1 Interface

While the previous IOD HDF5 plugin achieved its objectives, use of the plugin required substantial changes to HDF5 applications. Not only did most HDF5 calls need to be replaced with FastForward specific versions, the application also needed to keep track read context and

version numbers and coordinate these with other processes. The entire consistency paradigm, which determines when written data and metadata can be accessed, needed to be reworked from the application's perspective. This was particularly difficult for applications whose processes or execution threads are highly independent from each other, like Legion.

The new version focuses on simplifying the interface in order to minimize the changes needed to port applications to the new stack. While there are a small number of new API routines, there is no need to use special versions of existing routines. The container versions, or epochs, are hidden from the application, and it is not necessary to coordinate globally to achieve consistency. This was made possible primarily through the support, in the new DAOS stack, for concurrent use of multiple container handles for the same container, each with their own epoch stream.

One major difference from native HDF5 that remains is that, while all metadata write operations in native HDF5 must be collective, in the DAOS plugin they are by default independent. This aligns the plugin with the highly independent design of DAOS, and allows applications to leverage DAOS' capability for highly independent access to the container. We believe that giving applications the option to create file objects independently will allow developers more flexibility in creating applications and could be a major reason for them to adopt HDF5/DAOS as opposed to native HDF5. To ease application porting, there is also an option to use the more traditional collective metadata access.

3.2.2 Client Only Architecture

Another major difference from the IOD plugin is the move to from a client/server architecture to a client-only plugin. Since IOD and the old version of DAOS were too heavyweight to run on the compute nodes, the IOD plugin was divided into a client HDF5 plugin to run on compute nodes and server executable to run on I/O nodes. Operations issued by the application were packaged by the client and shipped to the server, where they were executed. Since the new version of DAOS was designed to run everywhere, including compute nodes, there is no longer a need for separate server code. The HDF5/DAOS plugin now calls DAOS directly on the compute node, simplifying the code and allowing DAOS to more intelligently route communication as needed.

3.2.3 Map Objects

The first FastForward project introduced a prototype for a new HDF5 file object, the map. Maps were again implemented for the new HDF5/DAOS plugin. A map object is a container for a flexible application-defined key-value store. When creating a map, the application defines the key datatype and value datatype, both HDF5 datatypes. The keys and values are then stored as data in the format defined by these datatypes, and API calls that interact with the map supply “memory” key and value datatypes, which must be compatible with those used to define the map, so HDF5 can convert between them. The map object is an object within the HDF5 group structure similar to datasets and groups.

4 Implementation

4.1 Overview

The ESSIO stack is radically different from the previous Fast Forward stack as most of the layers and dependency libraries were different. The HDF5 library now uses the DAOS client API directly without the need to function ship calls since the function shipping is done in DAOS. The prototype stack runs on a commodity cluster with a tmpfs backend with multiple clients and servers. Developers were also able to build and run the cluster on individual Linux* VMs. The build process is highly automated and well documented. This section describes how both HDF5 and DAOS were implemented and highlights major features and challenges at each layer.

4.2 DAOS

DAOS is designed from the ground up and is completely in user space. The DAOS client is implemented for minimum jitter to limit interference with the client application, so no threads are forked in the DAOS client side. The server side interfaces with the network and registers RPC handlers to process RPC requests from the clients.

4.2.1 DAOS Dependencies

DAOS requires a C99-capable compiler and the scons build tool. In addition, DAOS leverages the following open source libraries:

- MPI & PMIX [12]: In order to support multi-process launch and management, an MPI implementation is required. MPI is not leveraged in the DAOS core for other than that purpose; no process intercommunication is done with MPI. PMIx is required to demonstrate failure of an MPI rank without having all the MPI ranks abort. PMIx uses hwloc library.
- Argobots [13]: a lightweight, low-level threading and tasking framework used to manage DAOS threads on the server side.
- Mercury [14]: a user-space communication library designed for HPC systems. It provides an asynchronous RPC framework that abstracts native fabric interface through a Network Abstraction Layer (NAL) guaranteeing portability. Mercury supports both small data transfers for metadata operations and bulk data transfers for actual data. The Mercury interface is generic to allow any function call to be shipped.
- CART [15]: To serve as a network transport for DAOS, Mercury needed additional features that did not seem fit to add to the Mercury core. CART was designed on top of Mercury to provide an RPC framework with the following additional features:
  - Scalable collective communications with reply aggregation
  - Fault tolerance and RAS system integration
  - Request Formatting
  - Flow control

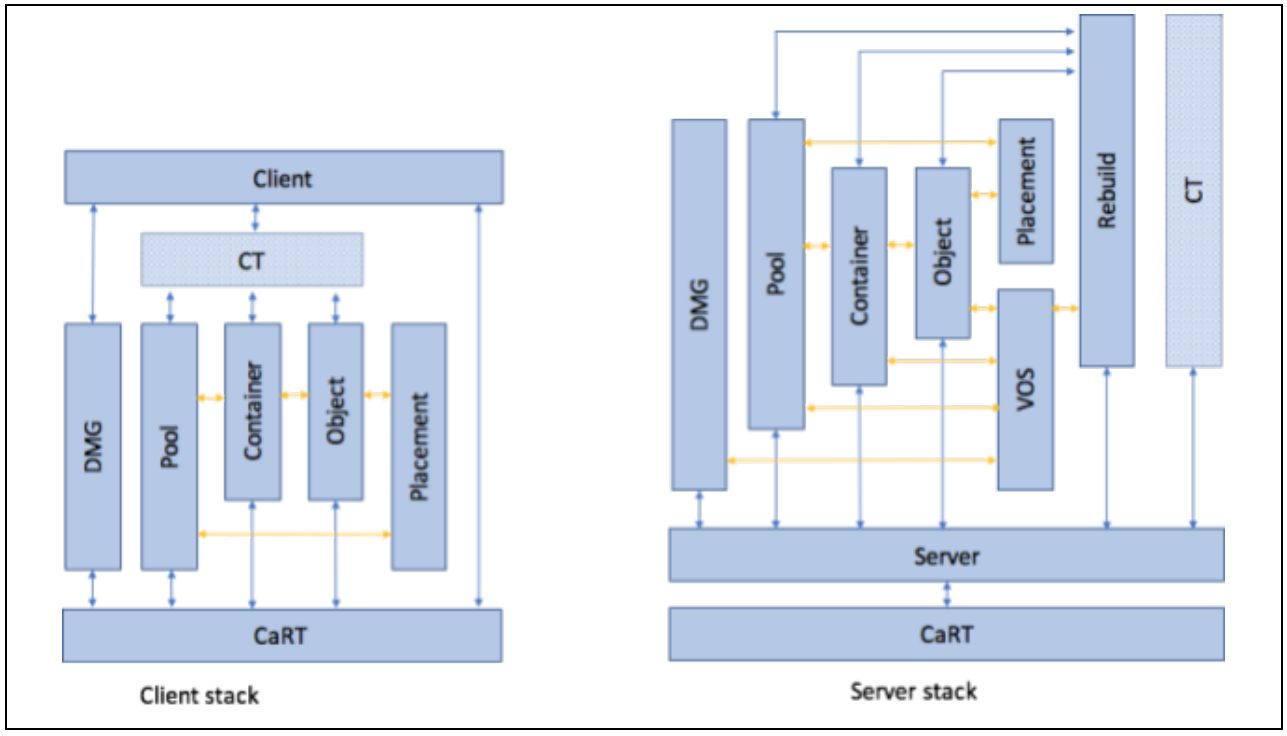
- OFI [16]: a framework focused on exporting fabric communication services to applications. The user space API is used through a Mercury NA plugin to ship RPC calls between the clients and the servers.

4.2.2 DAOS Layers

DAOS layering is shown in Figure 4-1. On the client, the application calls in through the DAOS API. Depending on what these calls are, they are routed to their specific module; for example management operations (like creating/destroying pools) are handled by the DMG layer, Pool connections are handled by the Pool layer, Object I/O operation by the I/O layer, etc. These layers can interact with each other in some cases. For example, when an object needs to refresh its pool version, it needs to interact with the pool layer.

On the server side, similarly, the DAOS operations come in from the RPC Layer through Cart/Mercury, and are handled by their respective module. Some layers like VOS and Rebuild are shared between the layers for functionality that will be explained in later sections.

Figure 4-1. DAOS Stack Layers



4.2.3 Client API

The DAOS client API is designed to provide maximum flexibility for the wide range of applications and I/O middleware libraries to build on. This comes at a cost of some complexity. The API includes methods to connect to pools, to add or disable targets, to open containers, to do object I/Os, to commit or abort epochs, to manage snapshots, etc. The client library implements many of these methods by invoking RPCs to the DAOS services.

The API also includes utilities that allow a group of processes to share a context established with the services by one member process. For example, after a process has established a connection to a pool, it may call a method to pack the information underlying the connection context into a buffer and broadcast the resulting buffer to all other processes with whomever it would like to share the connection. An actual implementation may choose to restrict the context to members of the same client process set. Each of the receiving processes may call a method to unpack the data and reproduce the connection context locally. All these processes then become a process group to the pool service—any member may act on behalf of the whole group. The following are the major components of the DAOS API:

- Pool API: exports function to create a pool and connect to it, disconnect, and destroy the pool. APIs to exclude targets or evict all connection to a pool are included.
- Container API: exports function to create, open, close, or destroy a DAOS container. In addition, properties of a container can be queried and returned to the caller as well as container attributes.
- Epoch API: export epoch management function to query the current state of a container through the container handle, hold a particular epoch for updates to that epoch and greater, commit or discard an epoch, and slip to a future epoch to trigger aggregation.
- Snapshot API: take a snapshot of a container at a particular epoch, destroy the snapshot and list epochs where snapshots have been taken.
- Object Class API: Register a new object class in addition to the default ones and query properties of existing classes.
- Object API: exports functions to manipulate DAOS objects, such as declaring and opening an object with a unique object identifier, closing an object, and punching keys/values from an object. I/O functions include updating and fetching entries on that object, and enumerating those entries.

Each object is identified in the container by a unique 256-bit object address. On object creation, the user should provide a unique 192-bit sequence number that will be completed with the encoding of internal DAOS metadata like the object type and class ownership to form the object address inside the container. It is thus the responsibility of the DAOS user to implement a scalable object ID allocator. This 192-bit identifier can be used by the upper layers of the stack to store their own object metadata, as long as it is guaranteed to be unique inside the object address space. On successful creation, the object identifier is the full 256-bit address. An object address is for single use only and can be associated with only a single schema.

4.2.4 Non Blocking API

All the DAOS APIs are exported with an option for executing the API in a non-blocking manner. Each API function has an event parameter, which can be NULL indicating that the call should not return unless completed (blocking). If a valid event is initialized with the DAOS event API, and passed to the API function, the function call is scheduled internally and returns immediately after the RPC has been fired. The progress on that call is made and queried

through the event queue API. An event queue is a data structure that tracks multiple events, and when polled on, makes progress on the RPC library and internal DAOS scheduler and returns events in its context that have completed.

The Event and Event Queue API is good for simple workflows, where the user is required to track dependencies between non blocking calls. For example, to open an object and update it requires the caller to first issue the object open, poll on the open to complete, and then issue the update call. While this model works fine for simple use cases, it does not easily support more sophisticated use cases and I/O middleware libraries, which develop asynchronous APIs on top of DAOS. Those users would be forced to create their own non blocking "engine" on top of DAOS to track DAOS calls and create dependencies between those calls.

Since DAOS internally implements a non blocking engine to track internal tasks, it seems fit to generalize that implementation and make it available for other users to use. The engine allows users to create any number of tasks with predefined body functions, and optional prepare and completion callbacks, set dependencies between those tasks to indicate the order of task execution (similar to a DAG), and make progress on the engine that holds those tasks. This generic engine can be used with any library and does not rely on the DAOS library.

We augmented the DAOS non blocking APIs with a few APIs that allow creating DAOS tasks for operations instead of calling the DAOS API functions directly. This allows users to create DAOS specific tasks and schedule them into a progress engine. In addition, the generic engine API can be used to create dependencies between the tasks which allows for applications and middleware libraries to schedule all their DAOS calls at the same time.

For example, instead of calling:

```
daos_obj_open();
poll();
daos_obj_update();
poll();
```

The user can create tasks:

```
t1 = daos_task_create(OBJ_OPEN);
t2 = daos_task_create(OBJ_UPDATE);
daos_task_set_dependency(t1, t2);
daos_progress();
```

4.2.5 DAOS Addons

As mentioned earlier, the DAOS API is very flexible and generic but comes with some complexity. More functionality and APIs can be built on top of DAOS, including a simplified data model and API. We group this functionality into an "addons" API that is part of the DAOS library, but is implemented using the DAOS core API. The addons include:

- DAOS Array API: A new API to create and access a 1-D array of any fixed-sized element. The array is implemented as a DAOS KV object and is distributed over a scaling set of dkeys, meaning as the array grows in size, the implementation will use more dkeys to ensure scalability. The element size and dkey size are specified on array creation and are

stored in a special dkey as metadata for the array object. Figure 4-2 shows a sample implementation of a Byte Array.

Figure 4-2. Array Mapping to DAOS KV

- The array API is straightforward to access the array without worrying about the mapping to the DAOS KV, and just basically exports accessing the array elements using a vector structure. Other management routines to set and get the size of an array are included, but not fully implemented yet. The Array API can be used in-turn to implement higher level middleware libraries like MPI-I/O and even POSIX access functions on top of DAOS.
- Simple DAOS KV API: A simpler API with put and get operations and a 1 level string key (dkey) would satisfy the need for many applications. These APIs will be implemented internally using the main DAOS KV APIs, and can be used with the DAOS library.
- Directory: A POSIX directory implementation in C using the DAOS KV API. This exports a namespace to users using the DAOS API and unifies the implementation for others to build upon.

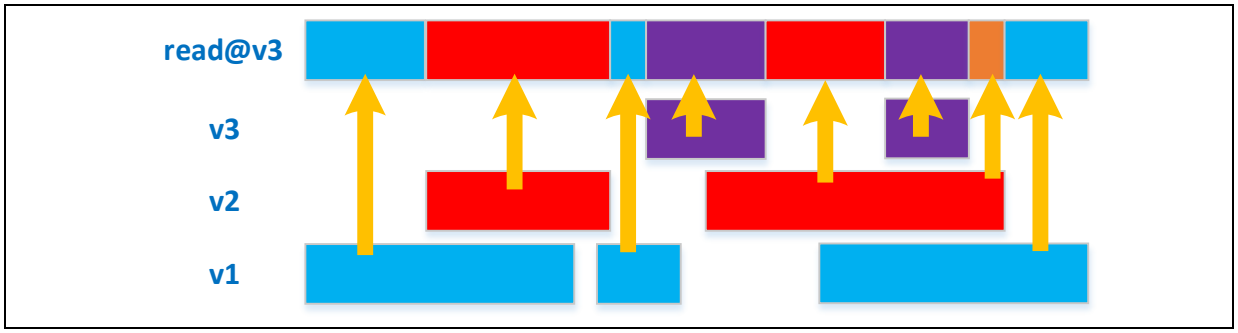
4.2.6 Versioning Object Store (VOS)

DAOS is byte granular, which means that it does not perform any read-modify-write internally and is not alignment or size sensitive. It provides a lockless consistency model at the byte granularity and thus avoids many of the scalability problems of traditional parallel filesystems or object stores. To achieve this, DAOS uses a persistent memory data model to store both internal metadata and byte-granular data in NVRAM. As shown in the figure below (Figure 4-3), all key-value updates are logged into a persistent index, which is directly mapped into the

process address space of the DAOS storage server. The Versioning Object Store (VOS) is the library implementing this persistent index over the NVML library.

The primary purpose of the VOS is to capture and log object updates in arbitrary time order and integrate these into an ordered epoch history that can be traversed efficiently on demand. This provides a major scalability improvement for parallel I/O by correctly ordering conflicting updates without requiring them to be serialized in time. For example, if two application processes agree how to resolve a conflict on a given update, they may write their updates independently with the assurance that they will be resolved in correct order at the VOS.

Figure 4-3. Key-value updates logged into a persistent index map



VOS uses two different types of data structures: B-tree for unstructured data and EV-tree for structured data. EV tree stands for Extent Versioning tree and is a new type of tree data structure based on rectangle trees. The primary difference between r-tree and ev-tree is that ev-tree splits overlapping rectangles, such that there is always just one valid lookup for an <extent, version> pair. It is designed to represent <extent, versions> in a 2D space and to reduce metadata space and improve performance efficiency for a versioned array of fixed-size records. The diagram below (Figure 4-4) is an example of EV-tree. In each node, the first range represent the index range and the second one is the epoch range in which the referenced extent is valid.

Figure 4-4. Example of an EV-tree for a versioned array of fixed-size records

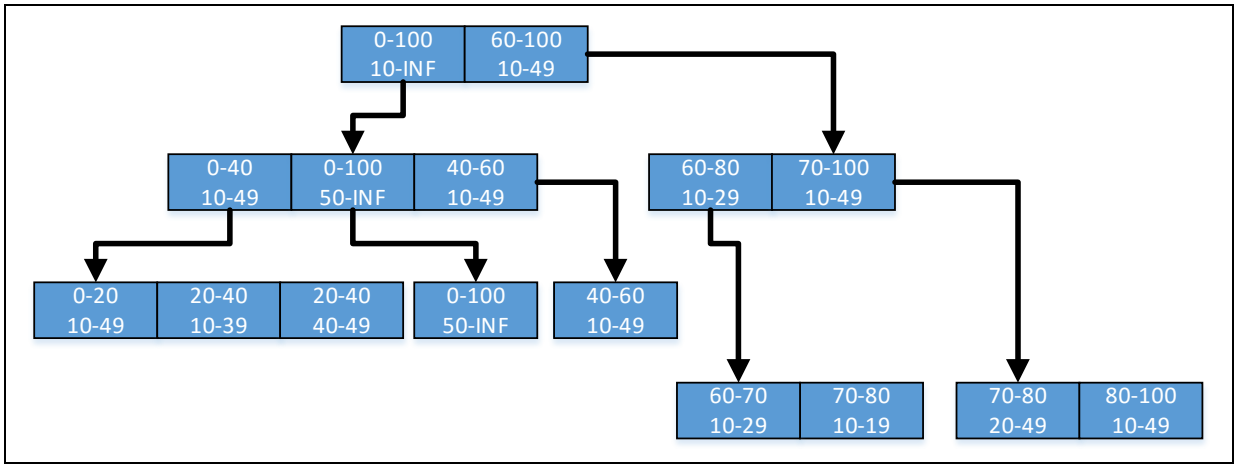
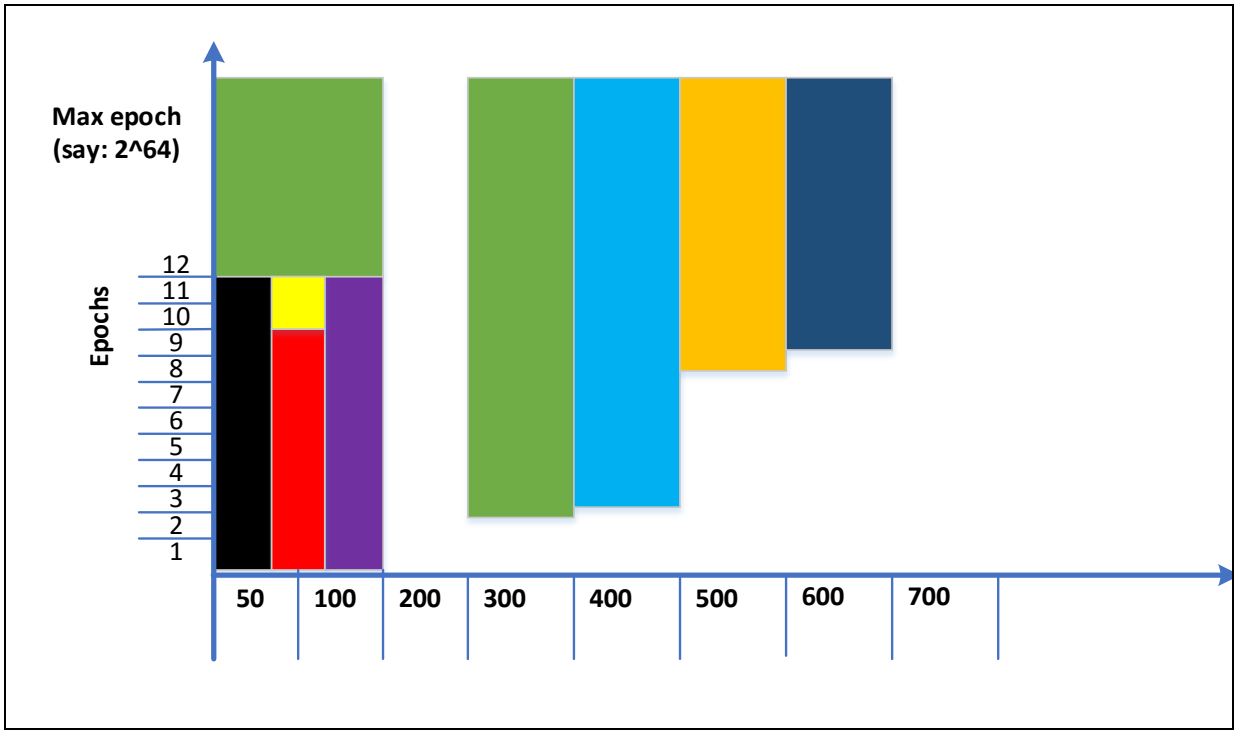


Figure 4-5. Versioned arrays in 2-D space for EV-tree



The size of the persistent index grows with the number of I/Os and epochs being used. DAOS tracks what epochs are used by application and pro-actively triggers epoch aggregation in the background to reduce the complexity of metadata and reclaim space. This process is done by starting a dedicated argobots ULT in charge of this flattening process on each storage servers. To avoid impacting I/O performance, the flattening ULT is throttled and is designed to run in the background when the system is idle. VOS also allows object updates associated with a given epoch to be discarded. This functionality ensures that when a DAOS transaction must be aborted, all associated updates are discarded before the epoch is committed and becomes immutable.

4.2.7 Object Schemas

The DAOS object schema describes the definitions for object types, data protection methods, and data distribution strategies. Based on these definitions and the requirements, upper layers can register a number of object classes. An object class has a unique class ID, which is a 16-bit value, and can represent a category of objects that use the same schema and schema attributes. A successfully registered object class is stored as container metadata; it is valid in the lifetime of the container. In addition, DAOS provides some predefined object classes for the most common use. While creating an object, the upper level stack of DAOS always needs to provide a registered class ID. DAOS uses this ID to find the corresponding object class, and

then distribute and protect object data based on algorithm descriptions of this class. On completion of object creation, DAOS adds the class ID into reserved bits of the object ID and returns it to the upper level stack. The upper layer should use the complete object ID to access this object in the future. For those classes predefined by DAOS, they are part of the common protocol that can be directly used between the DAOS client and server. For those customized object classes, they can never be changed after the registration, so both the DAOS server and client can cache them to reduce queries.

The current object classes provided by DAOS include:

- DAOS_OC_TINY_RW: stripe all keys of the object over 1 server only with no replication.
- DAOS_OC_SMALL_RW: stripe the object over 4 servers with no replication.
- DAOS_OC_LARGE_RW: stripe the object over all servers with no replication.
- DAOS_OC_REPL_2_RW: stripe the object over all servers with 2 way replication of the object shards.
- DAOS_OC_REPL_MAX_RW: stripe the object over 2 server with maximum replication of shards over all servers.

4.2.8 Replication

Replication ensures high availability of object data because objects are accessible while any replica exists. Replication can also increase read bandwidth by allowing concurrent reads from different replicas.

Client replication is done fully in the client stack to provide high concurrency and low latency I/O for the upper layer stacks. I/O requests against replicas are directly issued via the DAOS client; there is no sequential guarantee on writes in the same epoch, and concurrent writes for a same object can arrive at different replicas in an arbitrary order. Because there is no communication between servers in this method, there is no consistency guarantees if there are overlapped writes or KV updates in the same epoch. The DAOS server should detect overlapped updates in the same epoch, and return errors or warnings for the updates to the client. The only exception is multiple updates to the same extent or KV having the exactly same data. In this case, it is allowed because these updates could potentially be the resending/retrying requests.

Server replication has a stronger consistency of replicas with a trade-off in performance and latency. With a server-side protocol, because the client only sends updates to the primary replica, conflicting writes can be detected and serialized by the server, and it would be safe to have overlapped writes in the same epoch.

4.2.9 Rebuild

If a target fails and some objects are lost, provided these objects are using schema with data protection, DAOS can rebuild data for them on other surviving targets. Failure detection in DAOS relies on the RAS system for notification. When a target fails, the RAS system should detect it and notify the Raft leader about the failure. The Raft leader marks this target as failed

in the pool-map, increases the pool-map version, then propagate the updated pool-map to all targets in the same DAOS pool. Because there is no active pool-map notification for the DAOS client, when a failure happens, a client may not see the pool-map update event immediately. However, it can get the failure notification from the I/O reply if the peering target has learnt about the failure. If the pool-map update is tiny enough and can be embedded in the I/O reply, then the client can use the embedded information to update its pool-map right away. Otherwise, it needs to fetch the updated pool-map from the corresponding server. The DAOS client should then abort all inflight I/O requests against the faulty target, and switch to the new pool-map for all future I/Os. The client should not read from the failed target anymore and should write to the failover/spare target that is replacing the failed one.

Because DAOS is using an algorithm to compute object distribution, a target can compute the layout of an object using metadata that it has:

- the Object ID (Object class ID is embedded in object ID)
- The affinity target ID
- Hash stride

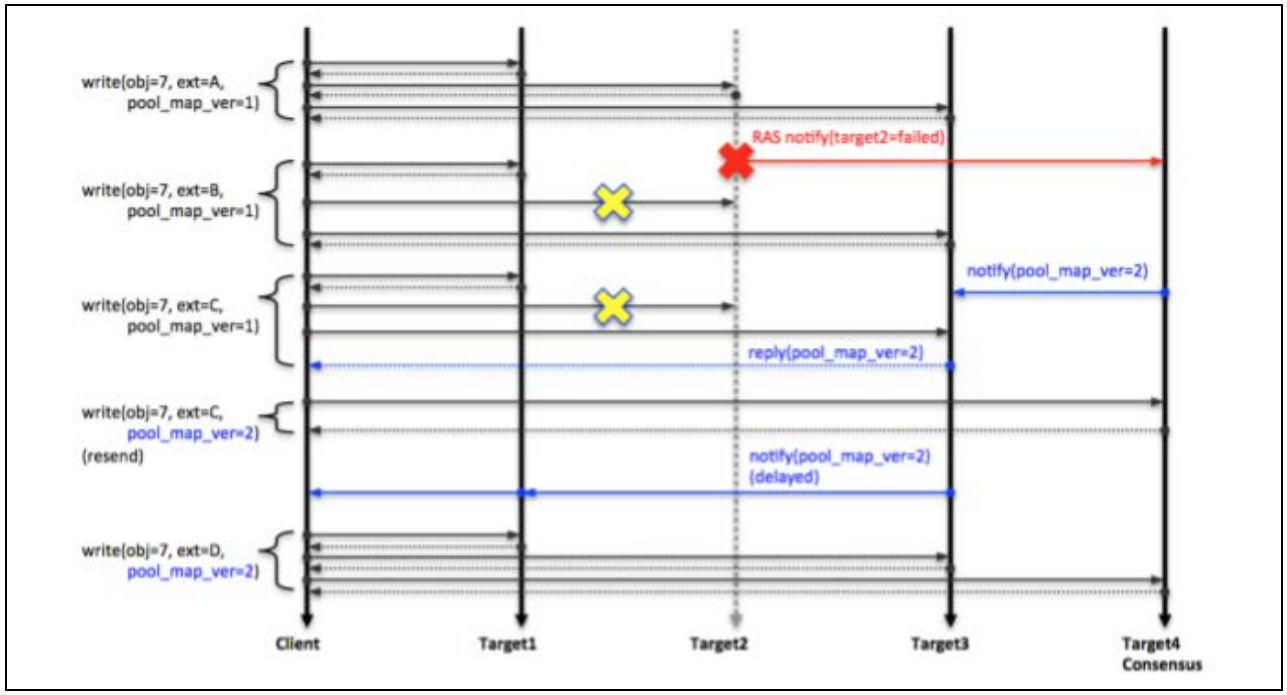
These metadata are tiny enough to be replicated for all object shards. Based on that metadata, any of the surviving targets can independently calculate out the redundancy group (RDG) members for its local object shards. If the RDG of a local object shard has a missing shard on the failed target, then data of that missing shard, in all existing epochs, should be rebuilt from the surviving shards in the RDG. To reconstruct missing object shards, DAOS uses replication.

For objects protected by replication, although a missing replica can be fully rebuilt from any surviving replica, to distribute rebuild workload and reduce impact on the overall performance, all surviving replicas should contribute to rebuild. Different replicas should concurrently send different portion of object data to the recovered replica. The strategy of sharing rebuild workload between different redundancy group members is straightforward: each of them should rebuild a sub-range of key on the destination target.

There is no synchronization between targets when rebuild is initiated, meaning when some targets have already started rebuilding, a few other targets may have not received the failure notification yet. The asynchrony of this protocol could result in data inconsistencies. This can be avoided by a client side protocol which we name Online rebuild. The Online rebuild protocol avoids inconsistencies during the rebuild phase by guaranteeing that new writes always reach the rebuilt object shard. If none of surviving targets in the redundancy group has learnt about the failure, it means none of them has started rebuild. Writes to the redundancy group will be rebuilt on the new shard later. As shown in Figure 4-6, while writing extent-B, target-2 has already failed, but none of these targets got the failure notification from the RAFT leader, so extent-B will be written to target-1 and target-3, and it will be eventually rebuilt on target-4. If any of surviving targets in the redundancy group has learnt about the failure, this target should notify the client about the failure. The client should abort the RPC to the failed target, and resend it to the failover target. In Figure 4-6, because target-3 has learnt about the failure, it can notify the client by piggybacking the notification in the reply. The client will find

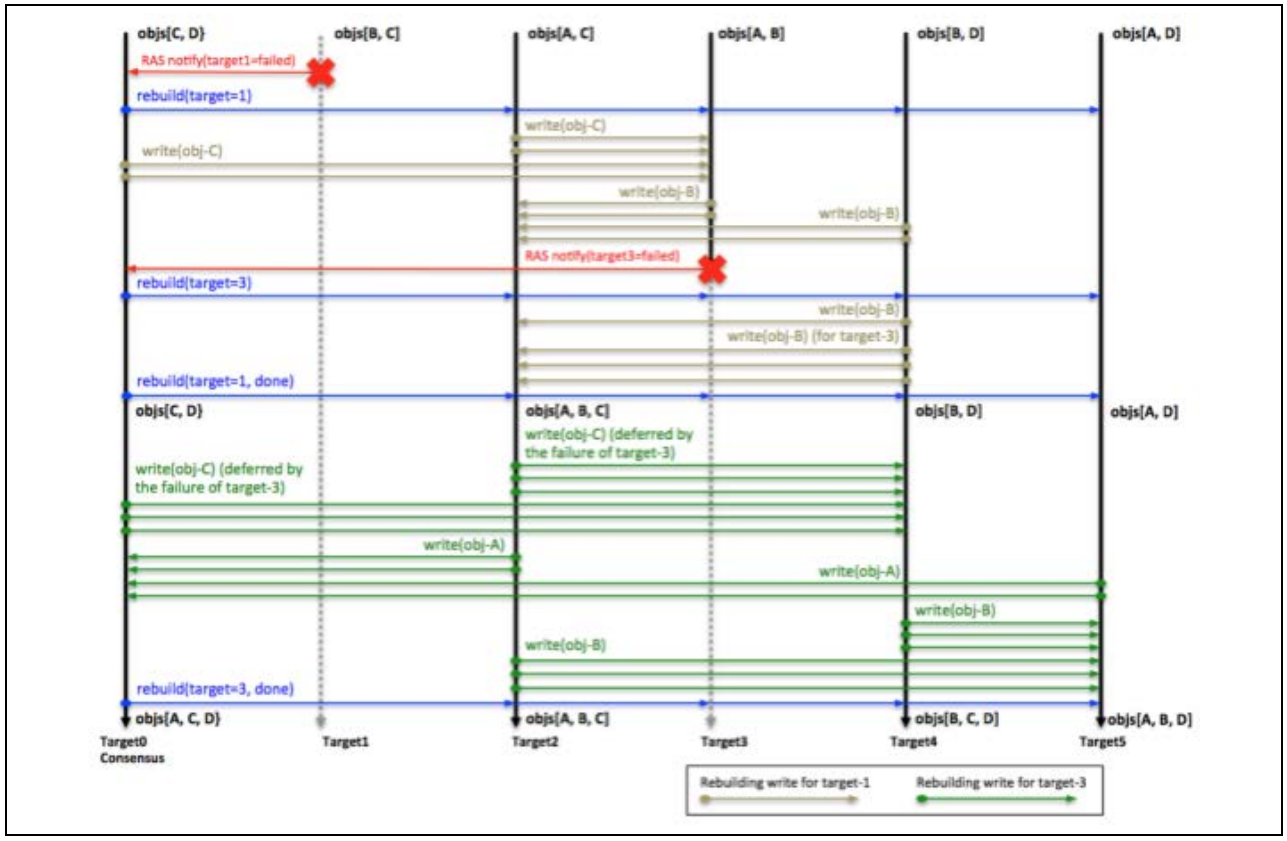
out that target-2 is gone, so it will resend extent-C to target-4. Therefore, this extent exists on all members of the new redundancy group.

Figure 4-6. Data Consistency with Online Rebuild



In case of more failures while rebuild for the first failure is still in progress, DAOS will not simultaneously handle these failures nor restart rebuild. Otherwise, rebuild time for each individual failure could be significantly extended and the rebuild process may never end if more failures happen. Figure 4-7 shows an example of a multiple target failures:

Figure 4-7. Multiple Target Failure Protocol



- If the second failed target is the destination of rebuilding data for the first failure, all rebuilding writes to the second failed target should be aborted. In the example above, target-3 failed while rebuilding data for target-1, so all writes to target-3 are aborted. With this approach, object-C cannot be rebuilt in the process of rebuilding the first failure. It will be deferred and rebuilt in the process of rebuilding the failure of target-3.
- If the second failed target is contributing to rebuild of the first failure, another target in the redundancy group should take over the workload of the second failed target. In this example, target-3 and target-4 were rebuilding object-B on target-2. When target-3 fails, target-4 fully takes over the workload of rebuilding object-B on target-2.
- If rebuild for the earlier failure is still in progress, no target should rebuild missing objects for the failures occurring after the first one. In this example, on completion of building the first failure (target-1), object-A, object-B and object-C only have 2 replicas; they lost the third replica because of the second failure (target-3).

Unrecoverable failures occur if the number of failed targets from different domains exceed the fault tolerance level of any schema within the container. For example, if there are a few objects protected by 2-way replication, then losing two targets in the different domains may cause data loss.

4.3 HDF5

Due to the fundamental changes to the DAOS interface since IOD in the first Exascale FastForward project, the decision was made to write a new HDF5/DAOS VOL plugin from scratch, repurposing sections of code from the IOD plugin as appropriate. The biggest change was the removal of the client/server architecture for the plugin, with all DAOS calls instead being made directly on the client. In addition, the interface is lower level, without the intermediate IOD interface. Starting from scratch proved to be beneficial because it allowed the plugin's internal design to match the new structure from the beginning. In addition, it gave us valuable insight into ways to improve the HDF5 virtual object layer (VOL) code, which is yet to be released.

While the goal is for VOL plugins to be external to the HDF5 library, and therefore possible to develop without knowledge of the internals of HDF5, it is currently still necessary to use some internal (private) library functions. The experience gained while developing the HDF5/DAOS plugin has given us more insight into what features need to be available as part of the public API. The most important areas that need to be improved are dataspace selection operations and datatype conversion. While there are public API functions available that deal with these subjects, it was still necessary to use some private functions and write a substantial amount of extra code in order to achieve full functionality. This experience will be used to improve the HDF5 API and VOL layer.

4.3.1 DAOS Mapping

The central paradigm for mapping HDF5 to DAOS is that HDF5 files are stored as DAOS containers, and HDF5 objects (groups, datasets, named datatypes, and maps) are stored as DAOS objects, both using a 1:1 mapping. All metadata and raw data associated with an HDF5 object is then stored as entries in the key value store associated with that object provided by DAOS.

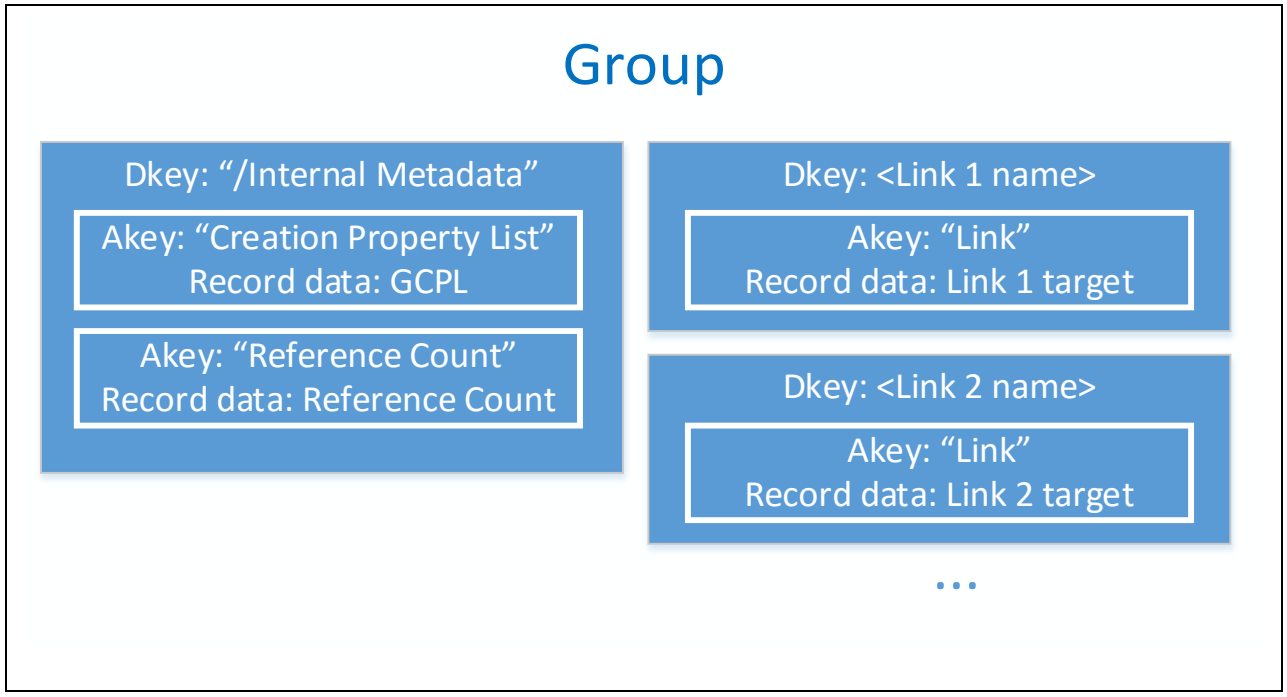
Since DAOS containers are identified by a UUID, the plugin uses a hashing function to generate a UUID from the file name. Each container for an HDF5 file contains at least two objects: the global metadata object and the root group. The global metadata object stores metadata that describes properties that apply to the entire file. Currently this is only the maximum object id. The root group is always the start of the HDF5 group structure, and uses the same format as all other groups.

Similar to the native HDF5 file format, the formats for the different object types are kept similar, with only differences as necessary to represent the specific object type. All object types have a creation property list and a reference count (not yet implemented) stored under the “/Internal Metadata” dkey and the “Creation Property List” and “Reference Count” akeys, respectively. All object types also have attributes stored under the “/Attribute” dkey. Each attribute stores its datatype under the “T-<attribute name>” akey, its dataspace under the “S-<attribute name>” akey, and its value under the “V-<attribute name>” akey.

4.3.1.1 Groups

The purpose of HDF5 groups is to store links to other objects, and each of these is stored under a dkey which equals the link name, with a constant akey ("Link"). Links can be hard, in which case it contains the object id for the target which can then be directly opened, or soft, in which case the link contains the path name of the target, which must be traversed to open the target.

Figure 4-8. Group Dkey and link examples



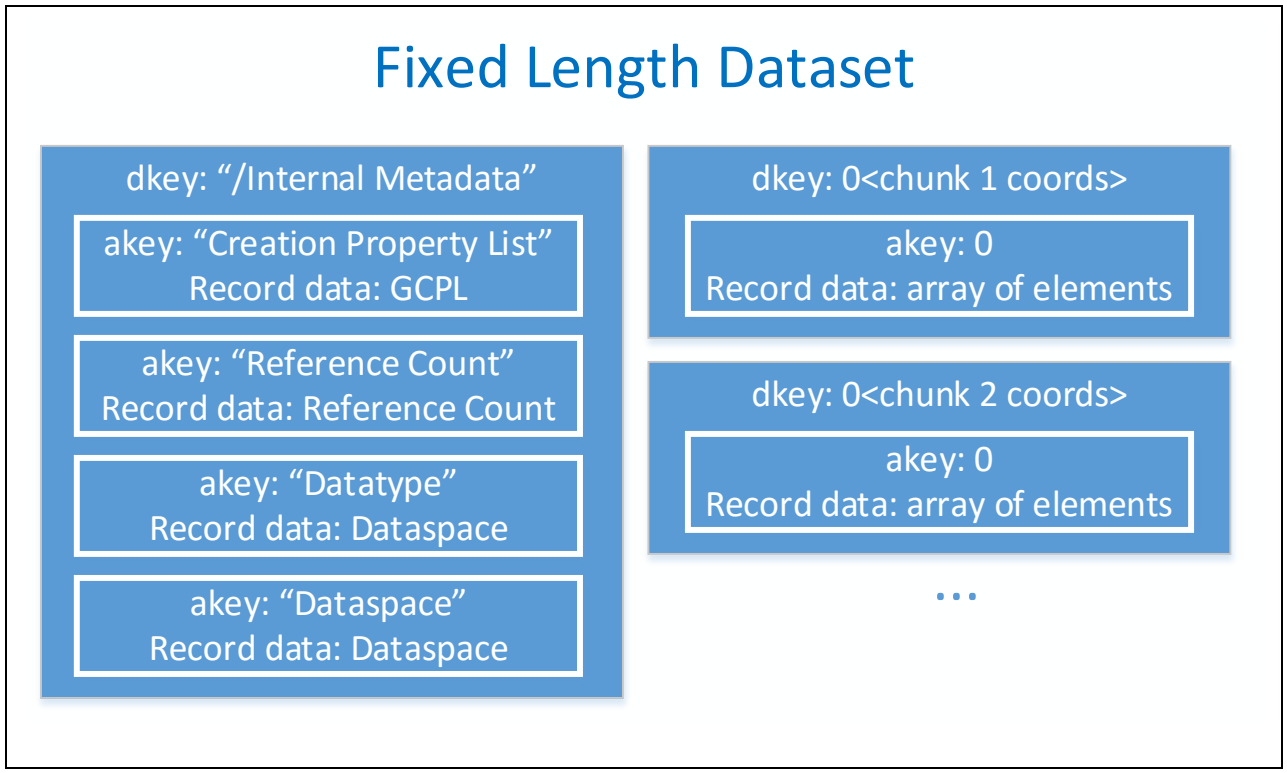
4.3.1.2 Datasets

Datasets are used to store the bulk array data in HDF5. In addition to the creation property list, these store the datatype and dataspace under the "/Internal Metadata" dkey and the "Datatype" and "Dataspace" dkeys respectively. Raw data is divided into "chunks" which are regularly spaced blocks of data, where the size of the block is specified by the application. Each chunk is stored in its own dkey. The dkey is encoded with a leading 0 byte, followed by <number of dimensions> 64 bit little endian unsigned integers which denote the chunk offset within the dataset. The "/" prefix for internal metadata and attributes prevents conflicts with arbitrarily named links, which cannot contain the "/" character, and the 0 prefix for chunks prevents the binary chunk dkeys from conflicting with the string metadata dkeys, since non-zero length strings cannot begin with 0 ('\0').

For datasets with fixed length datatypes, the data for the chunk is stored in a single akey (with a value of a 0 byte), as an array of records, where each record corresponds to an element in

the dataset. The record size is therefore equal to the size of the datatype and the number of records is equal to the number of elements in the chunk. For datasets with variable length datatypes, each element is stored in a separate akey, as a single record, whose size i equal to the total size of the variable length element.

Figure 4-9. Fixed Length Dataset Dkey and link examples



4.3.1.3 Committed Datatypes

Committed datatypes are an HDF5 datatype that has been stored as an HDF5 object in the group structure. The only object type specific data is the datatype itself, which is stored similarly to that of datasets, under the "/Internal Metadata" dkey and "Datatype" akey.

4.3.1.4 Map Objects

Map objects are an HDF5 object that contains an arbitrary application-defined key-value store. The map's key and value datatypes are stored under the "/Internal Metadata" dkey, and under the "Key Datatype" and "Value Datatype" akeys respectively. Values are stored under a dkey which is equal to the binary representation of the key, and under a constant akey ("MAP_AKEY").

4.3.1.5 Object IDs

DAOS objects are referenced through the DAOS API by a 192 bit object ID. Only the lower 64 bits are currently used by HDF5, the rest of the ID is set to 0 and/or set by DAOS to encode the

DAOS object class, which is always the same for each HDF5 object type. The lower 62 bits are simply set in increasing order, starting from 1, which is always the root group (0 is reserved for a global metadata object). The remaining 2 bits are used to encode the HDF5 object type. This removes the need to store the object type in the key-value store for the object itself, and allows the plugin to determine the object type and therefore the routines used to access it without needed to query DAOS, reducing the number of server requests needed for metadata operations. The lower 64 bits (including the encoded object type) are considered the "address" for the purposes of the HDF5 API, and is what is returned as "addr" from H5Oget_info and what is accepted for H5Oopen_by_addr.

Object ID allocation is currently not handled correctly when done independently by multiple processes. This is a temporary solution until a DAOS object ID allocator is available. This means that either all objects should be created by the same process, or they should always be created collectively using H5Pset_all_coll_metadata_ops().

4.3.2 Interface

4.3.2.1 Synchronization

While the previous IOD plugin gave the application direct control over DAOS epochs through the transaction and read context APIs, thereby leaving synchronization to the app, the new plugin simplifies this. The plugin now tracks the DAOS epoch internally, and commits this epoch when the file is flushed or closed. When a file is created or opened, all processes in the communicator used to create or open the file receive the same epoch number, which the library keeps internally. All subsequent operations use that epoch number, until H5Fflush() or H5Fclose() is called. Once H5Fflush() is called, the epoch is committed and the library increments the internal epoch number, and begins operating on the new number. When H5Fclose() is called with write access, the epoch is also committed. This means calls to that H5Fflush() and H5Fclose() must be collective across the communicator used to create or open the file

4.3.2.2 Collective Metadata Operations

Unlike in native HDF5, where metadata writes are always collective and metadata reads are by default independent, in HDF5/DAOS, all operations are by default independent. This gives application developers more flexibility, but represents a change in the programming model and can cause performance issues if all ranks issue the same operation. To solve these problems, we introduced an option to make metadata operations collective. This uses an existing function, H5Pset_all_coll_metadata_ops(). While in native HDF5, since metadata writes are always collective, this only affects read operations, in HDF5/DAOS this affects both read and write operations. Like in native HDF5, this function can be used to affect a single operation or all operations on a file, depending on the use of the property list passed to the function. Currently this function only affects object creation and opening, though it is planned to affect all other metadata operations as well. We may introduce an option to adjust reads and writes independently if we find it necessary, though for now it should be possible to

accomplish this by only calling this function on property lists for the operations that need to be collective, and using other property lists (or H5P_DEFAULT) for operations that are independent.

4.3.2.3 Multiple File Opens

It is possible for multiple process groups to independently access the same file, with each process group using a different communicator. In this case, each process group maintains its own epoch number and calls to H5Fflush() and H5Fclose() only need to be collective across the communicator for the process group. In the extreme example, each process may use MPI_COMM_SELF, thereby making each process a separate process group, and calls to H5Fflush() and H5Fclose() do not need to be collective. From a DAOS perspective, each process group operates on its own container handle.

Since the process groups operate independently from each other, it is possible for some groups' epochs to "run ahead" of others'. While the reader processes will probably want to read the latest data available, for consistency they will want to read from the highest globally committed epoch (HGCE), to avoid reading uncommitted data that may be incompletely written. If one writer process group is making fewer commits than others, its epoch numbers will be lower and it will therefore delay the availability of the other process groups' data to the readers. This delay will get worse over time. Therefore the application may wish to synchronize these epochs from time to time to ensure that readers can see the latest data. Currently the only way to do this is to close and reopen the file, though we plan to add an option to H5Fflush() to allow synchronization of epoch numbers across all process groups. When using this option, the process group's internal epoch will be fast forwarded to the highest partially committed epoch (HPCE)

4.3.2.4 Init/Term

The HDF5/DAOS plugin provides explicit initialization and termination functions, H5VLdaosm_init() and H5VLdaosm_term(). The purpose of H5VLdaosm_init() is to call daos_init() and connect to the DAOS pool. This function therefore needs to be given a communicator to use to collectively connect to the pool, a pool uuid, and a pool group. Since daos_init() currently needs to be collective across MPI_COMM_WORLD, the communicator passed to H5VLdaosm_init() must currently be MPI_COMM_WORLD, and all processes must call it, even if they will not participate in I/O. In the future, we plan to allow non-I/O processes to avoid connecting to the pool by allowing these processes to pass MPI_COMM_NULL. The same pool is used for all files accessed by the process.

While use of H5VLdaosm_init() is mandatory, H5VLdaosm_term() is optional. The purpose of H5VLdaosm_term() is to disconnect from the pool and call daos_fini(). This is normally called automatically when the HDF5 library shuts down but it can be called explicitly if the application needs to control the order in which things shut down

4.3.2.5 Snapshots

One advantage of DAOS is the ability to easily access earlier versions (epochs) of the container (HDF5 file). In the previous HDF5 IOD plugin, this was facilitated through the read context API. Since that has been removed from the new HDF5 DAOS plugin, we have added a snapshot API to allow access to earlier versions and to inform DAOS to preserve versions. When the application wants to preserve a version of the file for later use, it calls H5VLdaosm_snap_create(), which returns a snapshot id (currently just the epoch number), and when that epoch is committed, the plugin will tell DAOS to create a snapshot at that epoch. To access that snapshot at a later time, the application calls H5Pset_daosm_snap_open() with the previously returned snapshot id and a file access property list. Files opened with that property list will then be opened at that snapshot (in read only mode). Snapshots are not yet supported in DAOS, so for now all epochs are saved, allowing the HDF5/DAOS snapshot API to function as intended.

4.3.3 Map Objects

Due to DAOS's fundamental nature as a key-value store, the implementation for map objects is a straight translation from the abstracted HDF5 map key-value store to the key-value store provided by DAOS. This greatly simplified the implementation as there is no need to implement an index, ey search, or value storage. Operations supported are H5Mset() to set a key-value pair, H5Mget() to get the value for a key, H5Mexists() to check if a key has been set, and H5Mget_count() to get the number of keys. A function to iterate over all keys, H5Miterate(), was planned in the first FastForward 1 project but never implemented. It has also not been implemented in the new plugin but could be added in the future.

4.3.4 Asynchronous Operations

Asynchronous I/O operations allow an application to decouple its computation and I/O operations, providing a greater overall utilization of system resources. Asynchronous I/O operations were added to the HDF5 interface as part of the first FastForward I/O project, and have demonstrated this benefit to applications.

In the previous implementation, however, the asynchronous I/O and transaction interfaces were independent of one another. This has been a source of confusion for DOE stakeholders and based on their feedback we proposed exposing the asynchronous I/O interface to the user and hide the transaction objects within HDF5. Users are able to choose whether all the operations in an entire transaction are synchronous or asynchronous, but are not able to choose to make individual operations synchronous or asynchronous. Operations that have not completed in an asynchronous transaction will be finished before the transaction commit operation reports completion. Furthermore, if any operations have failed during the transaction, the transaction will not be able to be committed unless those operations have been retried successfully.

4.3.4.1 HDF5 asynchronous I/O problem

One of the main problems when doing asynchronous I/O is to make sure that the operations submitted can actually progress. One of the problems when transforming HDF5 synchronous operations on top of DAOS non-blocking calls into asynchronous operations is illustrated below (Figure 4-10):

Figure 4-10. HDF5 synchronous operations on top of DAOS non-blocking calls

Application I/O Operation (non-blocking) /* container/object creation/open */ 1- file = H5Fcreate(...); 2- group = H5Gcreate(file, ...); 3- dataset = H5Dcreate(group, ...); /* object access */ 4- H5Dwrite(dataset, ...); /* container/object close */ 5- H5Dclose(dataset); 6- H5Gclose(group); 7- H5Fclose(file); 1 – 2 – 3 – 4 – 5 – 6 – 7 (DAG operation order cannot change)	
(1) is composed of multiple non-blocking DAOS calls, (1) cannot be progressed/complete until progress is made on the first DAOS call internally. Consequently, no progress on (4) (the main write operation) can be made until (1) / (2) / (3) complete. Two components are therefore needed to implement asynchronous I/O within HDF5: 1. An “Event stack” or “Completion queue” that the user can use to track completion of operations; 2. A progress engine that tracks internal dependency and allows progress to be made on both the DAOS and HDF5 operations.	

We originally considered three solutions for a progress engine:

1. Use the existing DAOS polling mechanism through DAOS completion queue and progress HDF5 operations as DAOS operations complete (no thread / but dependency tracking required) using callbacks;
2. Schedule HDF5 operations or sub-operations for execution into a thread pool while using blocking DAOS calls (dangerous as this can create a potential locking/contention when reaching the size of the thread-pool);
3. Schedule HDF5 non-blocking operations / non-blocking DAOS tasks into a thread pool and track dependencies (thread required / dependency tracking required).

The first solution seems preferable but gets back to the initial write progress issue described above unless the user explicitly launches a progress thread or regularly waits and makes progress on operations, this is for now left upon the application to decide.

4.3.4.2 Asynchronous I/O and Transaction Abstraction

One of the key design decisions is to prevent HDF5 API modifications as much as possible (as opposed to what was done in the previous FastForward prototype): this is enabled by creating a context of execution. Progress is made on a context, which is composed of non-blocking HDF5 operations. One context is internally assigned one transaction: to hide the transaction model from the user, the DAOS epoch is held on file create/open and committed/increased on file flush/close operations.

The example below shows how full asynchrony of HDF5 operations is enabled using a context.

Figure 4-11. Full asynchrony of HDF5 operations is enabled through the use of a context

```
Example of HDF5 I/O Operation using HDF5 Context (updated / modified HDF5 API with context)

/* container/object creation/open */
1- context = H5CXcreate();
2- file    = H5Fcreate(..., context); /* Hold epoch */
3- group   = H5Gcreate(file, ..., context);
4- dataset = H5Dcreate(group, ..., context);
/* object access */
5- H5Dwrite(dataset, ..., context);
/* container/object close */
6- H5Dclose(dataset);
7- H5Gclose(group);
9- H5Fclose(file); /* Commit epoch */
10- H5CXclose(context); / H5CXcancel() / H5CXwait();
```

Creating a context internally instantiates a new DAOS event queue. The event queue is then used to poll DAOS generated events from posted DAOS operations. Note that new VOL callbacks have also been added to support the DAOS plugin context/request creation and progress. Progress is made in two stages: first by progressing DAOS operations and retrieving completed operations from the DAOS event queue; second by progressing HDF5 operations themselves, i.e., by marking these operations as completed once the underlying DAOS operations have completed and advancing the progress engine to treat the next HDF5 operation dependency.

4.3.5 Features Supported

5 Technical Findings

5.1 Overview

The ESSIO team learned from several lessons that were encountered during the earlier Fast Forward storage project where several team members were the same. There was some knowledge this time on what the layers will expect in terms of features needed. This made the integration process easier in ESSIO. However, as when designing and implementing new software, there are always issues that will come up, especially when supporting new applications and middleware I/O frameworks that the team haven't worked with before. Overall the team did well in addressing those challenges and communicating between the members internally and externally to the stakeholders.

5.2 DAOS

5.2.1 Large vs Small Record Sizes

The DAOS KV object supports two types of record values, Single and Array values. Array values, as the name suggests, contain multiple indexed records that can be accessed by specifying any range with any index. The Single value is 1 atomic value that is always overwritten entirely when update.

The first version of the API didn't clearly distinguish the two types and how they are accessed. An API change was introduced to have the caller indicate the type of the record and specify a single record size for all records in the array or the single value to be accessed.

The implementation at the server side used a BTree to index every record. This works well if the records are large. However, in many cases, including the HDF5 applications that were ported in this project, small records were used (integer/float Datasets). Tracking every record in a BTree entry was very inefficient, so for array type values, where the records are typically small, DAOS switched to use a flavor of Rectangle Trees called EVTree to track extent of records instead of every record. Single records will be tracked with the original BTree algorithm since those values are usually large and accessed by themselves.

5.2.2 Transactional Model

The original transactional model in Fast Forward one required a lot of synchronization and communication between the clients accessing the container, and prevented disaggregated clients from accessing the same container concurrently despite the fact that they guarantee that overwrites to that container would not happen. DAOS in the new ESSIO stack relaxes this requirement significantly by allowing multiple independent container handles to the same container to be accessed, each with its own set of local epoch properties.

This design issue was encountered during the project when porting the Legion app on top of HDF5 using the old stack, and as detailed earlier in this document, the DAOS transactional

model was adjusted to be more flexible to allow workflows and middleware frameworks such as Legion to utilize the ESSIO storage stack.

5.3 HDF5

Overall the HDF5/DAOS VOL plugin was a success. The simpler concurrency model, facilitated by the more flexible epoch model, greatly simplified application porting to the new stack. The support for multiple independent file opens, also facilitated by new DAOS features, enabled the porting of applications like Legion, which had previously proved impractical. Overall the plugin is now much more approachable and easier for application developers to experiment with. The work on the DAOS/HDF5 plugin in parallel with DAOS development also proved valuable for DAOS itself. Several bugs were discovered, and feedback was regularly exchanged between the DAOS and HDF5 teams based on what discovered while developing for the DAOS API

Due to the idempotency of DAOS calls, some programming errors now produce confusing results, making debugging more difficult. For example, attempting to create a dataset that already exists does not immediately cause an error. This could be fixed, however, by implementing a debugging/development mode for the plugin that would query the server in such cases to see if the object had already been created, or the key written to, or other similar issues. In addition, some features, such as object creation order, may prove difficult to implement without extensions to DAOS.

5.3.1 Legacy Capabilities

The legacy capabilities (capabilities also present in native HDF5) of the DAOS plugin are:

- Files
  - H5Fcreate()
  - H5Fopen()
    - Read only access
    - Read-write access
- Groups
  - H5Gcreate()
  - H5Gopen()
- Datasets
  - H5Dcreate()
  - H5Dopen()
  - I/O
    - Partial I/O
    - Datatype conversion
  - All datatypes except references
    - Variable length types must not be a member of a compound or array, cannot contain a variable length type in the base type, and there cannot be conversion of the base type
  - H5Dget_space/type/create_plist/access_plist()

  - Chunking
- Attributes
  - H5Acreate()/H5Acrete_by_name()
  - H5Aopen()/H5Aopen_by_name()
  - I/O
    - Datatype conversion
  - Same datatypes supported as datatsets
  - H5Aiterate()
  - H5Aget_name()H5Aget_/type()/H5Aget_space()/H5Aget_create_plist()
- Links
  - H5Lcreate_soft()
  - H5Lexists()
  - H5Literate()
- Committed Datatypes
  - H5Tcommit()
  - H5Topen()
  - H5Tget_create_plist()
- Generic Objects
  - H5Oopen()/H5Oopen_by_addr()
  - H5Oget_info()
- Collective metadata write (optional in plugin, mandatory in native HDF5)
- Option for collective or independent metadata read

5.3.2 New Capabilities

The new capabilities of the DAOS plugin not present in native HDF5 are:

- Map objects
  - H5Mcreate() (creat map)
  - H5Mopen() (open map)
  - H5Mset() (set relation)
  - H5Mget() (get value for key)
  - H5Mexists() (check if key exists)
  - H5Mget_types() (get map datatypes)
  - H5Mget_count() (get number of keys)
- Snapshots
  - H5VLdaosm_snap_create() (create snapshot)
  - H5Pset_daosm_snap_open() (open snapshot)
- Independent file open
- Independent metadata write (by default, option to make it collective)

6 Application I/O Strategies

This section discusses the strategies involved in porting scientific applications to DAOS. Section 6.1 gives an overview of porting an application, *HACC*, to the IOD/DAOS stack. Section 6.2 discusses porting *CLAMR* to use HDF5 DAOS and gives an overview of the usability and capabilities from the perspective of a typical application code. The application *Legion*, Section 6.3, demonstrates using DAOS in a data-centric programming model. The last two applications netCDF-4 and PIO, Sections 6.4 and 6.5, are higher-level I/O libraries built on top of HDF5 and show the utilization of a higher level of abstraction to simplify an application's interaction with HDF5 and DAOS, which in turn should ease the transition to DAOS.

All development and benchmarks described in Sections 6.2-6.5 were run on a small Intel DAOS prototype clustre, *Boro*. Boro consists of Intel® Xeon® processor E5-2699 v3 with two CPUs per node. The DRAM (Kingston* KVR21R15S4/8) is being used in place of the NVRAM, (Figure 3-1), and the final storage stage uses Seagate* Constellation* ES.3 ST1000NM0033 HDD.

Before presenting the applications that were evaluated and ported to utilize the new I/O software stack, it is worth mentioning that both the DAOS library and the HDF5 DAOS backend are still in a prototyping phase. The primary goal of this evaluation was to prove that utilizing the new I/O stack is doable and easy once a middleware library is properly designed on top of DAOS. Tuning for optimal performance in both HDF5 and DAOS was not done as part of this work. Therefore it is unknown how the following issues would affect the performance.

- All the applications, except CLAMR, utilizing DAOS did not use HDF5 chunking layout for dataset storage. Thus, only a single DAOS server with one service thread for I/O is utilized because in this case an HDF5 dataset is mapped to one DAOS object distribution. This undoubtedly will result in a bottleneck once the number of I/Os are increased.
- Communication to the DAOS server uses libfabric over TCP, whereas communication to the Lustre server is done over InfiniBand verbs.
- Lustre servers use spinning disk for storage, but DAOS used a *tmpfs* file system over DRAM (since NVRAM is not available for this cluster).
- The MPI-I/O driver that HDF5 uses on Lustre will aggregate small I/Os at the client side before submitting I/O to the Lustre servers. No such aggregation is available yet in the HDF5 DAOS plugin nor at the DAOS client, which is something that will be explored in the future.

6.1 HACC application

HACC (Hardware/Hybrid Accelerated Cosmology Code) [2] is an N-body cosmology code framework where a typical simulation of the universe demands extreme-scale simulation capabilities. However, a full simulation of HACC requires terabytes of storage and hundreds of thousands of processes, which far exceeds the computational resources available for this

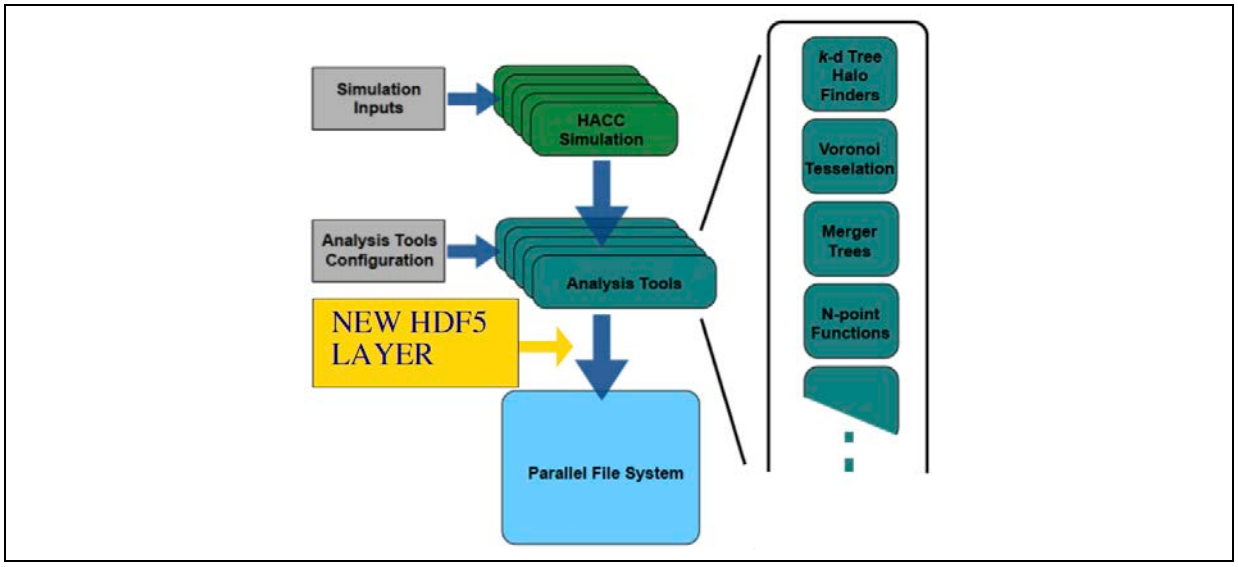
research. Consequently, a smaller I/O code, GenericIO, was used to mirror the I/O calls in HACC without the need to run an entire simulation. In its current implementation, all the "heavyweight" data is handled using POSIX I/O, and the "lightweight" data is handled using collective MPI-IO.

Critical features for HACC's I/O include:

- Resiliency for data verification
  - Checksumming from the application's memory to the file and vice-versa,
  - Mechanism for retrying I/O operations,
- Sub-filing

  Should avoid penalties in the file system associated with locking and contention,
- Self-describing file

Figure 6-1. HACC I/O scheme with HDF5 (hack_pflops.pdf, 2013)



A key component of the HACC code suite is the ability to do in situ data analysis. Performing data compression and data analysis before the output is dumped to the file system can reduce the storage requirements from petabytes to terabytes (Figure 6-1).

As for I/O strategies, the HACC team [1] found that creating one output file per process resulted in the best write bandwidth compared to other methods because it eliminates locking and synchronization between processors. However, this method is not used due to several issues:

- File systems are limited in their ability to manage hundreds of thousands of files,
- In practice, managing hundreds of thousands of files is cumbersome and error-prone,
- Reading the data back using a different number of processes than the analysis simulation requires redistribution and reshuffling of the data, negating the advantage over more complex collective I/O strategies.

The default I/O strategy in HACC is to have each process write data into a distinct region within a single file using a custom, self-describing file format. Each process writes each variable contiguously within its assigned region. On supercomputers having dedicated ION, HACC instead uses a single file per ION. The current implementation of HACC provides the option of using MPI I/O (collective or non-collective) or POSIX (non-collective) I/O routines. Additionally, GenericIO implements cyclic redundancy code (CRC) by adding it to the end of the data array being written [1].

Since HDF5 is a self-describing hierarchal file format, much of the "metadata" used in the current implementation of HACC, was automatically handled by HDF5. Thus, using HDF5 greatly reduces the internal bookkeeping and file construction required by HACC when compared to using POSIX or MPI-IO.

Algorithm 1 HACC I/O scheme

Require: Initialize DAOS stack

while Output variables remain **do**

Establish a read context for operations on HDF5 file

Create a transaction object

Start a transaction

Create HDF5 objects associate with variable

Write variable

Finish transaction

Close transaction

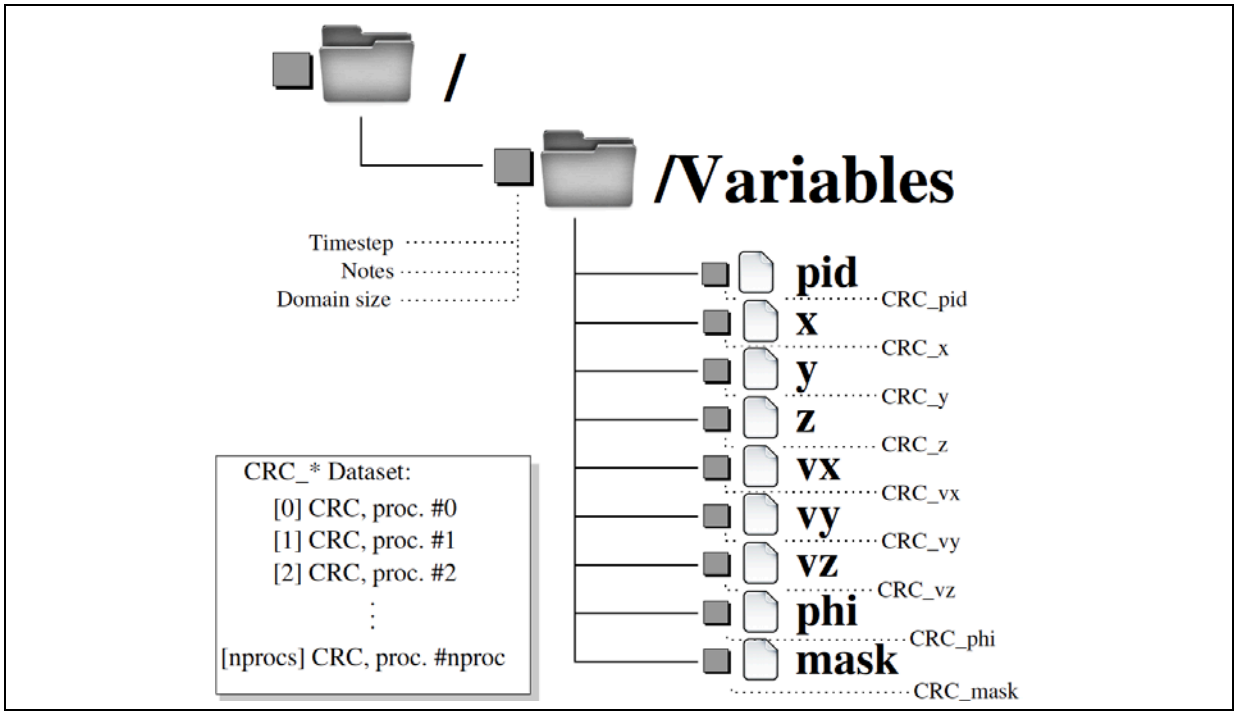
Release container

end while

Require: Finalize DAOS stack

The use of offsets as pointers to variables (note that these offsets are stored within the HACC file) is eliminated in the HDF5 implementation by instead using datasets to store variables. The HDF5 implementation stores HACC variables as datasets in the file's root group (/) (Figure 6-2). Attributes at the root level store time step information, such as the time stamp and notes about the simulation parameters. Additional attributes can easily be added if needed. All the datasets are stored under the 'variables' group.

Figure 6-2. Proposed HACC/HDF5 file structure



Associated with each variable's dataset is the cyclic- redundancy-check (CRC). The CRC uses the High-Performance CRC64 Library from Argonne National Laboratory. A CRC is computed for each variable, and each processor computes the CRC for the portion of the array residing on that process. Although the DAOS stack automatically performs a checksum from the ION to the disk, this is not the case for HDF5 files by default. However, the CRC can easily be implemented within the HDF5 file by simply computing a CRC for the array (assuming no partial writes are taking place) and writing the CRC as a dataset. The reading program can then read the dataset, compute the CRC for the read data and perform a comparison to the values stored in the CRC dataset. The implied restriction is that the layout of the array among the processors is the same for both the writing and the reading of the arrays. Ensuring a matching CRC for data written and data being read is important when creating raw binary files because the file can be transferred to a machine with a different endianness. Therefore, checks must be made to ensure the endianness conversions were implemented correctly. In HDF5 however, the library will convert and verify the byte-order automatically, so the use of the CRC may no longer be necessary. The rest of the implementation is straightforward, with only a slight modification of the original HDF APIs, Algorithm 1. As mentioned in Section 6, only HACC was not converted from using IOD/DAOS. Thus, Algorithm 1 highlights the complexity of handling the DAOS transactions using IOD/DAOS.

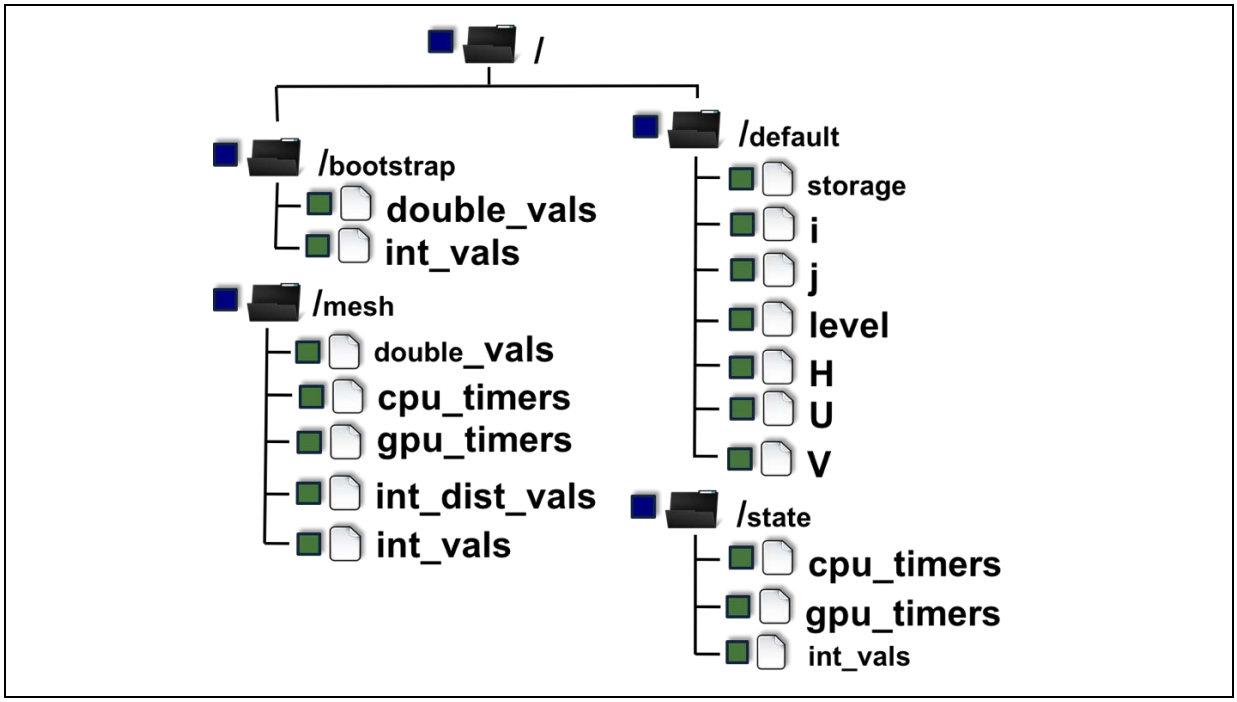
6.2 CLAMR Application

CLAMR (Cell-Based Adaptive Mesh Refinement) is a testbed application for hybrid algorithm development using MPI and OpenCL GPU code [3]. The use of one output file per process is rejected with CLAMR due to several issues:

- File systems are limited in their ability to manage hundreds of thousands of files,
- In practice, managing hundreds of thousands of files is cumbersome and error-prone,
- Reading the data back using a different number of processes than the analysis simulation requires redistribution and reshuffling of the data, negating the advantage over more sophisticated collective I/O strategies.

Thus, the I/O strategy for CLAMR is to create one output file per time step and to store each time step in an HDF5 file. Since HDF5 is a self-describing hierarchal file format, the "metadata" is automatically handled by HDF5. Thus, using HDF5 significantly reduces the internal bookkeeping and file construction required by CLAMR when compared to using POSIX or MPI-IO. The HDF5 implementation organizes the stored variables into datasets and uses groups to hold the datasets themselves (Figure 6-3).

Figure 6-3. The HDF5 file layout structure for CLAMR



Only a call to initialize DAOS was added to CLAMR's HDF5 implementation to utilize DAOS. This DAOS initialization requires a pool id which is passed to CLAMR by setting the environment variable `pid`. The pool id is obtained by starting the DAOS server::

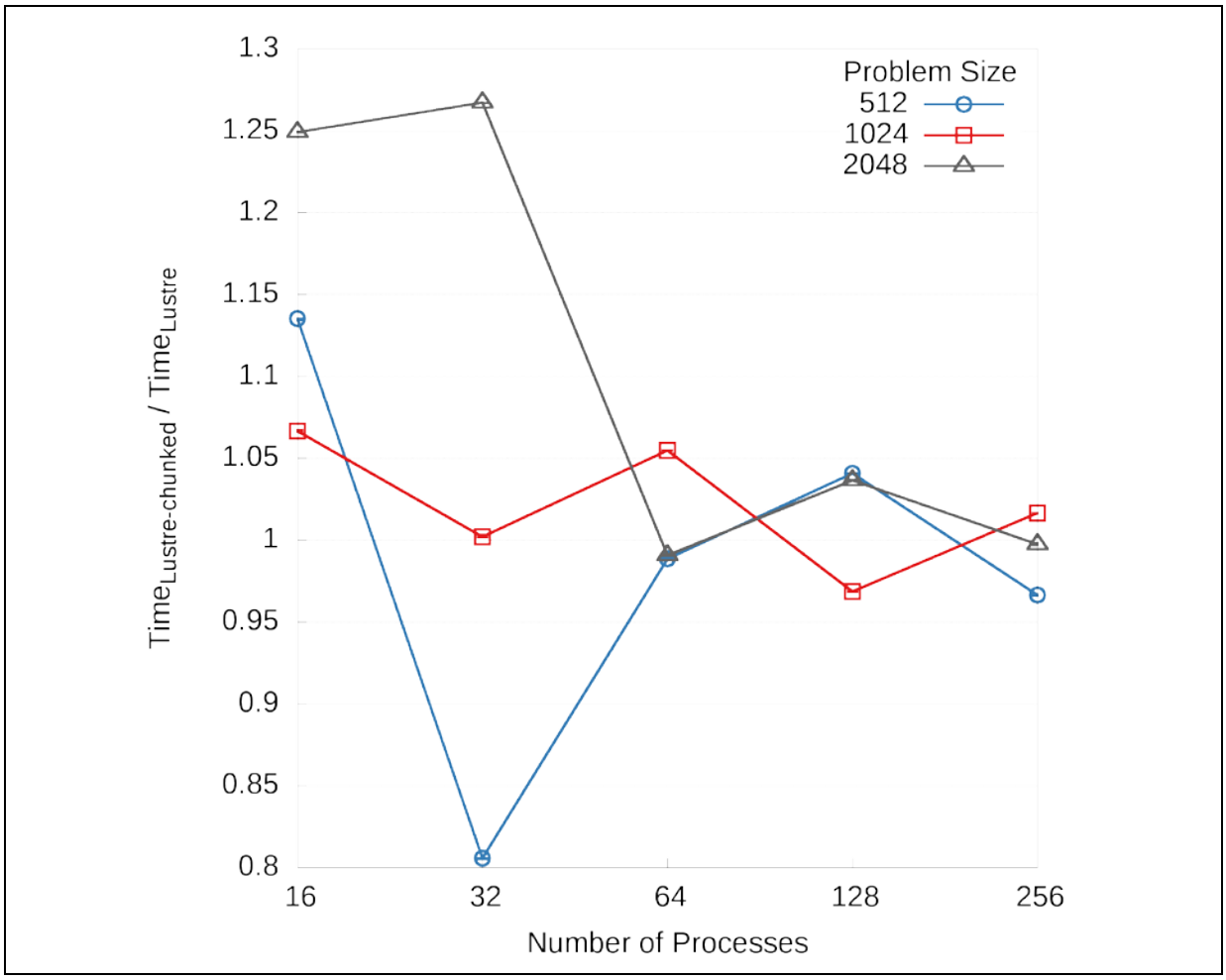
```
orterun -np 1 --report-uri ~/uri.txt daos_server -c 1
```

where c is the number of DAOS server threads, and then creating the pool and assigning the pool id to a shell environment variable:

```
export pid=$(orterun -np 1 --ompi-server file:~/uri.txt dmg create)
```

The first benchmark tested CLAMR for problem sizes ranging from 27 to 210 grid points by powers of 2, with the number of processors varying from 1 to 512 by powers of 2. For the initial run, a stripe count of four and a stripe size of 4 MB was used. The time to write the checkpoint files substantially increased as the number of processors increased.To gain a better understanding of what performance factors could be tuned, the Lustre parameters were varied for a fixed problem size of 211 grid points. The Lustre parameters started at a default stripe count of 1 with a stripe size of 1 MB, as well as the case of a stripe count of 4 and stripe size of 1 MB. The results were compared to the first case of a stripe count of 4 and stripe size of 4 MB. The study shows obtained results lead to the conclusion that the Lustre parameters had a slight effect on performance;, with the default stripe count of 1 and stripe size of 1 MB resulting in the best throughput (Figure 6 5Figure 6 5).Finally, different HDF5 chunking layout parameters using the default Lustre settings of a stripe count of 1 and stripe size of 1 MB were investigated. A variety of chunk sizes from 28 to 218 by powers of 2 were tested across different problem sizes to obtain the optimal chunk size for the given problem size. Using this optimal chunk size, the results were compared to the previously collected data for CLAMR's unchunked I/O performance (Figure 6-4).

Figure 6-4. Comparison of Lustre write performance of CLAMR for various problem sizes between unchunked I/O and chunked I/O using the respective optimal chunk size for the given problem size. Lustre parameters are a stripe count of one and stripe size of 1MB. Values above 1 represent chunked CLAMR I/O on Lustre performing worse than unchunked CLAMR I/O on Lustre; values below 1 represent chunked CLAMR I/O on Lustre performing better than unchunked I/O on Lustre

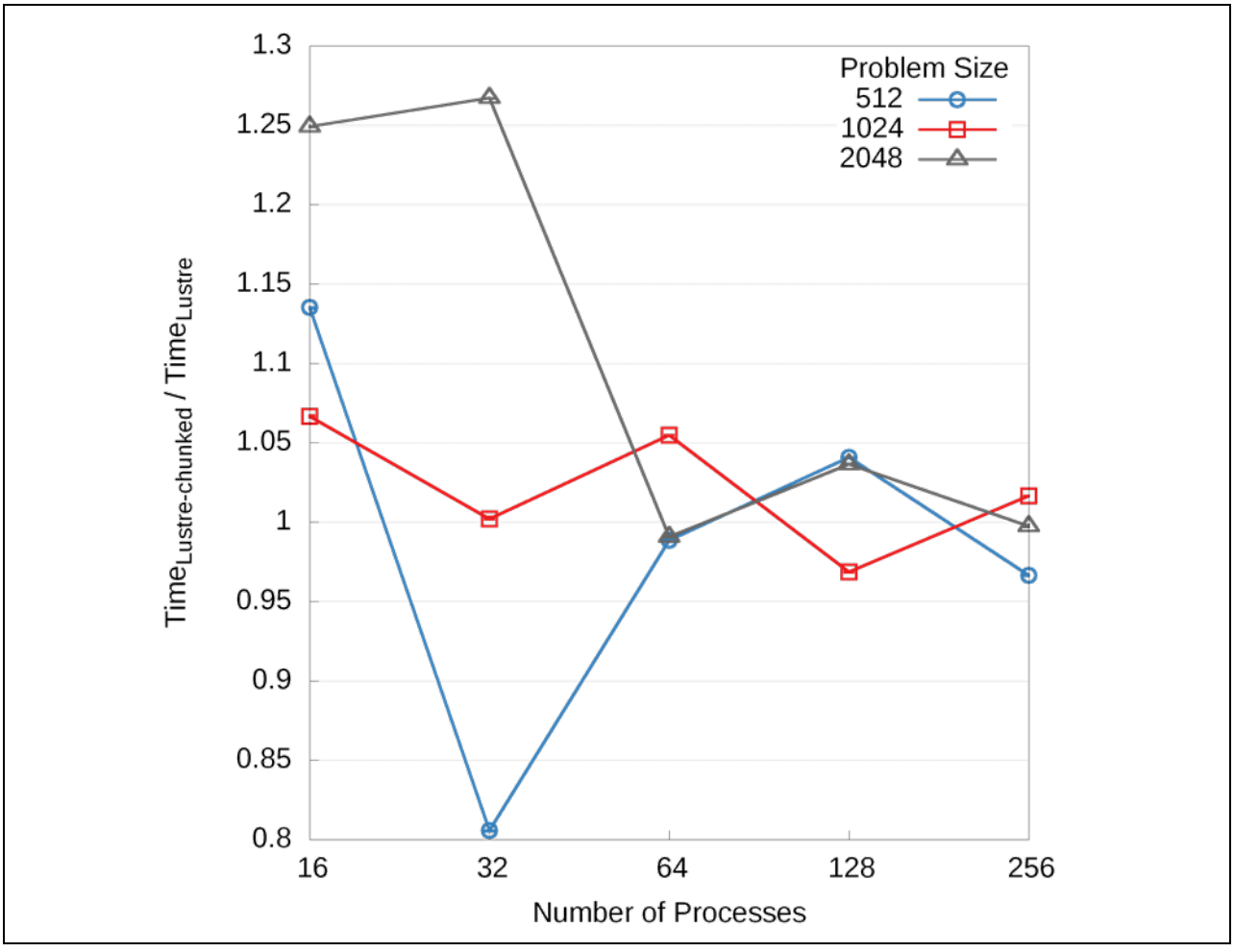


These results show that chunking had little positive effect on the performance of CLAMR checkpoint writing, with the result being marginally faster in a few cases and on par with or significantly worse than unchunked I/O performance in the other cases. When the chunk size of 8192 was used with 32 processors on the 512 problem size, a performance increase of 20% is observed, however no chunk size was able to replicate this performance improvement for the other sets of parameters.

CLAMR write performance was then assessed using the DAOS stack both with and without the use of chunking. Using the previous data gathered for CLAMR's performance in writing checkpoint files to a Lustre filesystem, a comparison was drawn, showing that DAOS without chunking performs twice as well on average as compared to Lustre (up to nearly three times as well in specific cases) when the number of processors is sufficiently large (Figure 6-5). In order

to assess the performance impact that chunking has, 8 DAOS servers were started across 8 different nodes and a variety of chunk sizes ranging from 29 to 221 were tested. When chunking is enabled, DAOS performance can exceed the unchunked performance by approximately 25-50%, depending on the number of processors used and the appropriateness of the chunk size chosen for the given problem size (Figure 6-6). When the number of processors used is large enough, this in turn leads to a consistent 50-75% increase in performance as compared to Lustre unchunked I/O (Figure 6-7).

Figure 6-5. Comparison of unchunked write performance of CLAMR on DAOS/Lustre. Lustre parameters are a stripe count of one and stripe size of 1MB and DAOS parameters are 1 DAOS server. Values above 1 represent unchunked CLAMR I/O on DAOS performing worse than unchunked CLAMR I/O on Lustre; values below 1 represent unchunked CLAMR I/O on DAOS performing better than unchunked CLAMR I/O on Lustre.



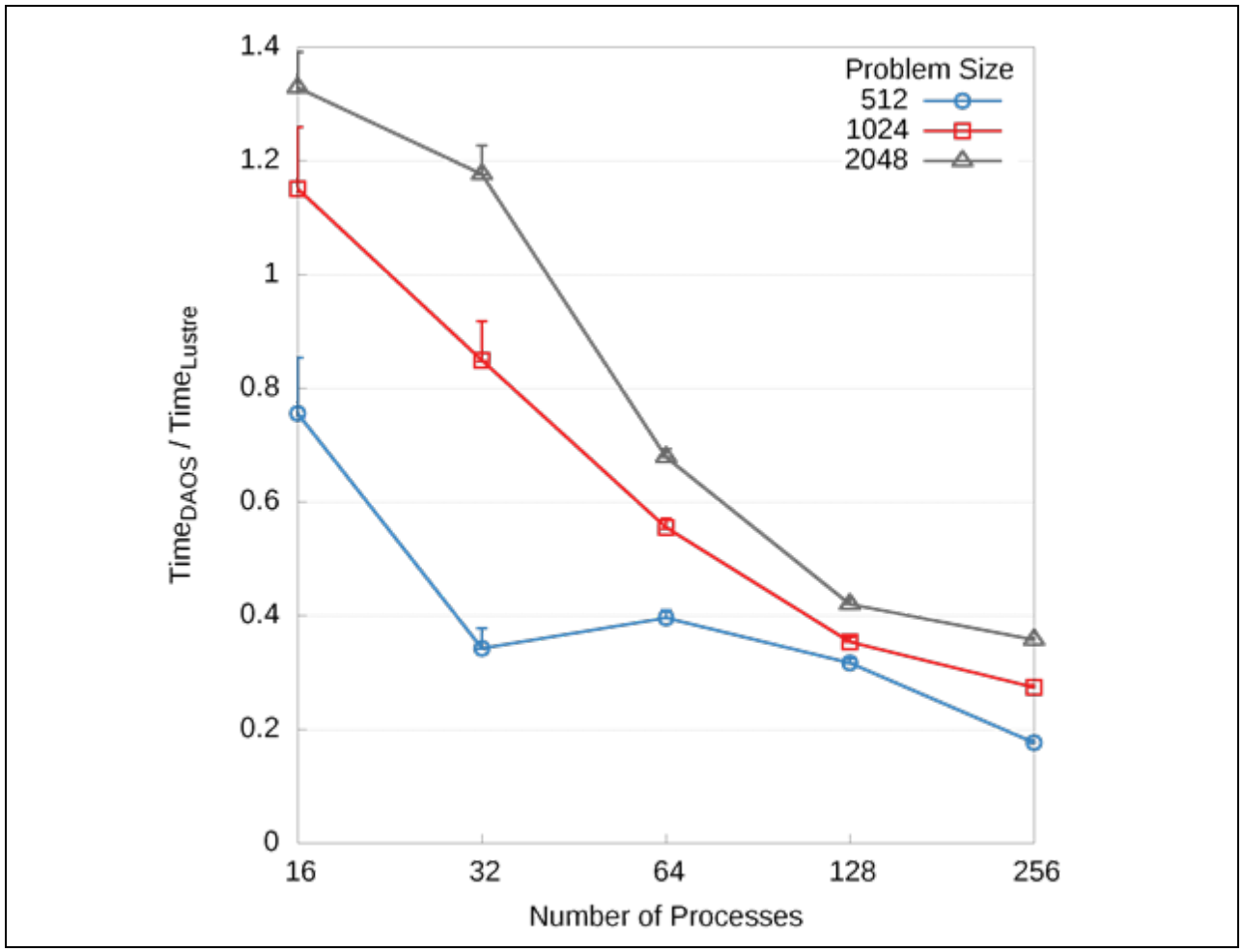

Problem Size
512
1024
2048
1.4
1.2
1
0.8
0.6
0.4
0.2
0
16
32
64
128
256
Number of Processes
Time_DAOS / Time_Lustre

Figure 6-6. Comparison of chunked and unchunked CLAMR write performance on DAOS. Unchunked CLAMR I/O on DAOS used 1 DAOS server; Chunked CLAMR .
I/O on DAOS used 8 DAOS servers across 8 different nodes. Values above 1 represent chunked CLAMR I/O on DAOS performing worse than unchunked CLAMR I/O on DAOS; values below 1 represent chunked CLAMR I/O on DAOS performing better than unchunked CLAMR I/O on DAOS

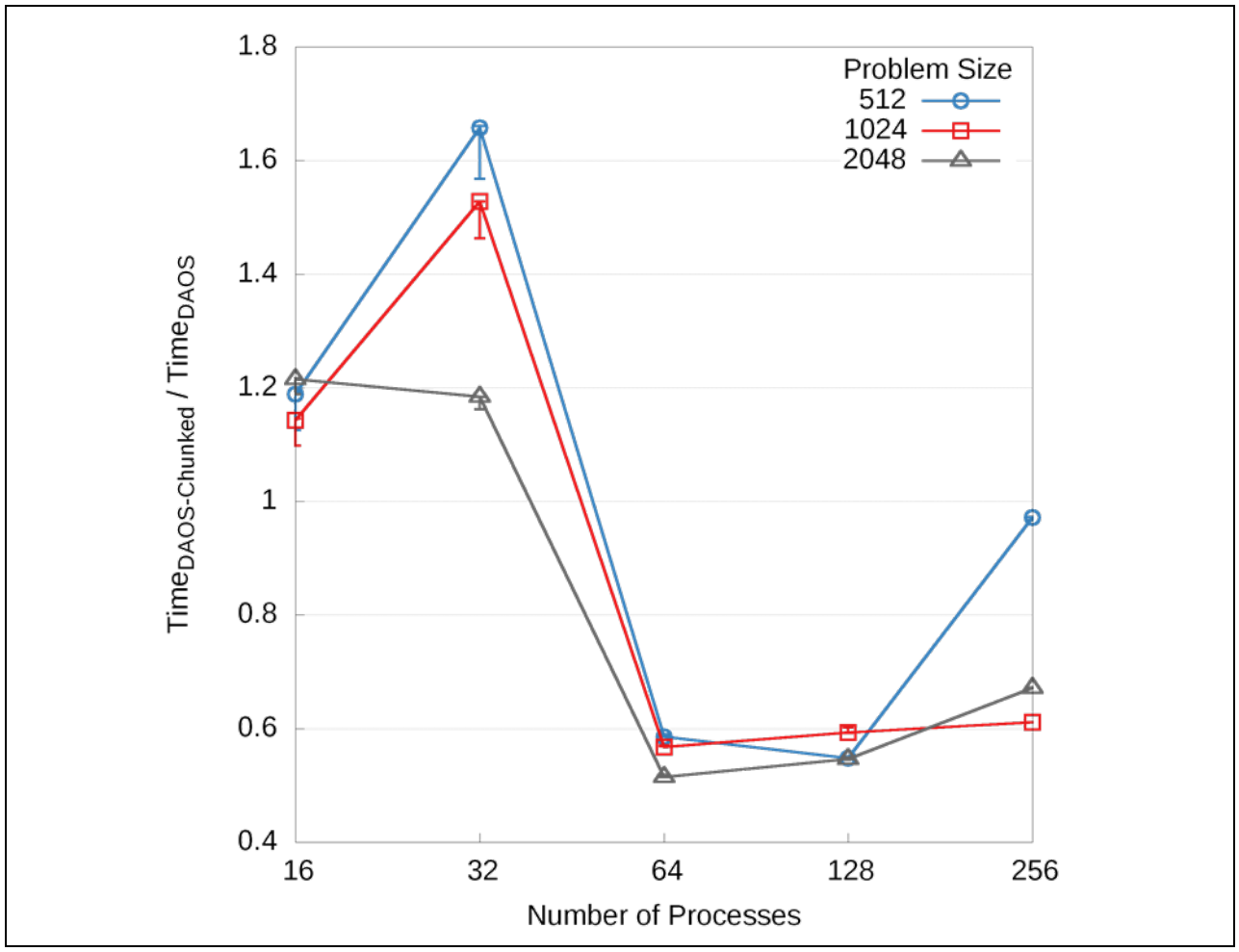
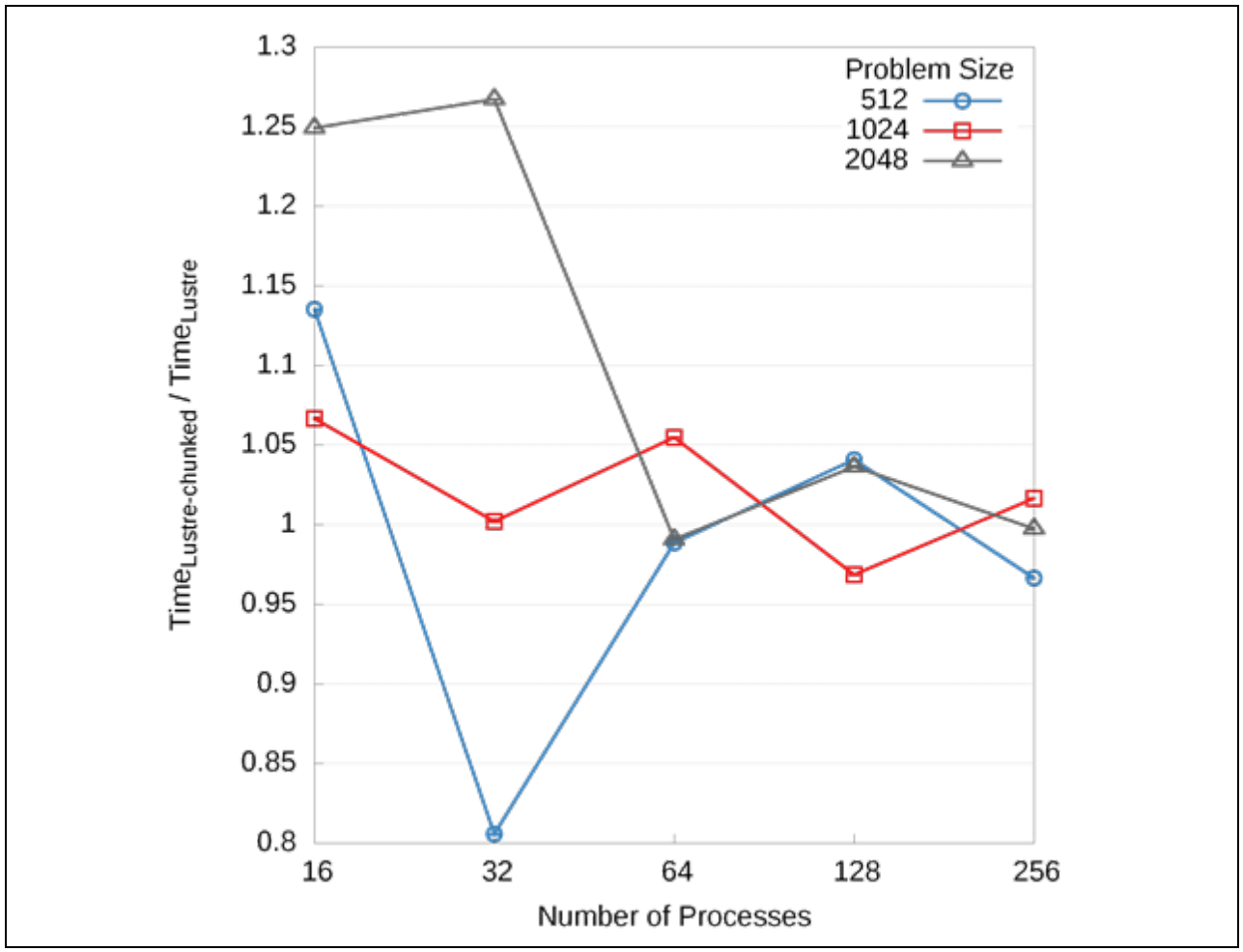
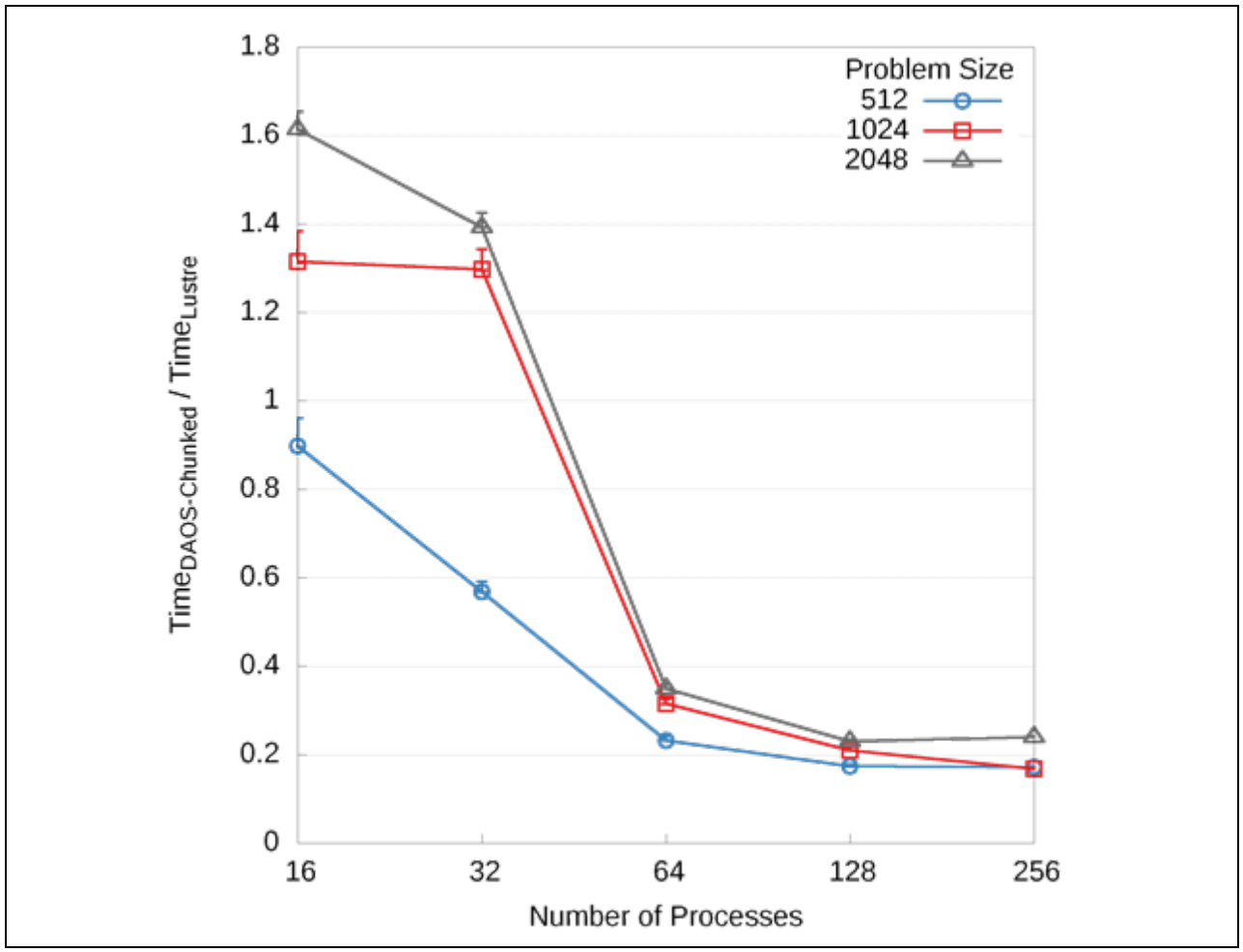


Figure 6-7. Comparison of chunked CLAMR I/O on DAOS to unchunked CLAMR I/O on Lustre. Lustre parameters are a stripe count of one and stripe size of 1MB and DAOS parameters are 8 DAOS servers across 8 different nodes. Values above 1 represent chunked CLAMR I/O on DAOS performing worse than unchunked CLAMR I/O on Lustre; values below 1 represent chunked CLAMR I/O on DAOS performing better than unchunked CLAMR I/O on Lustre.





6.3 Legion

Legion [4] is a high-level data-centric and task-based application runtime. Legion's primary data model is based on Logical Regions, which are the cross product of an N-dimensional index space and a multi-variable field space. Logical regions are distinct from the physical regions (memories) that underlie the logical region and provide the physical instantiation of the data. The data model also provides a method of coloring the index space and/or the field space, which can be used to partition an index space or slice the field space of the logical region. The runtime can then use this coloring and a partitioning applied to the logical region to manage a distributed instance of the logical region.

Legion is currently capable of attaching to HDF5 files using the low-level Realm runtime on which Legion is built. Realm maps HDF5 files (or groups) to memory objects within the runtime using a DMA mechanism to collect updates to individual datasets in an HDF5 file. As part of demonstrating Legion's use of HDF5 on the DAOS storage stack, an example application is created that demonstrates the capabilities of the storage stack. The application, tester_io, is

part of Legion's Github repository [5] and can be found in the source tree at test/hdf_attach_subregion_parallel. Tester_io is a straightforward tour of the HDF5-DAOS features from a simple Legion code and is designed to run the Legion runtime on multiple compute nodes, talking to multiple I/O and storage server nodes.

Legion leverages the DAOS epoch model by making two assumptions:

- The Legion scheduler always serializes the execution of conflicting tasks (i.e. overlapping updates). In other words, record overwrite can happen from two tasks running at the same time. For atomic value, the record associated with a given key is not updated concurrently. As for byte array value, no overlapping extents are submitted concurrently.
- The commit operation in DAOS has a very low latency and is several orders of magnitude shorter than the latency of the flush operation.

When the legion framework starts, the workflow scheduler connects to the pool on behalf of Legion and passes the pool handle as an environment variable to the Legion runtime.

Legion can open the container from one or multiple runtime processes. This operation grants a private container handle to the runtime process and returns the current value of highest partially committed epoch (HPCE). To modify the container, the legion runtime process must obtain an epoch hold. This operation returns a lowest held epoch which is guaranteed to be higher than the current HPCE (typically HPCE+1). The legion runtime process can then read from the lowest held epoch and submit new updates against this epoch or the next ones. To complete any write operations, the runtime flushes and commits its updates and then close its container handle. This guarantees that any subsequent operations will have full visibility of previously committed operations even if the operations are scheduled on a different process than the process which previously committed operations to the container. Indeed, any new container handle should return an HPCE greater than or equal to the HCE of the already completed tasks.

In the current implementation of HDF5 on top of DAOS, transactions and the event stack are handled internally in the HDF5 library, and consequently, the original HDF5 APIs could be used as no extra parameters are needed. Therefore, the number of changes from the original Legion and HDF5 implementation was minimal. Two new HDF5 APIs were introduced into Legion:

- *H5VLdaosm_init* – Initialize the VOL plugin by connecting to the pool and registering the driver with the HDF5 library.
- *H5VLdaosm_term* – Shut down the DAOS VOL.

These functions must be called only once by all the processes within Legion.

Furthermore, Virtual Dataset (VDS) support is a relativity new feature added to HDF5. A VDS does not store any data directly, rather, it stores data in a set of source datasets. Source datasets are created like any other source dataset, and the virtual dataset is created with a set of mappings between regions in the virtual dataset and regions in the source dataset. I/O requests to the VDS are then translated into a set of I/O requests to the source datasets whose mappings overlap with the requested region in the VDS. Virtual datasets mapped

closely to Legion's data model and were added to Legion to replace the use of H5Lcreate_external, which was used to create external links in a master HDF5 file, linking to the shard files. However, H5Lcreate_external is not, and will not, be supported in the current HDF5 DAOS implementation. Since the current HDF5 DAOS implementation does not include the VDS APIs, this functionality of creating a master HDF5 file linking the shard files via VDS was removed in the current Legion implementation. The functionality can easily be restored once the HDF5 VDS APIs are ported to DAOS. The VDSfeature was implemented for this project in the original IOD/DAOS HDF5 version and legion used this feature successfully.

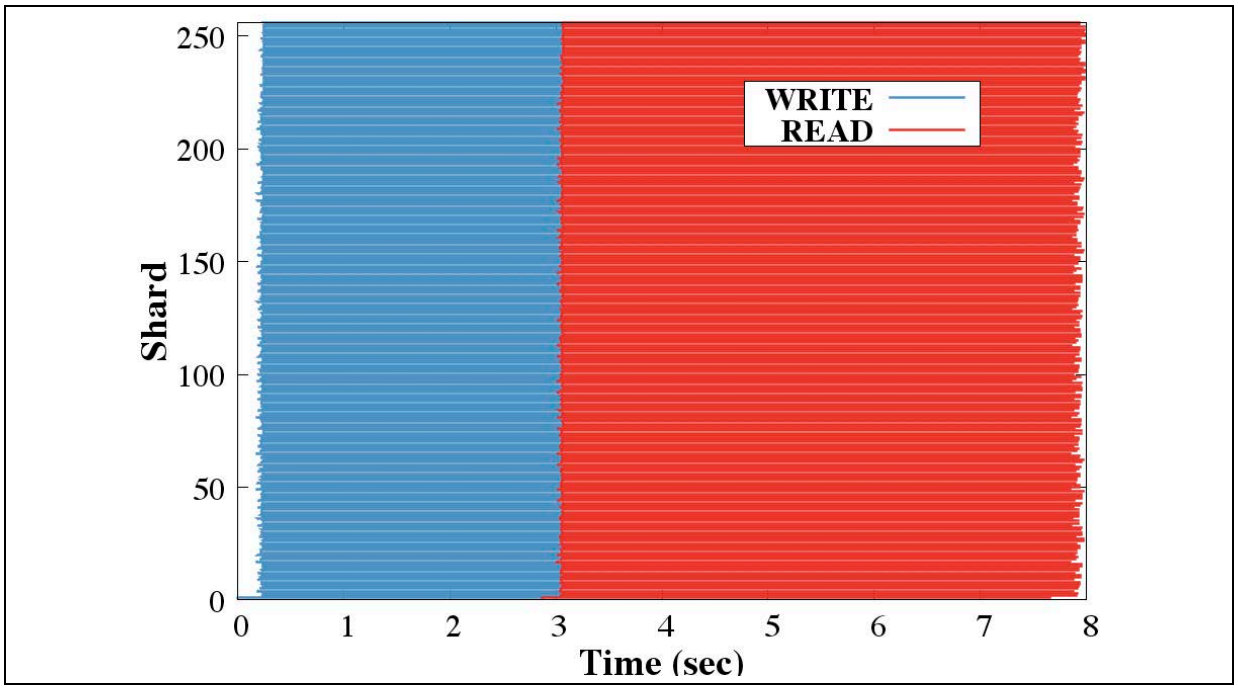
The Legion benchmark relies on the low-level networking layer gasnet [6] for network-independent, high-performance communication primitives. The mpiexec command in the gasnet wrapper script gasnetrun_ibv needed to be updated to include DAOS specific parameters. For example, the command for running tester_io on 3 nodes with 131072 elements and 256 shards is,

```
GASNET_BACKTRACE=1 GASNET_USE_XRC=0 GASNET_MASTERIP='...' GASNET_SPAWN=-L
gasnetrun_ibv -n 3 tester_io -n 131072 -s 256
```

which on Boro (Intel's cluster) will run on three nodes. As was the case with CLAMR, the pool id is passed to Legion through the environment variable pid.

The non-bulk synchronous ability of DAOS via the transaction and epoch model will be demonstrated by *Tester_io*. Figure 6-8 shows the typical workload for the read/write phase for different shards in *Tester_io*. Additionally, Figure 6-8 highlights Legion's capability of scheduling different phases of tasks based on explicit dependencies and, consequently, allows for reading tasks to run concurrently with writing tasks on the same logical region [7].

Figure 6-8. Time-series of the I/O phases (write and read) associated with each shard's global data structure being persisted for HDF5 with DAOS, highlighting Legion's ability to perform independent I/O phases associated with each shard of global data for DAOS



Tester_io was used to study the effects of the number of MPI processes as a function of the number of Legion subregions, where the size of the problem was fixed for the number of elements of 2^{29}. The number of MPI processes was 4, 8 and 15, where each compute node executed only one MPI process, and each node could run 74 tasks. The higher number of process made a large difference as the number of sub regions increased greater than 256 for both reading and writing (Figure 6-9 and Figure 6-10).

The performance of DAOS and Lustre is presented in Figure 6-11 for 15 MPI processes and where the number of elements is 2^{30}. The slowdown in DAOS and the big performance hit when compared with Lustre in this case was expected, since the Legion tester code generates a large number of I/O calls, and the HDF5 implementation in Legion did not use chunking layout for the Datasets. This results in all the I/O calls being serialized at the DAOS server to one service thread, whereas the Lustre server utilizes many threads to handle the incoming I/Os. Furthermore, as mentioned earlier, the network interface used by DAOS on this cluster was libfabric over TCP which is slower compared to InfiniBand verbs utilized by Lustre. We also noticed that the *tmpfs* file system was returning out of space errors for larger problem sizes, even when space was available, after a certain number of memory allocations triggered by the large number of I/O operations from the client. All those issues will be addressed in future development of both DAOS and the HDF5 DAOS backend.

Figure 6-9. DAOS read performances as the number of mpi proceses is increased from 4 to 15 processes for a different number of subregions.

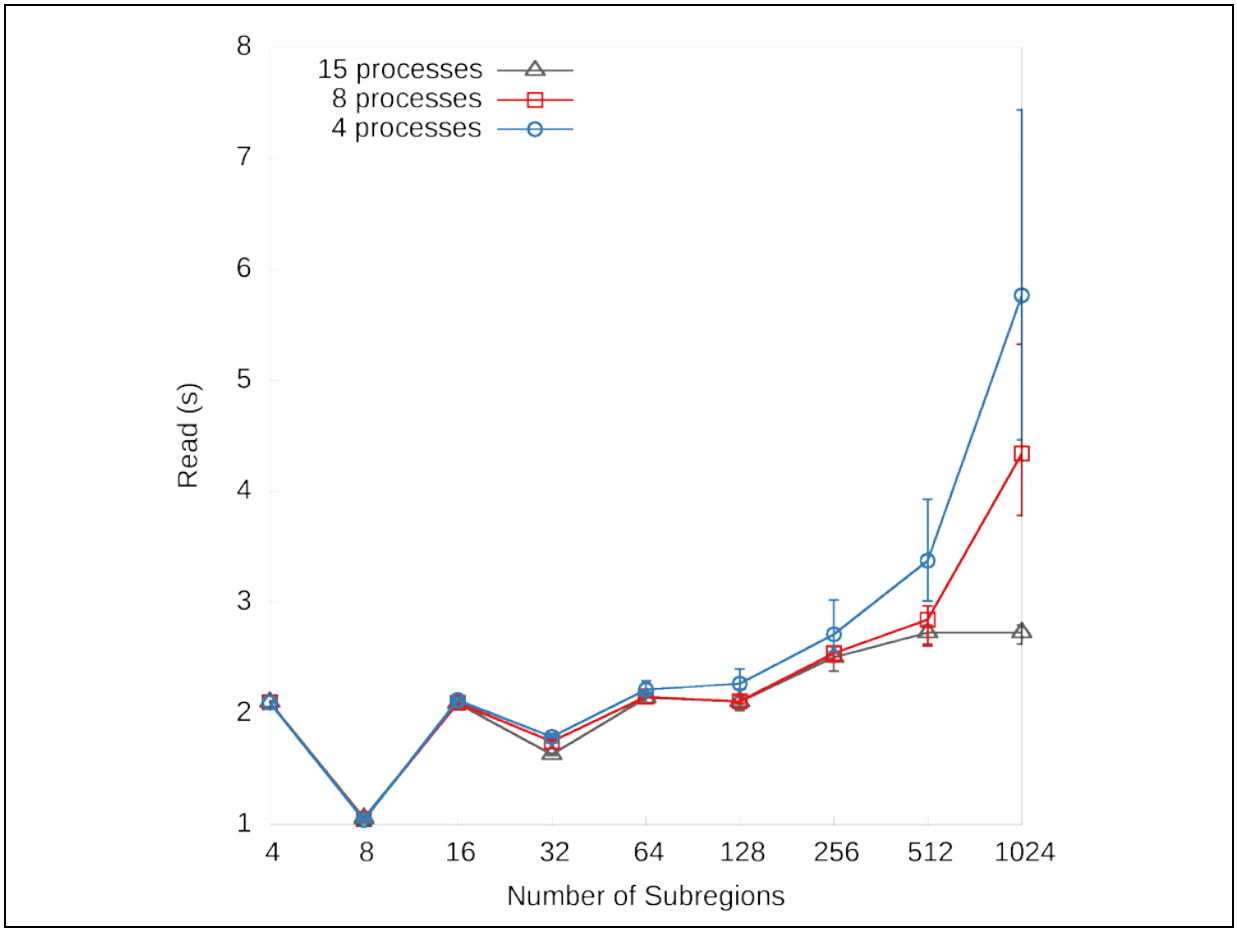


Figure 6-10. DAOS write performances as the number of mpi proceses is increased from 4 to 15 processes for a different number of subregions.

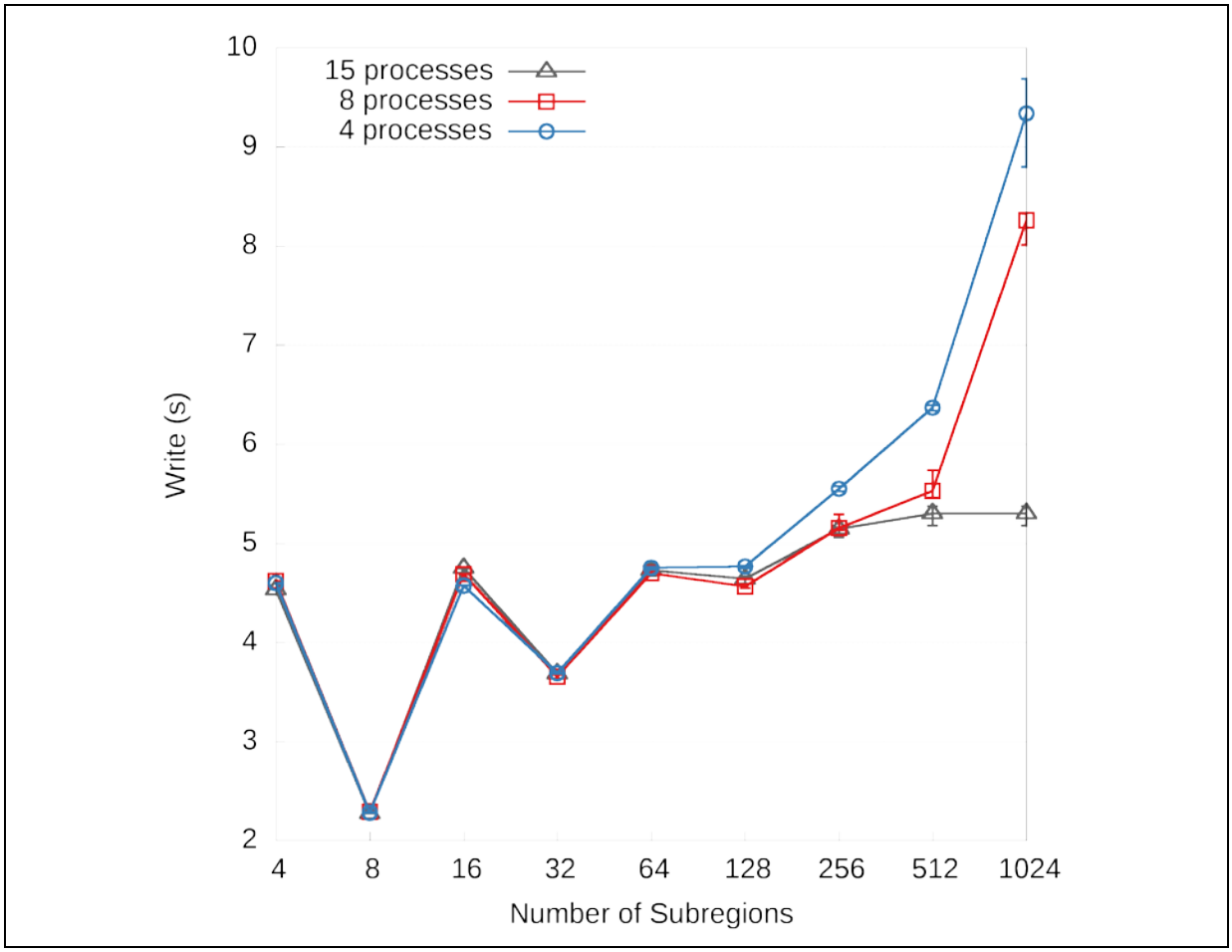
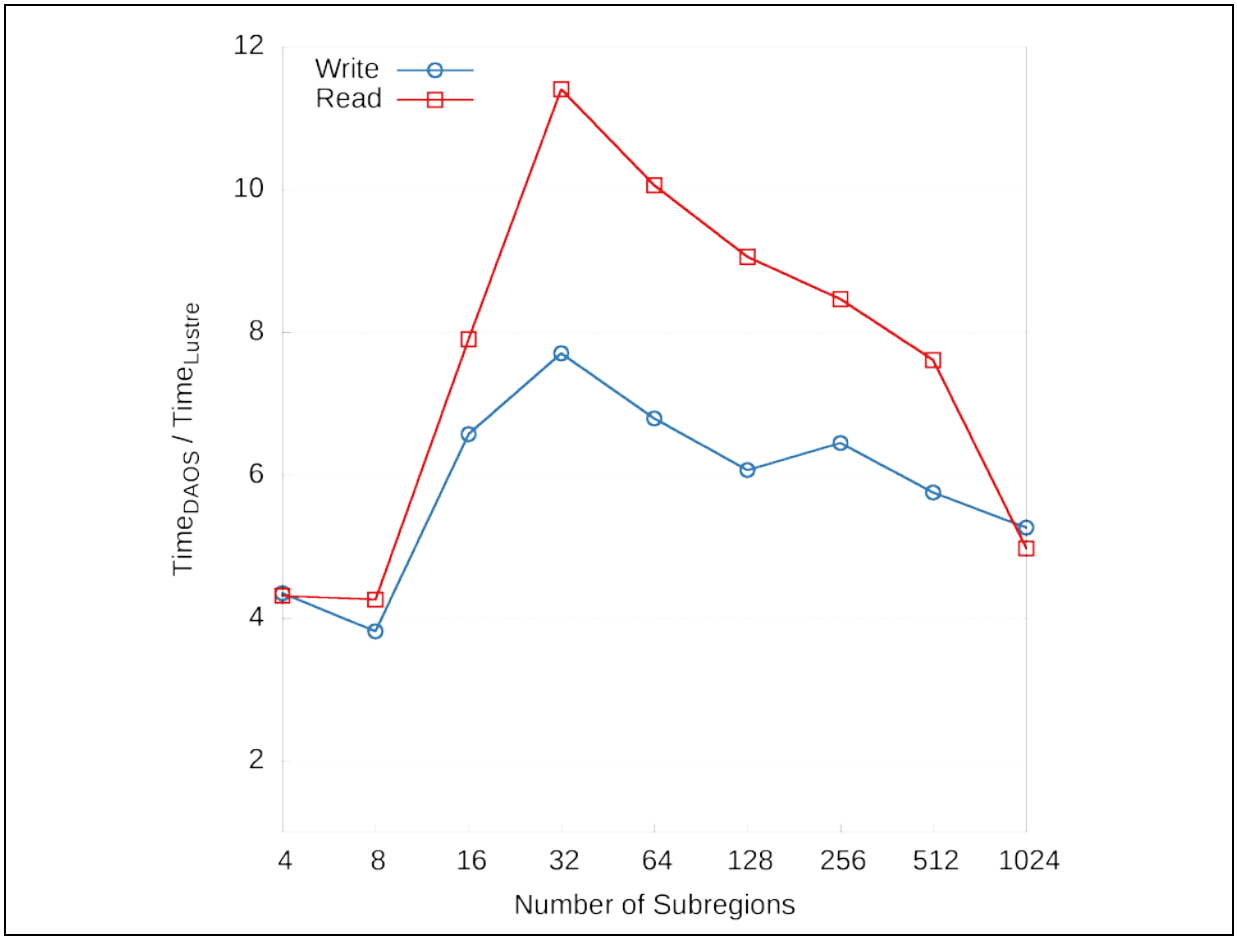


Figure 6-11. DAOS read and write performance in comparison to Lustre.



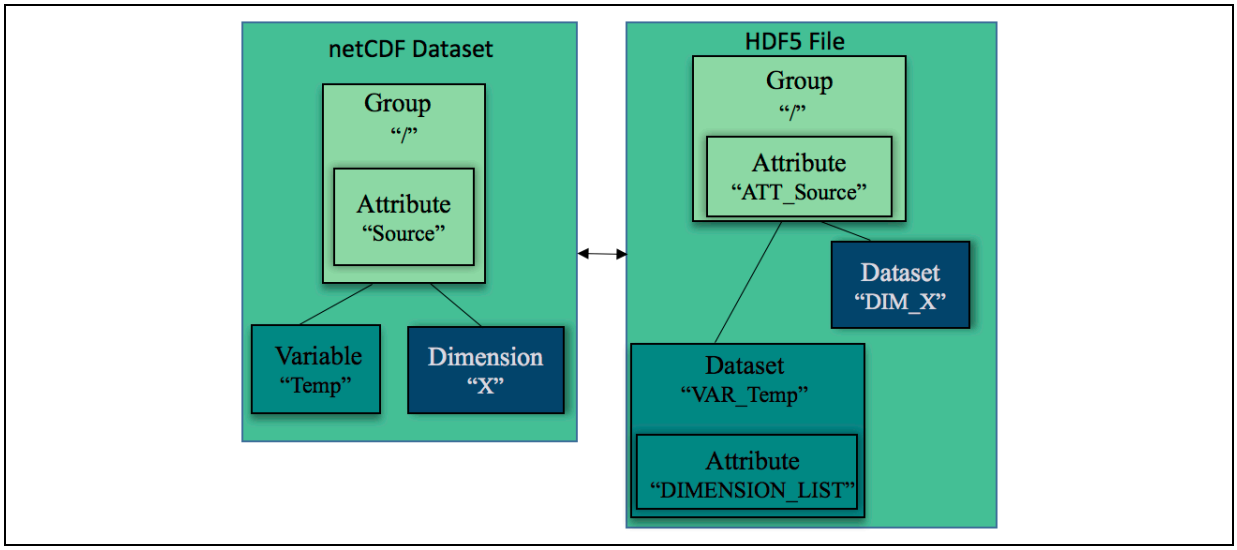
6.4 DAOS netCDF implementation

NetCDF [10] is a set of software libraries used to facilitate the creation, access, and sharing of array-oriented scientific data in self-describing, machine-independent data formats. A new set of DAOS NetCDF APIs were derived from the original NetCDF APIs, maintaining most of the original functionality.

The non-DAOS version of NetCDF added dimensions to variables by the use of HDF5 *Dimension Scale* APIs. Storing dimensions with coordinate variables (variables used as a dimension scale for a dimension of the same name) is intuitive and is self-describing for applications that access the file directly through HDF5. However, this approach introduces several dependencies between datasets and attributes that do not fit well with the DAOS transaction model. In addition, it introduces many special cases that must be handled, increasing the difficulty of implementation. Finally, there is currently no DAOS implementation

of the Dimension Scale APIs and implementing them would be difficult due to the transaction model.

Figure 6-12. HDF5 schema for NetCDF/DAOS



Since backward compatibility with NetCDF's file format was not of a concern (i.e., having NetCDF/DAOS datasets/containers/files being accessed independently of the NetCDF/DAOS API) the existing NetCDF4 schema for HDF5 was abandoned in order to simplify the implementation. All variables and dimensions were implemented as HDF5 datasets, all groups as HDF5 groups, and all attributes as HDF5 attributes. As a convention, all dimensions have the string DIM_ prepended to the name in HDF5, all variables have the string VAR_ prepended, and all attributes have the string ATT_ prepended. Variables have an HDF5 attribute DIMENSION_LIST, invisible to the NetCDF API, that stores references to the dimensions for the variable, Figure 6-12. Dimensions are implemented as a scalar dataset of type H5T_STD_U64LE, where the value indicates the dimension length or all 1s (i.e., *(uint64_t)(int64_t)-1*) to indicate an unlimited dimension. This implementation avoids all name conflicts without having to add any special cases to the code, and also allows the removal of code paths for handling coordinate variables as a special case, instead of treating them like any other variable. All the NetCDF APIs that use this new schema were appended with a "*_ff*" in their names. Other DAOS additions to NetCDF included:

- Support for unlimited dimensions, but only for collective access and only for the slowest changing dimension;
- "Links" from variables to their dimensions, allowing the variables to be queried about their dimensions.

The current DAOS implementation of HDF5, in comparison to the inaugural IOD/DAOS version, removes access to the DAOS variables from the user (i.e. the read context identifier, the event stack identifier, the transaction identifier and version number) by handled them

internally within the HDF5 library. Therefore, there is no longer a need for separate DAOS HDF5 APIs, which were distinguished by a _ff prefix in the original implementation of HDF5 for IOD/DAOS. This greatly simplifies porting an application that is already using the standard HDF5 library to work with DAOS. The DAOS HDF5 APIs used by NetCDF are as follows:

- H5Acreate
- H5Awrite
- H5Aopen
- H5Aclose
- H5Aiterate
- H5Aget_name
- H5Aget_space
- H5Aget_type
- H5Dcreate
- H5Dopen
- H5Dwrite
- H5Dread
- H5Fopen
- H5Fcreate
- H5Gopen
- H5Gcreate
- H5Gopen
- H5Gclose
- H5Literate
- H5Oget_info
- H5Oclose
- H5Topen

The NetCDF version built using the DAOS HDF5 schema mention above, and can be found at:

https://hdfgit.hdfgroup.org/scm/ffwd2/NetCDF-c.git.

6.5 ACME/Parallel IO (PIO) Overview

The end goal of implementing a DAOS version of NetCDF is to demonstrate an application which uses NetCDF. The NetCDF application identified for this project is software associated with the Accelerated Climate Modeling for Energy (ACME) program. Their software uses the package Parallel I/O (PIO) to perform I/O which, in turn, uses as its backend the NetCDF file format. PIO uses a large subset (211) of the NetCDF functions (not all of which need to be DAOS compatible).

The PIO performance testing program pioperformance.F90 is an ACME I/O stand-alone driver program that closely duplicates the I/O pattern from an actual ACME application. Therefore, this project implemented the program pioperformance.F90 within the DAOS framework via a DAOS version of PIO. Pioperformance.F90 uses the following PIO functions:

- PIO_init
- PIO_Readdof
- **PIO_CreateFile**
- PIO_InitDecomp
- PIO_setframe
- **PIO_write_darray**
- **PIO_read_darray**
- PIO_freedecomp
- **PIO_OpenFile**
- **PIO_CloseFile**
- PIO_def_var
- PIO_def_dim
- PIO_def_att
- **PIO_enddef**

The PIO APIs in **blue** utilize DAOS compatible NetCDF APIs. Since no NetCDF APIs were changed as result of porting NetCDF to DAOS, the actual code modifications in PIO were minimal. For the most part, the changes simply included the addition of a call to initiating DAOS (H5VLdaosm_init) . The transaction number is automatically initialized and incremented as needed within HDF5. Similar to the NetCDF convention, all new DAOS PIO C APIs are indicated by appending a "_ff" to the function names.

The DAOS PIO source files and test code can be downloaded from:

https://hdfgit.hdfgroup.org/scm/ffwd2/parallelio.git

PIO expects as input from the application the partitioned data arrays for each process. Additionally, PIO has the option for requesting a subset of the CN that will perform the IO. Hence, PIO aggregates the IO from each process to only a subset of processes for IO. The IO processes then use NetCDF APIs to carry out the IO. PIO implements two methods for aggregating the IO from all the processes to the subset of IO processes. In the box method, each compute task will transfer data to one or more of the IO processes. For the subset method, each IO process is associated with a unique subset of computing processes for which each compute process transfers data to only one IO process [8]. In general, the subset method reduces the overall communication cost when compared to the box method. All the demonstrations use the box method.

Additionally, since PIO has the capability of using a subset of processes for I/O, *H5VLdaosm_init* (i.e., a DAOS HDF5 API used to start the DAOS stack) uses the MPI sub-communicator group so that only those processes involved in I/O will initialize the DAOS stack. This initialization of the DAOS stack happens automatically when the I/O MPI sub-communicator is created in PIO and it is finalized when this same sub-communicator is freed in PIO.

PIO's testing program \texttt{pioperformance.F90} uses two input files; the first file contains namelist settings for the testing parameters and the second file contains the decomposition information from a PIO program (e.g. CESM, ACME). The test program reads namelist and then generates test data consisting of integers, 4-byte reals and 8-byte reals. It then writes the data using DAOS NetCDF via PIO APIs and then reads the data back using DAOS PIO, checks for correctness and outputs the data rate in reading and writing the data.

Decomposition files are available at:

https://svn-ccsm-piodecomps.cgd.ucar.edu/trunk.

The format of the decomposition file name is:

piodecomp<***NUM_MPI_PROCESSES***>tasks<***NUM_DIMENSIONS***>dims<***COUNTER***>.dat

where:

NUM_MPI_TASKS is the number of MPI tasks/ranks (30, 1024, 2048 and 16384 are available), *NUM_DIMENSIONS* is the number of dimensions in the decomposition (typically

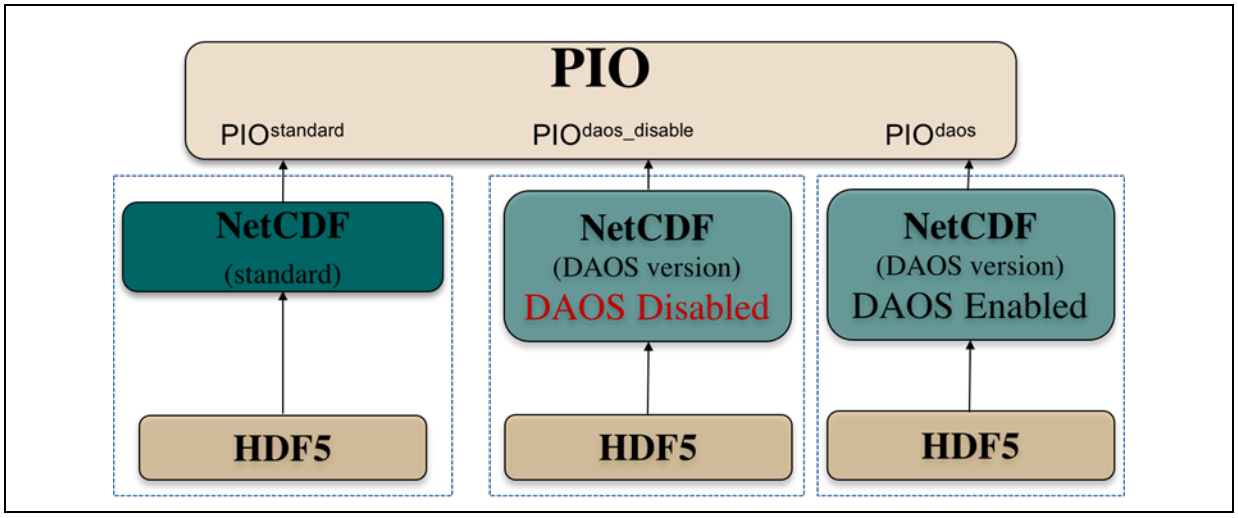
corresponding to the variable for which the decomposition was created), and *COUNTER* is a file identifier counter. [9] The test program reads namelist and then generates test data consisting of integers, 4-byte reals and 8-byte reals. It then writes the data using DAOS NetCDF via PIO API and then reads the data back using DAOS PIO, checks for correctness, and outputs the data rate in reading and writing the data.

As mentioned earlier, this research uses two versions of NetCDF: (1) the "standard" version is the unmodified v4.4.1 of NetCDF available from Unidata and (2) the "DAOS" version, which is a modified v4.4.1 NetCDF, as previously discussed. There are three combinations of PIO, NetCDF and HDF5 that were investigated (Figure 6-13).

1. $PIO^{standard}$ – The PIO configuration uses standard HDF5 with the standard version of NetCDF.
2. $PIO^{daos\_disable}$ – The PIO configuration uses standard HDF5 with the DAOS version of NetCDF, where the DAOS capabilities in NetCDF are disabled.
3. PIO^{daos} – The PIO configuration uses the DAOS version of HDF5 with the DAOS capabilities in NetCDF enabled.

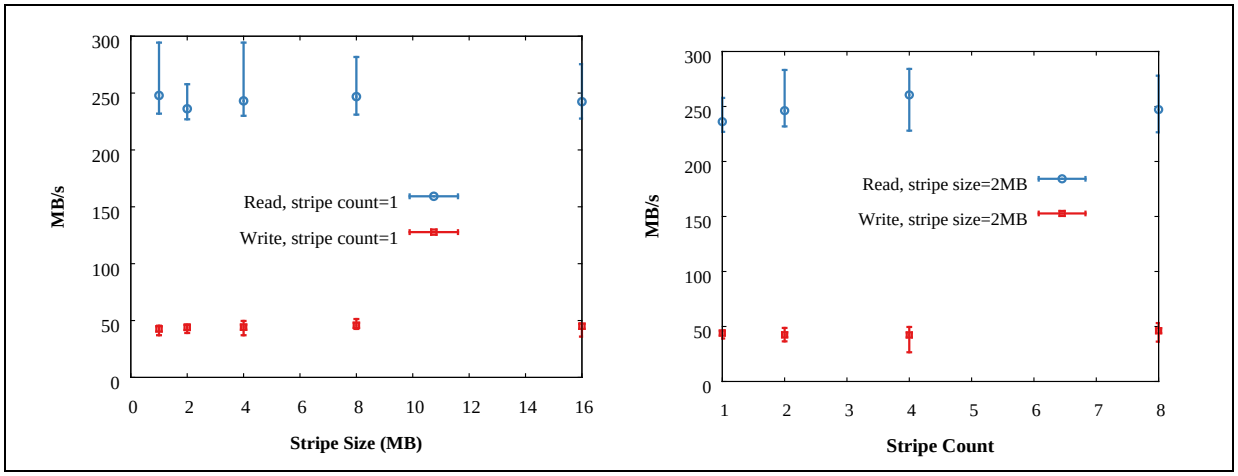
Thus, for both case 1 and 2, a POSIX file is getting written, via HDF5, to a Lustre backend.

Figure 6-13. Three combinations of PIO, NetCDF, and HDF5



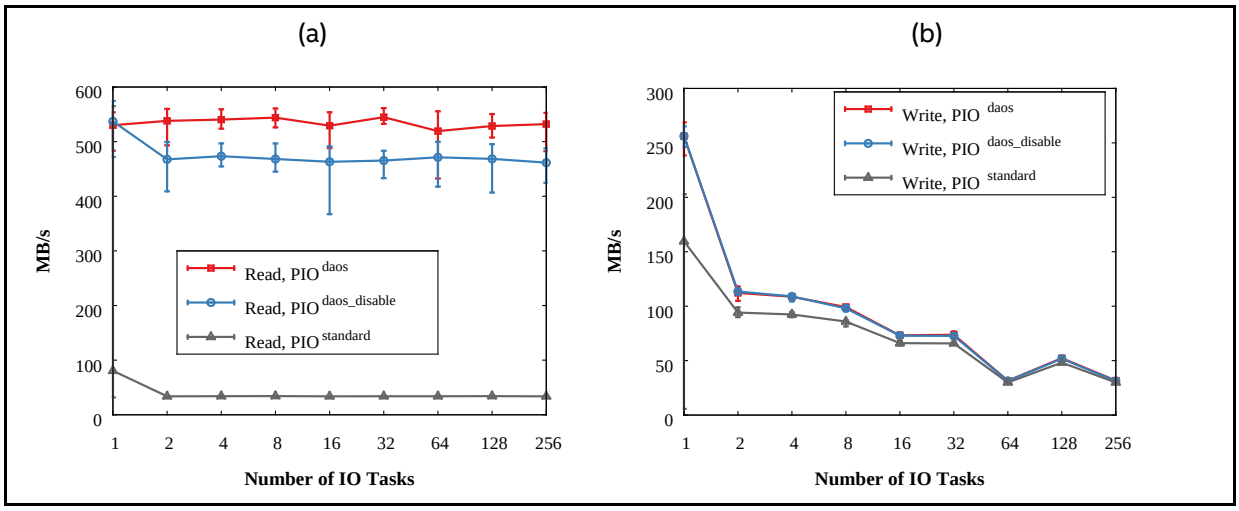
All tests were made on Intel's Boro cluster to compare PIO with Lustre and DAOS, and to verify the DAOS PIO implementation. The Lustre parameters were investigated for the 30 processes case, Figure 6-14, and accordingly, a stripe count of 1 with a stripe size of 2MB was chosen for all the Lustre benchmark results. The benchmarks were run ten times, and the average IO over all ten runs is plotted as a symbol in the subsequent benchmarking figures. The vertical line segment represents the minimum and maximum of IO over the ten runs.

Figure 6-14. Lustre parameter Stripe Size (a) and Stripe Count (b) effects on reading and write performance



The benchmark used 1024 processes and the decomposition file, piodecomp1024tasks03dims05.dat. The modified DAOS version of netCDF outperforms the standard version for both reading and writing. Furthermore, the read performance is nearly twice as fast as the write performance for both cases of the DAOS version of NetCDF. The DAOS version also outperforms Lustre for reading and is nearly identical to Lustre for writing. Furthermore, the DAOS implementation is on average faster than the Lustre implementation for reading, Figure 6-15(a), and matches the Lustre performance for writing, Figure 6-15(b).

Figure 6-15. Read (a) and write (b) performance of 1024 processes as a function of IO tasks



7 Methodology

The ESSIO program is funded by the DOE to create a storage stack that targets the needs of future HPC systems. Program development ran from July 2015 through June 2017 for a total of eight quarters. The output for the program was to create a working DAOS prototype and an HDF5 VOL plugin that uses DAOS as a backend storage for the HDF5 data model. In addition several applications were investigated and ported to use HDF5 on the ESSIO storage stack.

7.1 Processes

With a development team distributed internationally, at least once a week, development meetings were held to discuss:

- Milestone scheduling
- Technical development and architecture of features and integration
- Preparation of software demonstrations
- Test cluster setup and usage
- Document preparation

Whenever possible, team representatives would meet in person about once per quarter for more in-depth discussion about the stack architecture, design, and future planning. Skype was used mainly as an online communication tool for short technical discussions and collaboration.

Quarterly milestones were demonstrated to stakeholders. A weekly telecon was held between program management and the contract and technical customer representatives. Signoff of quarterly milestones was requested after the conclusion of software demonstrations and delivery of accompanying documentation.

The quarterly demonstrations were preceded by a high level review two quarters in advance and a detailed level review one quarter in advance. This allowed engineers and stakeholders to ensure a common understanding of what would be delivered. This cadence was a successful risk mitigation approach for the majority of milestones.

7.2 Test Resources

In addition to each developers private VMs, Intel's Boro cluster was made available to the ESSIO team. The cluster composed of 81 nodes each with 2 Xeon DP Haswell-EP E5-2699 v3 CPUs. DAOS leveraged a tmpfs file system over DRAM since NVRAM technology was not available. Boro provided a useful system for development, testing, and benchmarking. The team coordinated node access using Slurm reservation manager.

The DAOS team maintained a DAOS installation in the shared scratch file system that was used to build HDF5 and other middleware and application layers to utilize DAOS. The installation was updated whenever bug fixes or new features were added and were needed by the HDF group.

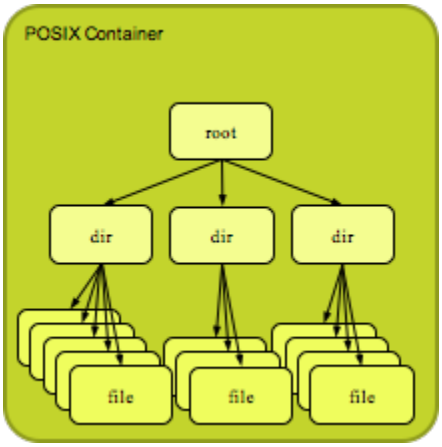
8 Future Work

8.1 DAOS

8.1.1 POSIX Legacy Support

Many applications, workflows, and middleware I/O libraries rely on POSIX for their I/O. Legacy applications using POSIX or MPI-I/O will be supported by adding a lightweight POSIX implementation using DAOS. This implementation will not support all of the POSIX standard, but most of the features that most applications rely on. This relaxed POSIX compliance will provide a highly scalable POSIX I/O implementation for data and metadata, with some extensions that can be added for asynchronous I/O and snapshots. The following figure shows a simple POSIX to DAOS mapping of an encapsulated namespace:

Figure 8-1. POSIX to DAOS Namespace Mapping



A global namespace support in DAOS requires a significant investment and amount of work, so at least for the near future, a parallel file system partition like Lustre, will be utilized to expose a system namespace to users.

In addition to POSIX, we plan to work on a DAOS backend for MPI-I/O. Other applications and middleware libraries that we plan to work with include:

- ECMWF model for weather forecasting
- ADIOS I/O library for scientific applications
- SCR with native checkpoint restart for MPI applications

8.1.2 DAOS NVMe Support

NVMe devices provide much better latency and throughput than spinning disks, but accessing those devices with the same software stack will consume a significant amount of the storage time. To address that issue, Intel has created a set of drivers and a complete, end-to-end reference storage architecture called the Storage Performance Development Kit (SPDK). SPDK

provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications. It achieves high performance by moving all of the necessary drivers into userspace and operating in a polled mode instead of relying on interrupts, which avoids kernel context switches and eliminates interrupt handling overhead.

DAOS will leverage SPDK and BlobFS which is built on top of SPDK to access the NVMe tier and take advantage of the efficient storage stack.

8.1.3 System Integration

DAOS management services should be integrated with the HPC system components. Integrating with the batch scheduler would allow the pool services to allocate DAOS pools for a user job.

As mentioned earlier, DAOS will delegate node aliveness monitoring to a dedicated service called RAS, so integration with the RAS system is important for obtaining information on the nodes' status.

8.1.4 Security Model

Because DAOS works in userspace, implementing a security model is not easy. A pool should only be accessible to authenticated and authorized jobs. The security framework to be used must be selected and configured at pool creation time. To do this, DAOS can rely on either a third-party service (such as Kerberos) or on the network transport that exports information related to each process set. An additional authorization level can be enforced when opening a container if required. By default, all containers in a pool inherit the security parameters of the parent pool. This means that once access is granted to the pool, any container can be opened. It is however possible to restrict access to a container by specifying different security rules for this container.

8.1.5 Data Analytics

For a long time, storage workloads on most HPC systems were dominated by checkpoint restart codes. However, more different workloads are emerging, particularly in data analytics with the ever increasing amount and type of data generated from HPC applications. Big data systems such as Spark, Hadoop and HDFS formats are becoming more appealing for HPC application developers. Those technologies don't particularly work well in a typical HPC system with the current storage stack as it is not suited for such I/O workloads.

We plan to explore and develop support for some data analytics frameworks in DAOS such as Apache Arrow and Spark RDD. Furthermore, we plan to provide an analysis shipping framework, where users can send python analysis scripts to execute on DAOS servers locally to where the data is located.

8.1.6 Erasure Code

For objects protected by erasure code, the missing object shard can be reproduced from all the other object shards. All surviving members in the redundancy group are mandatorily involved. Rebuilding a shard for an erasure-coded object is compute intensive and should be distributed to different members in the redundancy group.

8.2 HDF5

8.2.1 Remaining Legacy Features

While the most important and widely used features of HDF5 were implemented for the HDF5/DAOS prototype, some features still need to be added before the plugin is ready for production. The following is a list of native HDF5 functions that are planned to be implemented:

- Attributes
  - H5Aexists()/H5Aexists_by_name()
  - H5Arename()/H5Arename_by_name()
  - H5Adelete()/H5Adelete_by_name()/H5Adelete_by_idx()
  - H5Aget_info()/H5Aget_info_by_name()/H5Aget_info_by_idx()
  - H5Aget_name_by_idx()
- Datasets
  - H5Dcreate_anon()
  - H5Dset_extent()
  - H5Dflush() (simply alias for H5Fflush())
  - H5Drefresh()
- Files
  - H5Fis_accessible() (replacement for H5Fis_hdf5())
  - H5Fget_create_plist()/H5Fget_access_plist()
  - H5Fget_info()/H5Fget_intent()/H5Fget_name()
  - H5Fget_obj_count()/H5Fget_obj_ids()
- Groups
  - H5Gcreate_anon()
  - H5Gflush() (simply an alias for H5Fflush())
  - H5Gget_info()/H5Gget_info_by_name()/H5Gget_info_by_idx()
  - H5Gget_create_plist()
- Links
  - H5Lcreate_hard()
  - H5Lcreate_external() (possibly)
  - H5Lmove()
  - H5Lcopy()
  - H5Ldelete()/H5Ldelete_by_idx()

  - H5Lget_info()/H5Lget_info_by_idx()
  - H5Lget_val()/H5Lget_val_by_idx()
  - H5Lvisit()/H5Lvisit_by_name()
  - H5Lget_name_by_idx()
- Generic objects
  - H5Oopen_by_idx()
  - H5Olink()
  - H5Ocopy()
  - H5Ovisit()/H5Ovisit_by_name()
  - H5Oexists()_by_name()
- Property list routines
  - H5Pset_link_creation_order()
  - H5Pset_char_encoding()
  - H5Pset_create_intermediate_group()
  - H5Pset_elink_cb()/H5Pset_elink_prefix()/H5Pset_elink_fapl()/H5Pset_elink_acc_flags()
  - H5Pset_fill_value()
  - H5Pset_virtual()/H5Pset_virtual_view()/H5Pset_virtual_printf_gap()
  - H5Pset_obj_track_times()
  - H5Pset_attr_creation_order()
  - H5Pset_copy_object()
  - H5Padd_merge_committed_dtype_path()
  - H5Pset_mcdt_search_cb()
- References
  - H5Rcreate()
  - H5Rdereference()
  - H5Rget_obj_type()/H5Rget_region()/H5Rget_name()
- Committed datatypes
  - H5Tcommit_anon()

The functions to support creation order, H5Pset/get_link_creation_order() and H5Pset/get_attr_creation_order(), as well as the related fields of the structures returned by H5Lget_info and H5Aget_info, will be particularly problematic because of the independent nature of DAOS. We will probably want to design the object ID allocation scheme to allow us to extract creation order as well when requested, and we will need to implement a similar scheme for attributes (though no object IDs are necessary).

8.2.2 New Features

8.2.2.1 Asynchronous I/O

The asynchronous I/O implementation in the prototype is currently on a branch off the mainline repository. This branch therefore needs to be merged back, and asynchronous I/O

needs to be implemented for the features that were added since the branch was created (and for all features added in the future).

8.2.2.2 Contexts

While the automatic handling of transactions/epochs has been a great asset to application porting, some applications may want more explicit control. We plan to extend the context API created for asynchronous I/O to also handle transactions/epochs. When a context is created, it can be associated with an epoch, and this context can be used to explicitly commit and advacne epochs, as well as read from previous epochs without the need to create a snapshot. To enable the latter feature, we will need to disable automatic epoch slip in DAOS when contexts may be used with epochs, and the library will keep track of open contexts and slip explicitly as they are closed (the application will not need to handle epoch slips).

8.2.2.3 Query/Indexing and Analysis Shipping

One feature that was implemented for the original HDF5/IOD plugin was metadata indexing. Indexing metadata using operations issued by the client would prove inefficient due to many small I/O requests, therefore we plan to implement these operations on the DAOS server. In the IOD plugin, due to its innate client/server architecture it was relatively easy to execute the indexing operations on the server. For the new DAOS plugin, however, we will need to implement a DAOS server plugin to execute the indexing operations.

8.2.2.4 Map Objects

While map objects were implemented for this prototype and performed as expected, we wish to add a way for the application to list all key/value pairs in a map. Therefore we plan to implement a H5Miterate() function, which will make a callback for every key in the map. The application can then get the key's value, simply store the key in a list, or take any other conceivable action.

8.2.2.5 End-to-End Data Integrity

DAOS provides a facility to send a checksum with writes and receive that checksum with reads. We plan to implement an option in the HDF5/DAOS plugin to enable the use of these checksums, calculating them on write, and calculating them again on read for the received data and comparing against the checksum returned by DAOS. This will ensure that the application's data is not corrupted. For large datasets with small datatypes, the current scheme may prove inefficient, so we may consider storing only the checksum for each chunk and doing I/O only on entire chunks when checksums are enabled.

8.2.3 Testing

While a regression test was written for the DAOS plugin, it is not thorough enough to use for production, and does not cover every case written for in the code. The regression testing effort will consist of two distinct phases. First, we plan to create a subset of the native HDF5

regression test that is applicable to the DAOS plugin. This work may also be applicable to generic VOL testing. Second, we will need to create tests for the new and unique features of the DAOS plugin, and tests designed to exercise specific code paths in the plugin code. These tests will of course be specific to the DAOS plugin and will not be run for other plugins.

8.2.4 HDF5 Tools

The HDF5 tools currently do not support use with any VOL plugin, including the DAOS plugin. In order to support the use of the tools we will need to allow registration of VOL plugins within the tool, possibly using a dynamically loaded library scheme similar to how data filters work in native HDF5. We will also need to avoid using any features of HDF5 that are specific to the native plugin when operating with a non-native VOL plugin. While this work is mostly part of the generic VOL productization effort, the DAOS plugin will also need to be updated to make sure this works, including adding a way to pass through the pool uuid and group, possibly through an environmental variable.

8.2.5 Productization

There is a substantial amount of additional work needed in order to make the DAOS plugin ready for production. In order to be more robust to application programming errors, and make them easier to find, there need to be more checking of API input parameters, and more assertions need added in the code. In addition, it would be beneficial to add a “debug mode” to the plugin which performs additional server side checks, for example to make sure the same object is not created twice. Currently this condition does not immediately return an error but can cause confusing behavior later on.

Currently, object IDs are not assigned collectively. This means that objects must either be created in collective mode, or always created by the same process. In order to remove this limit we will need to either use an object ID allocator provided by DAOS, or implement our own method to allocate object IDs in a way that ensures independent processes do not interfere with each other.

9 References

[1] Habib, S., Pope, A., Finkel, H., Frontiere, N., Heitmann, K., Daniel, D., Fasel, P., Morozov, V., Zagaris, G., Peterka, T., Vishwanath, V., Lukic, Z., Sehrish, S., and Liao, W.-k. (2014). HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures. 4.

[2] HACC (2016). http://trac.alcf.anl.gov/projects/genericio.

[3] CLAMR (2017). https://github.com/lanl/CLAMR.

[4] Legion (2017). http://Legion.stanford.edu/.

[5] Legion (2017). https://github.com/StanfordLegion/Legion.git.

[6] Gasnet (2017). https://gasnet.lbl.gov.

[7] Watkins,N.,Jia,Z.,Shipman,G.,Maltzahn,C.,Aiken, A., and McCormick, P. (2015). Automatic and transparent I/O optimization with storage integrated application runtime support. Proceedings of the 10th Parallel Data Storage Workshop on - PDSW '15, pages 49–54.

[8] http://ncar.github.io/ParallelIO/decomp.html

[9] https://groups.google.com/forum/#!topic/parallelio/vtvOXP-sjZE

[10] NetCDF (2016). www.unidata.ucar.edu/software/netcdf/.

[11] HACC (2013). Blasting Through the 10 Petaflops Barrier: HACC on the BG/Q, presentation. http://press3.mcs.anl.gov/salman-habib/files/2013/05/hacc_pflops.pdf

[12] PMIX: https://pmix.github.io/pmix/

[13] Argobots: Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Adrian Castello, Damien Genet, Thomas Herault, Prateek Jindal, Laxmikant V. Kale, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, and Pete Beckman. Argobots: A Lightweight, Low-Level Threading and Tasking Framework, http://www.mcs.anl.gov/papers/P5515-0116.pdf, Argonne National Laboratory, 2016. http://www.argobots.org/

[14] Mercury: J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, Mercury: Enabling Remote Procedure Call for High-Performance Computing, IEEE International Conference on Cluster Computing, Sep 2013. https://mercury-hpc.github.io/

[15] CART: https://github.com/daos-stack/cart

[16] OFI: https://ofiwg.github.io/libfabric/

[17] Fast Forward I/O Stack: https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and+IO+Program+Documents

[18] Diego Ongaro , John Ousterhout, In search of an understandable consensus algorithm, Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference, June 19-20, 2014, Philadelphia, PA