Date: 25 March 2016	# M3.1 Legion on the Exascale FastForward I/O Stack Extreme Scale Storage and I/O RND

NOTICES

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

Legion on the Exascale FastForward I/O Stack

Introduction

Legion is a high-level data-centric and task-based application runtime which is intended for use with HPC and distributed systems. HDF5 is a scientific data format and library used to store data in an organized, self-describing, and high-performance manner. 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 regions (memory objects) within the runtime using a DMA mechanism to collect updates to individual datasets in an HDF5 file.

The FastForward I/O stack is a high performance parallel file system meant to enable apps to scale to next generation systems while providing an interface for versioning and snapshotting. The interface layer for the current version of FastForward is called IOD, while the interface for the version in development is called DAOS-M.

HDF5 has been implemented on top of IOD, with many extensions to the HDF5 interface to expose transactions, read contexts, and other new FastForward I/O features not covered in this document. We plan to implement HDF5 on top of DAOS-M when it is available. This document describes attempts to port Legion to HDF5 on top of IOD, and discusses plans for implementation on HDF5/DAOS-M.

Virtual Datasets

Virtual dataset support is a new feature that was recently added to HDF5. A virtual dataset (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 map closely to Legion's data model, so we decided to add VDS support to Legion as part of this work. This meant we needed to add support for VDS to the IOD implementation of HDF5. This ended up taking a significant amount of work, and we had to impose the limitation that source datasets must reside in the same container as the virtual dataset, due to restrictions with IOD that are described below.

Details on FastForward I/O

In IOD and DAOS-M, the analog for the traditional file is called a “container”, which consists of a collection of key-value store and byte array objects. In IOD, all processes are required to open a container collectively, whereas in DAOS-M, this requirement will be relaxed to allow containers to be opened independently. When an application wants to write to a container, it must first start a transaction (through the HDF5 API). To do this, the application supplies the version number for the transaction it is starting. This version number is global and must be consistent across processes, so the application must coordinate distribution of version numbers to compute processes, or ensure that processes share a transaction. When finished writing, the application commits the transaction. When an application wants to read data from a container, it similarly provides a version number and requests a read context on the container. This version number uses the same sequence of values as those for transactions, and so committing transactions determines which versions are visible for a read context. Any committed transactions with a version lower or equal to the version of the read context will be visible, while transactions above that value aren't. Finally, IOD does not allow reading from a read context when any lower transactions have not been committed. DAOS-M, however, will not have this limitation.

Implementation on IOD

General Approach

In order to port Legion onto the FastForward stack, we decided to extend the existing Legion+HDF5 implementation to handle both normal HDF5 and FastForward HDF5. The primary interface for this is a new function, *attach_hdf5_ff*, which mirrors the existing *attach_hdf5* function, but instead instructs Legion to use the FastForward interface and thereby create the file as an IOD container. Subsequent requests to the HDF5 layer check if it is a FastForward file, and uses the new HDF5/FastForward API calls if so.

In the initial implementation, all processes opened the same transaction and read context, with the transaction version number being one higher than the read context version. In order to be able to read back data that was written, all processes were required to collectively call another function, *commit_update_hdf5_ff*, which committed the transaction, opened it as a read context, and started a new transaction with a version number one greater than the previous.

Resolved Issues

One issue that came up during testing involved the threadsafety of HDF5/FastForward when compiled with the threadsafe option. Threadsafety in HDF5 is implemented with a simple global lock protecting the entire library. Since HDF5/FastForward can be run with both the client and server process in a single process (co-resident mode), and the server can make HDF5 calls when the client needs to wait for a response, it must drop the threadsafe lock to allow the server to proceed and avoid deadlock. This causes concurrency issues when multiple client threads are making HDF5 calls, as other threads can enter the library while one is waiting, and the HDF5 library is not designed to allow internal concurrency. It is therefore necessary to implement a secondary lock at the Legion level to prevent concurrent access to HDF5. While Legion actually already attempted to do this for the standard HDF5 implementation, it uses read-write locks and only requests a read lock for some operations, allowing multiple “read” operations into the HDF5 library at the same time, which is not supported.

Unresolved Issues

The original implementation with IOD used a global transaction model and required a collective management of container versions at the application level to increment the version numbers so data written could be read back. This caused problems with the Legion port because we could not find an easy way to have Legion execute the call to increment the versions exactly once in each process.

The next implementation relied on assigning each task a unique index within the index space, using this index to generate a unique transaction version for each task. It was assumed that each task would make the same number of write calls. When reading, the task would acquire a read context for the highest version number that had been committed, as calculated using the number of tasks and the highest task index. Unfortunately, we could not find a way to generate these task indices.

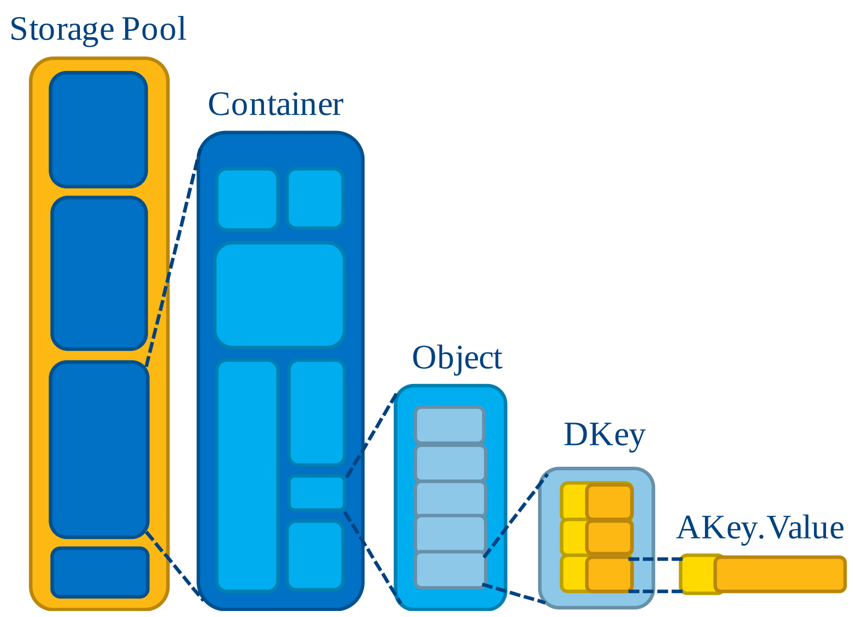
The final implementation relies on the fact that each subregion in Legion is uniquely assigned to a single process, with all I/O for that subregion going through that process. It partitioned the version number space, with each process keeping an independent stream of version numbers within the partition so the versions did not need to be coordinated between processes. It was believed that IOD could read from any version that had been committed, but it turned out that IOD and the older version of DAOS require that all previous versions be committed. Consequently, this implementation could not work since processes with rank greater than 0 would never be able to acquire read contexts unless the other processes used their entire version number partition.

Another issue involves the implementation of VDS on HDF5/FastForward. IOD's requirement that all container opens be collective causes problems when attempting to access a source dataset located in another IOD container. The client would have to either broadcast a message to all other clients to participate in the collective open, or all clients would have to always keep all of the containers for source datasets open. Either option would hurt performance significantly. Since the container open limitation is planned to be addressed with DAOS-M, we decided on, for now, requiring source datasets to reside in the same container as the virtual dataset. This requires “fooling” Legion into thinking it's opening a file when it's really opening a group, and resulted in making adjustments to the file set-up routines in the Legion examples. We will remove these workarounds in the next version.

Plan with the upcoming DAOS-M

The new DAOS stack should improve our ability to implement Legion on top of the FastForward stack due to several planned features. We will first present the DAOS storage and transaction model and then study how Legion could support this new storage paradigm.

DAOS Storage Model



The figure above represents the fundamental abstractions of the DAOS storage model. A pool can host multiple transactional object stores called DAOS containers. Each container is a private object address space, which can be atomically modified and snapshotted independently of the other containers sharing the same pool. DAOS objects in a container are identified by a unique object address and are effectively key-value stores with two-level key (distribution and attribute keys) and supporting partial or atomic values.

DAOS Pool Handles

A pool is only accessible to authenticated and authorized applications. Security is enforced when connecting to the pool. To establish a pool connection, a client process calls the pool connect method with the pool UUID, the pool service address, and the requested capabilities. Upon successful connection to the pool, a pool handle is granted to the job. There is no restriction on the number of jobs that can connect to the pool.

The pool handle must be shared with any process of the job that requires access to the pool. This is achieved by calling local2global() on the pool handle to extract and dump all information relative to the pool handle to a buffer that can then be transferred to peer processes via the application communication middleware (e.g. MPI). Upon reception of the buffer, the peer processes must call global2local() to generate a local copy of the pool handle.

The workflow scheduler is responsible for reporting job failures to DAOS. DAOS will then revoke the pool handle granted to the failed job and reclaim all resources (e.g. container handles) associated with this pool handle.

DAOS Container Handles

Access to a container is controlled by the container handle. To acquire a valid handle, an application process must open the container and pass the security checks (user/group permission, …). The container open operation must reference a valid pool handle. All container handles are attached to a parent pool handle and are stored persistently in the container metadata.

The opening process may share this handle with some or all peer processes of the same job. To do so, the same mechanism as for the pool handle is used. local2global() produces a buffer that can be transferred and consumed by global2local() to generate a local copy of the container handle. This mechanism is similar to the openg() POSIX extension. It is also worth noting that the local2global() and global2local() operations do not involve any communication with the storage system which, as a consequence, does not track the process group membership.

A set of processes sharing the same container handle is called a process group. One process may belong to multiple process groups corresponding to one or more containers. A container can be opened by multiple process groups at the same time, regardless of their open mode.

A container handle is revoked, either on explicit container close or on parent pool handle revocation by the system resource manager. In the latter case, the pool handle granted to the job is invalidated which also revokes all container handles associated with this pool handle. The explicit container close operation must be called by a single member of the process group and is effective immediately. In other words, no reference count is maintained on the container handle by the storage system and a close request from any member will invalidate the handle for the whole process group.

DAOS Operations

As mentioned in the storage model section, DAOS objects are effectively key-value (KV) stores with locality feature. The key is split into a distribution key and an attribute key. All entries under the same distribution key are guaranteed to be collocated on the same target. Enumeration of the attribute keys is provided. The value can be either atomic (i.e. value replaced on write) or a byte array (i.e. arbitrary extent read/(over)write). DAOS objects support the following operations: lookup values associated with a list of keys, update/insert KV pairs, punch KV pairs, enumeration of distribution keys as well as all attribute keys under a given distribution key. Atomic values are fully overwritten on updates whereas byte-array values support partial updates (i.e. extent read/write with arbitrary alignment).

All DAOS operations are non-blocking and associated with an event. The result of the operation execution (called event completion) is returned asynchronously through an event queue that can be polled at any time. Upon successful completion of a DAOS operation, the result of the update is immediately globally visible by all container handles. That said, it is not guaranteed to be persistent yet and might be buffered on the server. Therefore, the flush operation is provided to wait for all previously executed operations to become persistent. This assures that all caches are properly drained and updates are stored persistently.

In the future, DAOS operations might be buffered on the client side and submitted to servers asynchronously. Another level of flushing (likely through a different DAOS operation) will then be required to push all changes from a client to the servers. This feature is not in the scope of ESSIO and not considered in this design.

DAOS Transactions

The primary goal of the DAOS transaction model is to provide a high degree of concurrency and control over durability of the application data and metadata. Applications should be able to safely update the dataset in-place and rollback to a known, consistent state on failure.
Each DAOS I/O operation is associated with an explicit, caller-selected transaction identifier called epoch. This effectively means that the DAOS transactions are exported to the top-level API, which is responsible for grouping together updates to move the data model from one consistent state to another. This approach is very different from traditional database management systems, object stores and file systems that rely on transparent transactions to guarantee internal consistency.

All processes submitting I/O operations with the same container handle effectively participate in the same transaction scope. This means that updates are associated with both a container handle and an epoch number. This allows DAOS to independently rollback changes submitted by different container handles.

To modify a container, a process group must first declare its intent to change the container by obtaining an **epoch hold**. This operation must be performed by a single member of the process group and can take a minimal held epoch as input parameter. On successful execution, the handle lowest held epoch (i.e., **LHE**) is returned to the caller. The process group is now expected to commit every epoch greater than or equal to the LHE.

Once an epoch hold is obtained, the process group can submit changes against the any epoch higher or equal to the LHE. Once all updates up to a given epoch have been *submitted* and *flushed*, the process group can commit this epoch. **Epoch commit** is performed by any member of a process group, to atomically make all updates submitted by the process group, in all epochs up to the commit epoch, durable. By committing, the process group guarantees that all updates in the transaction have been applied to its satisfaction. On successful commit, the handle highest committed epoch (i.e. **HCE**) is increased to the committed epoch and the LHE to the handle HCE + 1. It is the responsibility of the programming model, library or user to determine if all members of the process group have contributed changes prior to commit.

All operations submitted by a process group tagged with an epoch smaller or equal to the HCE are guaranteed to be applied and durable. On the other hand, operations tagged with epoch greater than the handle HCE are automatically rolled back on failure.

A hold can be released at any time, which causes the container handle to become quiescent. On close, the hold is automatically released, if not already. This causes uncommitted updates submitted by this process group to all future epochs to be discarded. A member of the process group can also invoke the discard operation directly, to discard all its updates submitted in a given range of uncommitted epochs.

DAOS also tracks and exports the smallest and highest HCE across all the container handles. The highest globally committed epoch (**HGCE**) is the smallest HCE across all the container handles and is guaranteed to have immutable data. The highest partially committed epoch (**HPCE**) is the highest HCE. Epoch hold is guaranteed to return a LHE strictly higher than the HPCE.

Moreover, any container handle is granted a default read reference on the current HGCE. This reference, called the handle lowest referenced epoch (i.e., LRE), guarantees that the current HGCE, as well as any future globally-committed epochs, remain readable and thus cannot be aggregated. The handle LRE can be moved forward to a newer globally-committed epoch through an explicit call to epoch slip. A process group can **snapshot** any epoch smaller than its HCE and greater or equal to its LRE. A snapshot is a persistent reference that isn't associated with any container handle. The snapshotted epoch is guaranteed to be readable by any handle until the snapshot is explicitly destroyed.

At the container level, the container metadata tracks the lowest globally referenced epoch (**LGRE**), equal to the smallest LRE across all container handles. Epoch aggregation is triggered each time the container LGRE is moved forward (as a consequence of slip or container close). When all container handles are closed, the container LGRE is equal to the container HGCE, which is then the only available unnamed container version.

As a summary, the typical flow of a DAOS transaction is the following:

1. Open the container
2. Obtain an epoch hold
3. Submit I/O operations against an epoch >= LHE
4. Flush all the operations
5. Commit the epoch, the HCE is set to the committed epoch and the LHE is increased to HCE+1
6. Goto 3 if more updates must be submitted
7. Release the epoch hold
8. Close the container

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.

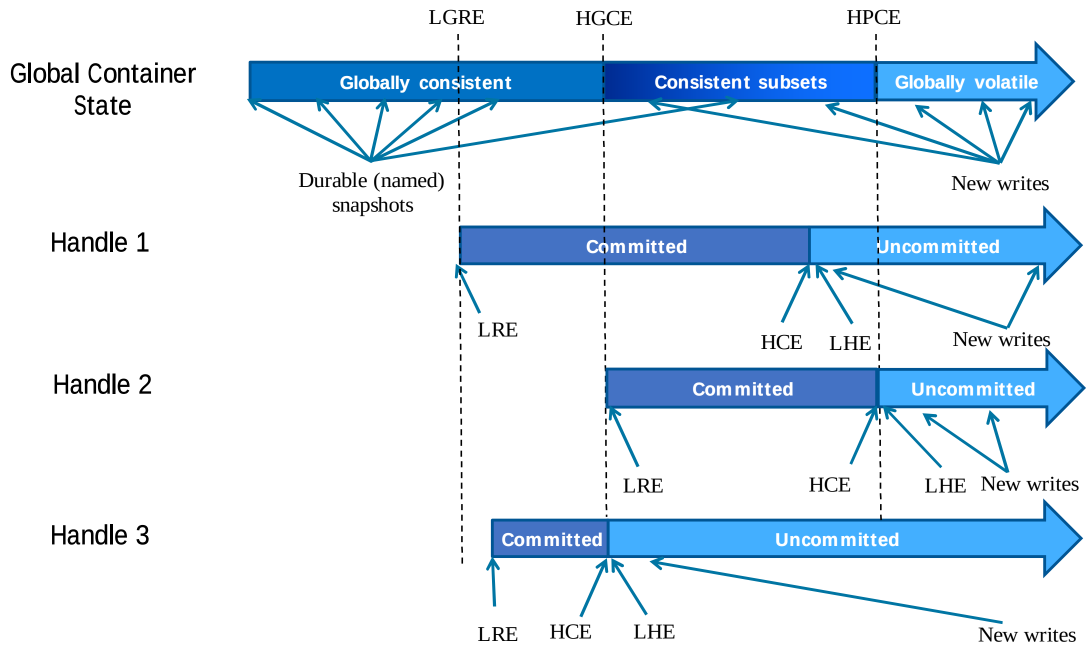
· *HCE*: highest epoch committed by this handle. Any changes submitted by this container handle with an epoch <= HCE are guaranteed to be durable. On the other hand, any updates submitted with an epoch > HCE are automatically rolled back on failure of the container handle. The HCE is increased on successful 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.

The figure below represents the container with three active handles.



Conflict Resolution & Concurrency Control

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. 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. 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, provided that 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 serializability or recoverability rules.

That been said, 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 mechanism 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 won't be reordered. This means that concurrent overlapping I/O operations are not guaranteed to be properly serialized and will generate non-deterministic results.

Legion Use Case

Let's now see how Legion could leverage the DAOS epoch model. In this proposal, we are 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. This will be measured and verify with the DAOS prototype.

When the legion framework starts, the top-level task can connect to the DAOS pool and pass the pool handle (either the global buffer or just the pool handle UUID) on-demand to the low level runtime (e.g. when attaching to a region). Alternatively, the workflow scheduler could connect to the pool on behalf of Legion and pass the pool handle as an argument or 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 also returns the current value of HPCE. A read-only operation on this container could then read with an epoch number equal to the HPCE and eventually refresh its knowledge of the HPCE though a DAOS query whenever required (e.g.

notification from other processes).

To modify the container, the legion runtime process must obtain an epoch hold. This operation returns a LHE which is guaranteed to be higher than the current HPCE (typically HPCE+1). The legion runtime process can then read from the LHE and submit new updates against this epoch or the next ones. To complete any write operations, the runtime should flush and commit 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.

When two tasks want to communicate through the storage system, the producer task must first flush and commit its changes. Once notified, the consumer can query the current HPCE (or the legion framework can pass the epoch number from the writer to the reader) and read from this epoch. If the consumer wants to submit new changes resulting from the data generated by the producer, it must then use an epoch number higher than the HPCE.

If one task fails, the Legion scheduler should close the container handle on behalf of the failed task. DAOS will roll-back any uncommitted updates submitted by this task and will report completion of the close operation. Once the close has completed, the Legion scheduler can restart the failed task.

If the whole Legion runtime fails, the workflow scheduler informs DAOS and revokes the pool handle. This automatically closes all container handles allocated to the Legion tasks and rolls back any uncommitted updates.

In Conclusion

Implementation of Legion on top of IOD proved to be difficult due primarily to a conflict between Legion's highly decentralized task execution model and IOD's strict global version number scheme. We could not find a reasonable way to either generate a globally consistent set of version numbers for each task, or have all tasks coordinate operations on a sequence of shared version numbers. We have outlined enhancements to DAOS-M that should allow us to implement Legion on top of the Exascale FastForward I/O stack by adding flexibility in transaction number sequencing and allowing independent container opening. These improvements should also aid implementation of other highly decoupled parallel applications on top of FastForward.