Date: December 13, 2012	# SOLUTION ARCHITECTURE - HDF # FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Revision History

Date	Revision	Author
2012-12-14	1.0 Draft for Review	Quincey Koziol, HDF Group Ruth Aydt, HDF Group

Introduction

The following solution architecture applies to the Extreme-Scale Computing Research and Development Storage and I/O contract signed 9/21/2012. Three main areas that together cover the Exascale I/O stack from top to bottom will be researched, developed, and demonstrated as a single I/O stack. These areas are (1) Application I/O, (2) I/O Dispatcher (IOD), and (3) Distributed Application Object Storage (DAOS). The utility of the I/O stack will be demonstrated by using it to create, store, and run computations on Arbitrary Connected Graphs (ACGs).

This portion of the architecture document describes the Application I/O area, which includes:

An object-storage API based on HDF5 to support high-level data models, their properties, and relationships

An HDF5-IOD Virtual Object Layer (VOL) plug-in to translate I/O requests made by the application via the HDF5 API into IOD API calls.

A Function Shipper to ship IOD and POSIX function calls made on a Compute Node (CN) to an IO Node (ION) for execution, and ship the function results back to the caller.

An Analysis Shipper to ship data reduction and transformation analysis operations to IONs or storage servers for execution, and ship the analysis results back to the caller.

The existing HDF5 Data Model includes *H5File*, *H5Group*, *H5Link*, *H5Dataset*, *H5Attribute*, *H5CommittedDatatype*, and *H5Reference* objects. Applications describe and operate on data and metadata in terms of these *H5Objects* via the HDF5 API[1]. Currently, an H5File is stored in the *HDF5 File Format*[2] as a file on disk or a bytestream in memory.

For the Exascale I/O stack, work will be done on the HDF5 Data Model, API, and storage format. The HDF5 Data Model will be extended to support pointer datatypes as needed for ACGs. The HDF5 API will be extended to expose and leverage the performance and data-integrity features of the IOD. *H5Files* will be instantiated in a new storage format defined in terms of the containers, structured objects, and key-value stores offered by the IOD.

In addition to the HDF5-specific components of the Application I/O area, a Function Shipper and an Analysis Shipper will be developed to allow Application I/O to take place on the appropriate hardware in the Exascale system.

[1] http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html

[2] http://www.hdfgroup.org/HDF5/doc/H5.format.html

Solution Requirements

1. Application I/O

All requirements below that are not explicitly marked as *optional*, are *mandatory* for successful completion of acceptance criteria.

1.1. HDF5 modifications to support Application I/O

1.1.1. HDF5 support for pointer datatypes

The HDF5 Data Model and API will support pointer datatypes, as needed by the Arbitrary Connected Graph work that is part of this contract.

1.1.2. HDF5 support for asynchronous I/O

The HDF5 API, building on the capabilities of lower levels in the Exascale I/O Stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS), will allow an application to specify asynchronous (non-blocking) I/O operations, and test for, or wait for, completion of those operations.

- Within a single process of an MPI application, it will be possible to specify that a given asynchronous I/O operation should not begin until another previously-issued asynchronous I/O operation has completed.

1.1.3. HDF5 support for transactional I/O

The HDF5 API will allow an application to perform transactional I/O, building on the capabilities of the IOD layer. Within a given transaction, either all I/O resulting from HDF5 calls will be completed at the IOD level or none will be completed.

- Multiple processes in an MPI job can participate in a transaction.
- A given transaction can only encompass operations on a single H5File.

1.1.4. HDF5 support for end-to-end data integrity

The HDF5 API will support end-to-end data integrity by returning an error if application data or metadata corruption is reported by lower levels of the Exascale I/O Stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS). (Note that actual checksum computations to ensure end-to-end integrity on data accessed will occur in the HDF5 IOD VOL Plugin, not in the HDF5 layer proper)

1.1.5. HDF5 support for data usage hints

The HDF5 API will allow an application to supply hints regarding lifetime and access patterns for H5Objects. The hints may be used to optimize storage layouts, prefetching, and caching.

1.1.6. HDF5 support for index building, maintenance, and query

An HDF5 index interface will be developed to support the creation, storage, maintenance, and query of indices on data values stored in H5Dataset objects as well

as metadata about HDF5 objects. An existing index can later be queried to identify HDF5 data and objects of interest for in-depth analysis. The HDF5 index API will provide an interface for adding user-defined indexing modules that can be used to create and store indices for later use during query operations.

1.1.7. HDF5 support for additional IOD and DAOS capabilities

Additional IOD and DAOS capabilities will be exposed to the application via the HDF5 API or other HDF5-based high-level interfaces to support application I/O using the Exascale I/O stack. The exact capabilities will be determined as the lower layers in the stack harden, but we anticipate capabilities such as selectively flushing HDF5 objects in the burst buffer to DAOS, pre-fetching HDF5 objects from DAOS to the burst buffer, making a snapshot at the DAOS level, and cleaning up open IOD transactions when a compute node fails.

1.1.8. HDF5 support for Python wrappers

Python wrappers will be written for all new HDF5 API capabilities, adding to the existing capabilities of the 'h5py' package[3].

1.2. HDF5 IOD VOL Plugin

The HDF5 IOD VOL plugin translates an application's HDF5 API calls on compute nodes to IOD calls on I/O nodes, interacting with the IOD container that resides on the I/O node.

1.2.1. IOD Storage

The IOD VOL Plugin will efficiently map HDF5 data model objects and operations to IOD storage objects and functionality.

1.2.2. Asynchronous operation

The IOD VOL Plugin will provide full support for asynchronous execution of HDF5 operations by calling asynchronous IOD operations.

1.2.3. Isolating operations within transactions

The IOD VOL Plugin will provide support for isolating HDF5 operations within transactions. Transaction information will be gathered by the HDF5 API and implemented by calls to the IOD layer.

1.2.4. Support for end-to-end data integrity

The IOD VOL Plugin will call IOD API routines that accept checksums wherever they are available, to ensure end-to-end integrity of data from the application layer, to the final storage in DAOS. When writing data, the IOD VOL Plugin will perform the checksum locally, on CNs, and pass that checksum, along with the data to store, to the IOD layer. When reading data, the IOD VOL Plugin will validate the checksum from the IOD layer against the data received.

[3] http://code.google.com/p/h5py/

1.2.5. Use of Function Shipper

When operating in a mode where the HDF5 library (and IOD VOL Plugin) is running as part of an application on a compute node, all IOD calls will be transparently sent to the I/O node with the Function Shipper capability (described below). When the HDF5 library is co-resident on a node with the IOD layer, the Function Shipper layer will be eliminated and IOD calls made directly.

1.3. Function Shipper

The Function Shipper will take I/O calls issued on a compute node, locally encode them, send them through the network to an I/O node where they in turn get decoded and executed—with the result being sent back to the issuing node.

1.3.1. Efficient bulk data transfer

The Function Shipper will support bulk data transfer in an efficient manner, according to the network transport method chosen.

1.3.2. Asynchronous operations

The Function Shipper will support asynchronous client operations. Client operations will be issued to the server and will return immediately to the caller, allowing the operation to complete asynchronously on the server, with notification returned to the client, where it can be checked for completion (or waited on).

1.3.3. Operation forwarding

The Function Shipper architecture will support forwarding POSIX and IOD operations as well as allowing the definition of new operations in a flexible manner. New operations will be able to be defined, shipped and executed without modifying the core Function Shipper code.

1.3.4. Transport layer abstraction

The Function Shipper architecture will support the addition of new network transport layers in a modular manner. A modular interface for adding network transports will be implemented in a way that allows users of the Function Shipper to define and choose new transport mechanisms without modifying the core Function Shipper code.

1.3.5. Data Integrity (*optional* requirement)

The Function Shipper will ensure the integrity of the data and operations it forwards from a client to a server.

This requirement is marked optional because the data integrity features provided by other layers of the storage software stack are sufficient to ensure end-to-end data integrity of application data.

1.4. Analysis Shipper

The Analysis Shipper will accept Python scripts that call HDF5 index and query API routines (as well as other HDF5 API routines), launch the script on the I/O nodes or

directly on DAOS storage nodes, execute the script on the I/O or storage node's container and then send the query results (which will include lists of HDF5 objects and/or selections within those objects, but not data elements) back to the user that initiated the query.

1.4.1. Analysis Shipper script execution on I/O and storage nodes

The Analysis Shipper will use the system's scheduler facility to launch applications that execute Python scripts containing query operations directly on I/O and storage nodes. These scripts may make use of the complete HDF5/IOD/DAOS software stack for query operations, etc. The script will execute I/O operations locally on the node it resides on, and stream only results from query operations back to the user, avoiding transfer of unnecessary data from the server.

1.4.2. Analysis Shipper launcher operation

The Analysis Shipper will provide a launcher component on compute nodes and remote workstations that can accept Python scripts to execute on the system's server nodes. The Analysis Shipper launcher will send Python scripts to the server nodes and will return the resulting information to the user.

1.4.3. Analysis Shipper results

The results returned from an Analysis Shipper script can be either lists of H5objects, selections of elements within H5Datasets, or some combination of both. The user calling the Analysis Shipper will use the lists of H5Objects or H5Dataset selections to perform further I/O operations on the H5File.

Use Cases

1. Application I/O

1.1. HDF5 modifications to support Application I/O

Parallel applications will organize their data and metadata using the extended HDF5 object data model, which will include support for pointer datatypes.

Using the extended HDF5 API, cooperating processes in a single MPI application will create, read, and write H5Objects, with an H5File being the H5Object that maps to a "container" in the IOD and DAOS layers of the Exascale I/O stack.

To realize the performance and resilience offered by lower layers of the Exascale I/O stack, the application will use the Exascale I/O features (asynchronous I/O, transactional I/O, end-to-end data integrity, and data usage hints) in the extended HDF5 API, and will coordinate the interactions of the application's processes with the H5File.

Creation of H5Objects in an H5File may be performed by either a single process (in the case of objects that will be only used by that process), or a group of processes (in the case of objects accessed by all those processes).

The application will be responsible for insuring that writes to different parts of an H5Dataset will not overlap, and will use transactions to group and order writes and reads.

1.1.1. Pointer datatype

The driving use case for pointer datatypes is the Arbitrary Connected Graph work that is part of this contract. A clear definition of this use case and determination of how the HDF5 Data Model needs to be extended with pointer datatypes to support the ACG work will be undertaken in Q1 of 2013.

1.1.2. Asynchronous I/O

Exascale applications will issue asynchronous (non-blocking) calls via the HDF5 API. These calls can create, read, or write H5Objects.

Use case

Create an H5File with two H5Datasets in different H5Groups. Perform compute/write phases to the datasets from different nodes

Phase 1: **Create H5File and define structure**

At startup, rank 0 in an MPI application asynchronously creates a new H5File (F1) with one H5Group (/G1), and two H5Datasets (/D1 and /G1/D2). After some other activity, rank 0 of the application waits until all of the asynchronous operations creating H5Objects have completed, and then broadcasts information about F1 to other ranks. *Requires ordered execution of create operations, /G1 and /D1 can't be created until F1 creation completes; /G1/D2 can't be created until /G1 creation completes.*

Phase 2: **Compute and update cycles**

Rank 0 and ranks 1-N all enter independent compute/write phases, with rank 0 writing to /D1 and ranks 1-n writing to different elements in /G1/D2. All writes are asynchronous and allow the application to overlap compute and I/O. When a compute phase finishes on one of the ranks, it waits until its previous asynchronous write completes before issuing the next asynchronous write request and entering the next compute phase. *Requires ability to asynchronously update elements in single dataset from multiple CNs. Application responsible for coordination which elements are updated by which nodes. Note there is no guarantee re: ordering of writes across nodes. Transactions (or application inter-node communication) would be needed to do that, and in this use case each asychronous I/O request is effectively in a single-operation transaction.*

1.1.3. Transactional I/O

Processes and threads in a parallel application can proceed to update data in a given H5File in an asynchronous, concurrent, and coordinated manner through the use of transactions. Applications will use the HDF5 API to obtain and assign transaction identifiers to individual HDF5 operations, with a single identifier used by multiple operations on a single node, or by multiple operations across compute nodes. Many transactions can be open at any given time, allowing – for example – nodes to proceed at different rates with computing and writing ACG data.

The HDF transaction support will depend heavily on the underlying layers in the stack, and details of the use case scenarios have not yet been solidified. The HDF5 transaction interface design is a deliverable for Q2 2013.

1.1.4. End-to-end data integrity

Scientists using HDF5 wish to detect data corruption in H5Files that is introduced during the process of storing and/or retrieving their application data. The HDF5 library will report information about data corruption detected during H5File access through the existing HDF5 error API. Layers below the HDF5 library proper (such as the HDF5 IOD VOL Plugin, Function Shipper, IOD, and DAOS) are responsible for detecting and reporting data corruption in a consistent and thorough way, for the HDF5 library to report to the application. The "chain of checksums" will begin when the HDF5 IOD VOL Plugin (described below) sends data from the application memory to the IOD layer and will end when the HDF5 IOD VOL Plugin verifies the data it retrieves from the IOD layer to store in application memory.

1.1.5. Data usage hints

An application developer wishes to indicate a non-default behavior for accessing or storing H5Objects. Some examples of these behaviors could include:

- "This object will be accessed by only this process"
- "The rows in this 2-D H5Dataset will be accessed sequentially"
- "Perform this operation collectively, with this sub-group of processes"

The set of behaviors and hints will be determined as the project proceeds, based on from feedback from users and application developers. In all cases however, the mechanism for passing a hint to the lower layers of the stack will be the same: the HDF5 library will define a new HDF5 property list API routine that an application will call with appropriate parameters, and then pass that property list to future HDF5 API calls. The HDF5 library will either take action internally based on the HDF5 properties set or translate those properties into the appropriate mechanism to influence behavior from lower levels of the I/O software stack.

1.1.6. Index building, maintenance, and query

A scientist wishes to perform data analysis queries on one or more H5Objects or elements within H5Datasets. Several steps are required to facilitate this:

1. Index Creation: An application developer uses the HDF5 index API to indicate which H5Objects or H5Datasets are likely to be queried in the future, and what indexing mechanism should be used to create the indices. The HDF5 library performs the operations desired, creating index objects that are stored in the file, for later access during query operations.
2. Query: When an application developer wishes to determine H5Objects or H5Dataset values that match their criteria, they will call the HDF5 query API with a query operation describing the information they wish to retrieve. The HDF5 library uses any existing indices that exist and can improve the query performance as part of the process of fulfilling the query operations requested. Should no appropriate indices be available, the HDF5 library will perform whatever "full search" operation is necessary to fulfill the query. The query results will be stored in an H5QueryResult object and returned to the application developer.

3. New Index Method Definition: As the HDF5 data model provides a broad and deep variety of data for indexing, application developers are expected to desire new index methods and technologies that extend, enhance or replace indexing methods provided natively with the HDF5 library. The HDF5 library will define an interface for adding new index modules that applications can leverage for their queries. When an application developer has implemented a new HDF5 index module (according to the developer guidelines defined later in this project), the developer will invoke the HDF5 index API's mechanism for registering their new index module with the HDF5 library, for later use by index creation & query operations (steps 1 & 2 above).

1.1.7. Additional IOD and DAOS capabilities

Applications may want to selectively discard HObjects in the IOD – for example, when a checkpoint is "out of date". Applications may also want to write HObjects in the IOD to DAOS. For example, "write every 3rd checkpoint to storage from the BB".

1.1.8. HDF5 support for Python wrappers

See 1.4.1, 1.4.2, and 1.4.3 for use cases where Python wrappers will be needed.

1.2. HDF5 IOD VOL Plugin

1.2.1. Use of the IOD/DAOS storage system

Application developers using the HDF5 interface wish to efficiently store their data on the IOD/DAOS storage system. An application developer running on an "exascale system" (i.e. with separate compute nodes and I/O nodes) uses the HDF5 API for selecting the IOD VOL Plugin and links in the Function Shipper code, allowing their application data to be remotely stored in IOD containers. When running on a system where each node contains the full HDF5/IOD/DAOS stack, the Function Shipper is not linked in, but the application operates without other change.

1.2.2. Asynchronous operations

Application developers using the HDF5 interface with the IOD VOL Plugin wish to perform asynchronous I/O operations on the H5File. The IOD VOL Plugin will translate all asynchronous HDF5 VOL operations into asynchronous IOD API calls and track the completion of those operations.

1.2.3. Isolation of operations within transactions

Application developers using the HDF5 interface with the IOD VOL plugin wish to isolate their HDF5 operations within a transaction, such that all the HDF5 operations within the transaction are either visible and durable or not. Application developers calling the HDF5 transaction API routines will have those operations passed to the IOD layer by the IOD VOL Plugin.

1.2.4. Support for end-to-end data integrity

Application developers using the HDF5 interface with the IOD VOL plugin wish to ensure that corrupted application data & metadata stored in an H5File will be efficiently detected. The IOD VOL Plugin will perform checksum computations on compute nodes, then call IOD API routines that accept the checksum and data together, for storage and later retrieval and verification.

1.3. Function Shipper

1.3.1. Asynchronous Metadata Operations

A client wishes to perform a metadata I/O operation on a remote system and have that operation performed asynchronously**.**

1.3.2. Asynchronous Bulk Data Operations

A client wishes to perform a bulk data I/O operation on a remote system and have that operation performed asynchronously.

1.3.3. Add new operation to Function Shipper

An application developer wishes to add a new operation to execute on the server.

1.3.4. Add new transport layer

An application developer wishes to add a new transport layer to the Function Shipper.

1.3.5. Detect dropped operations over transport layers

Client and server components must ensure that all operations issued by a client are received, executed and responded to by the server.

Note that this use case is tied to an optional requirement, and may be supported at a lower priority than other use cases.

1.3.6. Detect corrupted data over transport layers

Client and server components must ensure that all operations issued by a client are received without corruption.

Note that this use case is tied to an optional requirement, and may be supported at a lower priority than other use cases.

1.4. Analysis Shipper

1.4.1. Retrieve structure of remote H5File

A scientist wishes to efficiently retrieve a list of the H5Groups and H5Datasets in an H5File hosted on an I/O node or DAOS server. The scientist writes a Python script containing the HDF5 API calls that will retrieve the structure of the H5File. Then, the scientist uses the Analysis Shipper to send their Python script to the

storage nodes, running on a remote system and receives back the information describing the structure of the H5File.

1.4.2. Retrieve list of H5Datasets matching query criteria

A scientist wishes to efficiently retrieve a list of H5Datasets that match their search criteria. The scientist writes a Python script containing the HDF5 API calls that query for H5Datasets that match their criteria (e.g. "find all H5Datasets which store a temperature field and whose average value is greater than 32 degrees Celsius"). The scientist then uses the Analysis Shipper to send their Python script to the storage nodes running on a remote system and receives back the list of H5Datasets matching their criteria.

1.4.3. Retrieve selection of H5Dataset elements matching query criteria

A scientist wishes to efficiently retrieve the elements within an H5Dataset that matches their search criteria. The scientist writes a Python script containing the HDF5 API calls that query for the locations of the elements in an H5Dataset that match their criteria (e.g. "find all elements of H5Dataset 'X' which contain a temperature value greater than 64 degrees Celsius"). The scientist then uses the Analysis Shipper to send their Python script to the storage nodes running on a remote system and receives back an HDF5 selection object containing the set of locations for elements with the H5Dataset matching their criteria.

Solution Proposal

1. Application I/O

1.1. HDF5 modifications to support Application I/O

1.1.1. HDF5 support for pointer datatypes

The HDF5 Data Model and API will be extended to add pointer datatypes to support the ACG work. The exact meaning and treatment of the HDF5 pointer datatypes will be tailored to ensure that high-performance, scalable representations of ACGs are possible in the extended HDF5 Data Model. Possibilities under consideration include (1) within an H5Dataset element, pointer to data in that element, (2) within an H5Dataset, pointer(s) to other element(s) in the H5Dataset, (3) across H5Datasets, pointer(s) to element(s) in Dataset(s). Current H5Link, H5Reference, compound and user-defined datatype capabilities will be leveraged as appropriate.

1.1.2. HDF5 support for asynchronous I/O

The HDF5 API will be modified to allow an application to perform asynchronous I/O via extensions to existing HDF5 property lists[4], which modify existing HDF5 API routine behaviors. As necessary, new routines will be added to the current HDF5 API or a new HDF5 interface will be created.

The asynchronous I/O functionality is implemented at lower levels of the IO stack (HDF5 IOD VOL plugin, Function Shipper, IOD, DAOS).

1.1.3. HDF5 support for transactional I/O

The HDF5 API will be modified to allow an application to perform transactional I/O via extensions to existing HDF5 property lists[1], which modify existing HDF5 API routine behaviors. If necessary, new routines will be added to the current HDF5 API or a new HDF5 interface will be created.

The transactional I/O functionality is implemented at lower levels of the IO stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS).

1.1.4. HDF5 support for end-to-end data integrity

Lower levels of the IO stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS) will be responsible for detecting and reporting data and metadata corruption. The HDF5 API will be modified to make errors reported by these levels available to the application via the standard HDF5 error reporting mechanisms.

1.1.5. HDF5 support for data usage hints

The HDF5 interface will be modified and/or extended to allow an application to pass data usage hints via HDF5 property lists. The implementation approach will be determined as the R&D project progresses and the type and scope of usage hints is better understood. Expression and use of hints such as "these H5Objects will be accessed at the same time", "data in this H5Attribute will be needed by all nodes", or "rows in this 2D H5Dataset will be accessed sequentially, with one row per process" are examples of the types of hints envisioned.

Lower levels of the IO stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS) will be responsible for passing along or acting on the usage hints received.

1.1.6. HDF5 support for index building, maintenance, and query

A new HDF5 index API will provide an interface for adding user-defined indexing modules that can be used to build and store indices for use during query operations.

[4] http://www.hdfgroup.org/HDF5/doc/RM/RM_H5P.html

1.1.7. HDF5 support for additional IOD and DAOS capabilities

Additional IOD and DAOS capabilities will be exposed to the application via the HDF5 API or other HDF5-based high-level interfaces to support application I/O using the Exascale I/O stack.

Lower levels of the IO stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS) will be responsible for passing along or providing the capabilities.

1.1.8. HDF5 support for Python wrappers

Following the model of h5py[5], python wrappers will be written for all new HDF5 API capabilities.

1.2. HDF5 IOD VOL Plugin

The HDF5 library currently supports a Virtual Object Layer (VOL) interface[6] for providing methods of storing and accessing HDF5 data model objects (H5File, H5Group, H5Dataset, etc.) that are different from the current HDF5 file format. A new VOL plugin will be implemented that stores HDF5 data model objects using the IOD interface and objects. This new IOD VOL plugin will provide full access to the features available through the IOD API, including asynchronous operations and the ability to group HDF5 operations into transactions that are atomically visible and durable in the IOD container.

VOL plugin methods will be mapped to one or more IOD API calls, which are invoked within the IOD VOL plugin. HDF5 data model objects will be mapped into analogous IOD objects, per the following table:

HDF5 Object	**IOD Object**
H5File	Container
H5Group	K-V Store
H5Dataset	Array
H5CommittedDatatype	Blob

[5] http://code.google.com/p/h5py/

[6] https://confluence.hdfgroup.uiuc.edu/display/VOL/Virtual+Object+Layer

H5Attribute	Key/Value pair in Object K-V Store[7]
H5Link	Key/Value pair in H5Group K-V Store

Table 1 Mapping from HDF5 Objects to IOD Objects

1.3. Function Shipper

Using HDF5 on exascale systems will not be feasible without exporting the I/O API from I/O nodes onto the compute nodes. One solution to address this problem is to use a well-known method called function shipping. Making use of this method, I/O calls issued from the computes nodes are locally encoded, sent through the network to the I/O nodes where they in turn get decoded and executed—with the operation's result being sent back to the issuing node.

Building on the existing IOFSL[8] framework for I/O forwarding, we will extend IOFSL to enhance its asynchronous operation support, allow generic routines to be forwarded from a client (compute node) to a server (I/O node) and extend the network abstraction layer to allow new transport layers to be added in a modular fashion.

1.4. Analysis Shipper

As visualization and data analysis tasks grow more complex and the projected exascale I/O architecture more extended, a method of sending analysis tasks to operate closer to where the data is stored must be found. We will implement a method of sending data scripts from client nodes or remote workstations to execute on the I/O nodes or storage servers of an exascale system, called "Analysis Shipping".

We will build a client/server application that uses extensions to the Function Shipping framework (using its generic routine forwarding capability) to send Python scripts that perform data-intensive analysis operations from an Analysis Shipper client, located on a compute node of the exascale system or a remote workstation, to an Analysis Shipper server that runs on I/O nodes or storage servers of the system. The Analysis Shipper server will execute the Python script on the local storage objects, retrieve the results and send them back to the client. Placing the data analysis operations close to the actual storage will both remove the latency of issuing multiple I/O operations from a client and potentially greatly reduce the volume of data sent to the client.

[7] An IOD object K-V store is available on every IOD object (array, K-V store and blob), see IOD architecture & design documents for further details.

[8] Nawab Ali et al., "Scalable I/O Forwarding Framework for High-Performance Computing Systems," in *CLUSTER '09*, 2009, pp. 1-10.

1. Application I/O

This section describes key test scenarios that need to be verified to confirm each feature functions properly. It is assumed that good software engineering practices, including unit tests on all new functionality, will be practiced.

1.1. HDF5 modifications to support Application I/O

1.1.1. Pointer datatype

Use the HDF5 pointer datatype in the Arbitrary Connected Graph demonstration.

1.1.2. Asynchronous I/O

Using the HDF5 API, a parallel MPI application will issue multiple asynchronous read and write operations on H5Datasets, perform other processing, and use data read or overwrite buffers of data written after receiving notice that I/O operation is complete.

Using the HDF5 API, a parallel MPI application will issue multiple asynchronous metadata operations on H5Objects (such as creating/modifying H5Links, creating/modifying H5Attributes, etc), perform other processing, and verify that the HDF5 metadata operations have completed correctly.

1.1.3. Transactional I/O

Using the HDF5 API, a parallel MPI application will bundle multiple HDF5 write-related operations into a single transaction. The operations may be issued from a single process or from multiple processes. When the transaction completes successfully, all write operations will be visible by the IOD API. When the transaction is still open or has been aborted, none of the write operations will be visible.

Using the HDF5 API, a parallel MPI application will bundle multiple HDF5 read-related operations on an H5File that is in the process of being written into a single transaction. The reader will see consistent file contents throughout the duration of the transaction, even as the writes to the file are ongoing.

1.1.4. End-to-end data integrity

Using the HDF5 API, a parallel MPI application can detect errors reported by lower levels of the IO Stack (HDF5 IOD VOL Plugin, Function Shipper, IOD, DAOS). Errors in these lower layers will be manually introduced for testing.

1.1.5. Data usage hints

Using the HDF5 API, a parallel MPI application will supply hints regarding lifetime and access patterns for H5Objects. The hints will be used to affect behavior at the IOD and DAOS layers, with the results verified in an appropriate manner for each particular hint.

1.1.6. Index building, maintenance, and query

Using the HDF5 interface and a user-defined indexing module, an application will build an index from data values stored in H5Dataset objects. The index will later be queried to identify H5Dataset objects of interest for in-depth analysis, and those H5Dataset objects will be read from the H5File.

Using the HDF5 interface and a user-defined indexing module, an application can build an index for metadata stored in an H5File. For example, an index that contains H5Objects with a specific H5Attribute name/value pair. The index will later be queried to identify H5Objects of interest for in-depth analysis, and a list of those H5Objects will be returned to the application.

1.1.7. Additional IOD and DAOS capabilities

Using the HDF5 API or other HDF5-based high-level interfaces, applications will interact with lower layers of the Exascale I/O stack. The specific operations will be determined as the lower layers in the stack harden, and as application needs are identified.

1.1.8. HDF5 support for Python wrappers

Python scripts using the new HDF5 Python wrappers will be written that implement important exemplar cases of application I/O patterns that leverage the new HDF5 capabilities and they will be executed to verify proper operation. Python scripts that implement the query scripts described in the Analysis Shipper test case sections 1.4.1-1.4.3 below will be written and verified for correct execution.

1.2. HDF5 IOD VOL Plugin

1.2.1. Use of the IOD/DAOS storage system

Storing an H5File with the IOD/DAOS storage system will be chosen by setting an H5Property indicating that the IOD VOL plugin is to be used when storing the data, along with providing additional hints about the use of the IOD/DAOS storage. The HDF5 library will detect the property and configure its operations to use the IOD VOL plugin, accessing an IOD container for the H5File. All HDF5 API operations will be translated to IOD calls on the IOD container.

1.2.2. Asynchronous operation

Asynchronous HDF5 operations will be enabled by setting a "do async" property on an H5PropertyList, then passing that H5PropertyList to the appropriate HDF5 API routine. The HDF5 library detects the use of the "do async" property and creates an "async request" object, passing that to the IOD VOL plugin, which issues the proper asynchronous IOD operation(s), tying them to the HDF5 async request object. The HDF5 library sets an "async status" property in the H5PropertyList to the async request object and returns immediately to the calling application. The application retrieves the async status property and continues executing its code. The IOD layer asynchronously completes the operation(s) requested, and when all are complete, an application's call to test or wait on the async status object will return the status of the HDF5 operation (succeed or fail).

1.2.3. Isolation of operations within transactions

Before isolating operations within a transaction, an HDF5 transaction object ("H5Transaction") must be created first, with a transaction ID provided by the application. Once the H5Transaction object is created, it can be used to set the "transaction" property on an H5PropertyList that is passed to the appropriate HDF5 API routine. The HDF5 library detects the use of the "transaction" property, retrieves the transaction ID and passes it to the IOD VOL plugin, which uses it when making IOD calls for implementing the HDF5 operation's behavior. All IOD calls made within the transaction are made visible and durable within the IOD container when the transaction is closed (through an HDF5 API call that passes through the IOD VOL plugin to the IOD layer), or are removed from the IOD container if the transaction is aborted.

1.2.4. Support for end-to-end data integrity

Checksums are applied to data stored with IOD API calls, and are verified by reading the data and verifying that the checksum provided by IOD is correct. Proper checksum operation for data initially stored along one dimension, but read according to an orthogonal dimension will be verified. Data corruption will be simulated and verified to be detected and reported to an application.

1.3. Function Shipper

1.3.1. Asynchronous Metadata Operations

The Function Shipper will accept an operation from the client and send it to the server, enqueueing a request token on the client and returning immediately to the caller. Upon receiving a request from a client, the Function Shipper server will enqueue the operation in an operation queue, for execution by a helper thread. Helper threads on the Function Shipper server will retrieve operations from the queue, execute them, and send the status or results from the operation back to the client. When a client application checks the status of a request or blocks to wait for it, the communications from the server will be processed, and the results of the metadata operation will be returned to the client.

1.3.2. Asynchronous Bulk Data Operations

The Function Shipper will accept a bulk data operation (i.e. read or write of an application data buffer) from the client and send it to the server, enqueueing a request token on the client and returning immediately to the caller, but will not transmit the bulk data buffer immediately to the server. Upon receiving a request from a client, the Function Shipper server will enqueue the operation in an operation queue, for execution by a helper thread. Helper threads on the Function Shipper server will retrieve operations from the queue, retrieve the bulk data from client (ideally in a passive way that doesn't involve the client), execute the operation, and send the status or results from the operation back to the client. When a client application checks the status of a request or blocks to wait for it, the communications from the server will be processed, and the results of the bulk data operation will be returned to the client.

1.3.3. Add new operation to Function Shipper

Using the mechanisms defined in the Function Shipper package, the developer defines the interface for the new operation, encode/decode methods for the parameters and return value of that interface, and a callback implementation that executes the operation on the Function Shipper server. These interface and method definitions are captured in a module that is introduced into the Function Shipper package, which is then rebuilt, without other modifications. When the server is deployed, clients may then issue the new operation for execution on the server.

1.3.4. Add new transport layer

Using the interfaces provided by the Function Shipper package, the developer implements the necessary methods to provided communication across the new transport layer, including adding a new method for choosing the new transport mechanism. The new transport module is added to the Function Shipper package, which is then rebuilt, without other modifications. When the server is deployed, clients may choose the new transport layer as the method for communicating between the client and server.

1.3.5. Detect dropped operations over transport layers

Given that the set of clients for a server is static over the lifetime of a server, the Function Shipper server will maintain a operation counter for each client it communicates with and each client will label each operation with its counter, incrementing the counter after each operation is sent and enqueued. Should the server receive an operation from a client with a counter label greater than anticipated, it will prompt the client to retransmit the missing operation(s). Correspondingly, each client will label each operation it enqueues with a timestamp. Should the operation not be executed and responded by the server within a timeout period, the client will reissue the operation to the server.

Note that this test case is tied to an optional use case/requirement and may be implemented at a lower priority.

1.3.6. Detect corrupted data over transport layers

To ensure detecting corrupted data over transport layers, all operation requests, responses and bulk data transfers will have a checksum applied to the data sent. This checksum will be validated by the recipient, causing the operation, response or transfer to be reissued if an invalid checksum is detected. To avoid undesirable repeat retransmissions of large bulk data buffers, checksums will be applied periodically to portions of the bulk data buffer, allowing for partial retransmission to occur.

Note that this test case is tied to an optional use case/requirement and may be implemented at a lower priority.

1.4. Analysis Shipper

1.4.1. Retrieve structure of remote H5File

Upon receipt of a Python script querying the structure of an H5File, the Analysis Shipper launches the script on a remote storage node(s), returning only the information specified by the script.

1.4.2. Retrieve list of H5Datasets matching query criteria

Upon receipt of a Python script querying the list of H5Datasets that match a set of query criteria, the Analysis Shipper launches the script on a remote storage node(s), utilizing any available indices that are stored within the H5File. The list of H5Datasets that match the criteria is returned to the client that issued the request.

1.4.3. Retrieve set of H5Dataset elements matching query criteria

Upon receipt of a Python script querying the set of elements within an H5Dataset that match a set of query criteria, the Analysis Shipper launches the script on a remote storage node(s), utilizing any available indices on the H5Dataset queried. The set of elements within the H5Dataset that match the criteria are returned to the client that issued the request as an HDF5 selection object (which an application can then use to retrieve the elements directly).

Acceptance Criteria

Acceptance criteria for the Application I/O component of this project are:

- Demonstrate success for all key test scenarios described above, which cover the mandatory features described in the solution proposal section.

Unit testing on the functionality implemented.