Date: **March 4, 2013**	**Design Document – HDF5 API Changes (Includes asynchronous I/O, data integrity, transaction and data layout properties)** **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
Feb. 26, 2013	1.0	Quincey Koziol, The HDF Group
Feb. 27, 2013	2.0, 3.0	Quincey Koziol, Ruth Aydt, The HDF Group
Feb. 28, 2013	4.0	Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 1, 2013	5.0	Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 1, 2013	6.0	Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 4, 2013	7.0	Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group

Introduction

This document describes the design of multiple additions to the HDF5 library and API, including asynchronous I/O, end-to-end data integrity, transactions, and data layout property support. All changes for these capabilities were combined into one document for easier tracking; furthermore, because many of the features affect the same HDF5 API routines, they are easier to understand in combination.

Definitions

CN = Compute Node

ION = I/O Node

VOL = Virtual Object Layer

Changes from Solution Architecture

There are currently no changes from the Solution Architecture descriptions.

Specification

New HDF5 Library Capabilities

New functionality added to the HDF5 library is described below, with sections for each capability.

Asynchronous I/O

Support for asynchronous I/O in HDF5 will be implemented by:

1) Building a description of the asynchronous operation
2) Shipping that description from the CN to the ION for execution
3) Returning a request object back to the application while the operation completes on the ION

The application is free to continue with other actions while an asynchronous operation executes. The application may test or wait for an asynchronous operation's completion with calls to HDF5 API routines. All parameters passed to asynchronous operations are copied into the HDF5 library and may be deallocated or reused, except for the buffers containing data elements. The application must not deallocate, examine or modifie data element buffers used in asynchronous operations until the asynchronous operation has completed.

The HDF5 library tracks asynchronous operations to determine dependencies between operations. Dependencies exist between operations when a later operation requires information from an unfinished earlier operation in order to proceed. A simple "progress engine" within the HDF5 library updates the state of asynchronous operations when the library is called from the application. There is *no* use of background threads on CNs, only on the IONs, eliminating the possibility of "jitter" from background operations on CNs interfering with application computation and communication.

As a consequence of not using background threads on the CNs, when an asynchronous operation is called which has a dependency on an earlier operation that hasn't completed, the dependent operation may be delayed. When an operation is delayed waiting for an earlier operation to complete, the delayed operation will stall inside the HDF5 library until the earlier operation completes, then schedule its operation for asynchronous completion and return to the application. For example, if an application makes two asynchronous HDF5 calls: creating a dataset, following by writing data elements to the new dataset; it is possible that the data write operation may be delayed inside the HDF5 library until the dataset creation operation completes and the object ID for the dataset is available for the data write operation to use in scheduling its asynchronous write.

An application can mitigate these stalls in its asynchronous operations by issuing multiple non-dependent operations, followed by operations that depend on the earlier operations. For example, an application that wishes to create ten datasets and write data elements to each new dataset can minimize the possibility of asynchronous stalls by first creating all ten datasets, and then writing to each dataset, in the order the datasets were created. This order of operations, as opposed to creating each dataset and immediately writing to it, will give the maximum opportunity for earlier dataset creation operations to complete before their results are needed by later data write operations.

Asynchronous invocations of HDF5 routines that create or open an HDF5 object will return a "placeholder" object ID when they succeed. Placeholder object IDs can be used in all HDF5 API calls, with the HDF5 library tracking dependencies created as a result. If the asynchronous operation completes successfully, a placeholder object ID will transparently transition to a normal object ID and will no longer generate asynchronous dependencies. If the asynchronous operation fails, the placeholder object ID issued for the operation (and any placeholder object IDs that depend on it) will be invalidated and not be accepted in further HDF5 API calls. If a placeholder object ID is invalidated, all asynchronous operations that depend on it will fail.

See below, in the API and Protocol Additions and Changes section, for details on how existing HDF5 API routines are extended, along with new API routines to test and wait on a asynchronous request object.

End-to-End Data Integrity

When enabled by the application, end-to-end data integrity is guaranteed by performing a checksum operation on all application data before it leaves a CN. The checksum for the information (both data elements and metadata information, such as object names, etc.) in each HDF5 operation will be passed along with the information to the underlying IOD layer, which will store the checksum in addition to the information.

The HDF5 library will checksum application data before sending it from the CN to the ION for storage in the HDF5 container, and optionally can additionally checksum application data before copying it into internal buffers within the library (when it is copied). When data is read from the container, the IOD layer will provide a checksum with the data, which will be verified by the HDF5 library before returning the data to the application. If the checksum of the data read doesn't match the checksum from IOD, the HDF5 library will issue an error by default, but will also provide a way for the application to override this behavior and retrieve data even in the presence of checksum errors.

See below, in the API and Protocol Additions and Changes section, for details on new API routines to set properties for controlling the optional checksum behaviors.

Transactions

The HDF5 library will allow applications to atomically perform multiple operations on an HDF5 container through the use of transactions. New HDF5 API routines allow transactions to be started, aborted and/or committed, and existing API routines will be extended to accept transaction numbers that indicate which transaction each operation belongs to. Data from specific transactions may be prefetched to the IOD layer from DAOS storage, persisted from the IOD layer to DAOS storage, and removed from the IOD layer through the HDF5 API as well. Specific versions of a container may be retained with a snapshot operation, and container snapshots may be opened for reading by an application, or removed when no longer desired.

An application is fully in charge of all aspects of transactions. The application must specify the transaction number when a transaction is initiated, provide that transaction number to all HDF5 operations that occur within the transaction, and eventually conclude the transaction by committing or aborting it.

In addition to basic transaction management, changes resulting from each committed transaction are initially stored at the IOD layer, where they must be either persisted to DAOS and/or deleted – the HDF5 library will *not* manage the persist and delete operations automatically. When reading from an HDF5 container, an application may prefetch data for transactions from DAOS storage to the IOD layer, where it can be accessed more quickly.

Additional details on transaction semantics and container snapshots will be provided in Q4 of the project, as stated in the Statement of Work.

All HDF5 transaction operations may be asynchronous, with an application receiving request objects for later query, etc.

See below, in the API and Protocol Additions and Changes section, for details on how existing HDF5 API routines are extended, along with new API routines to create, commit, abort, persist, delete and prefetch transactions.

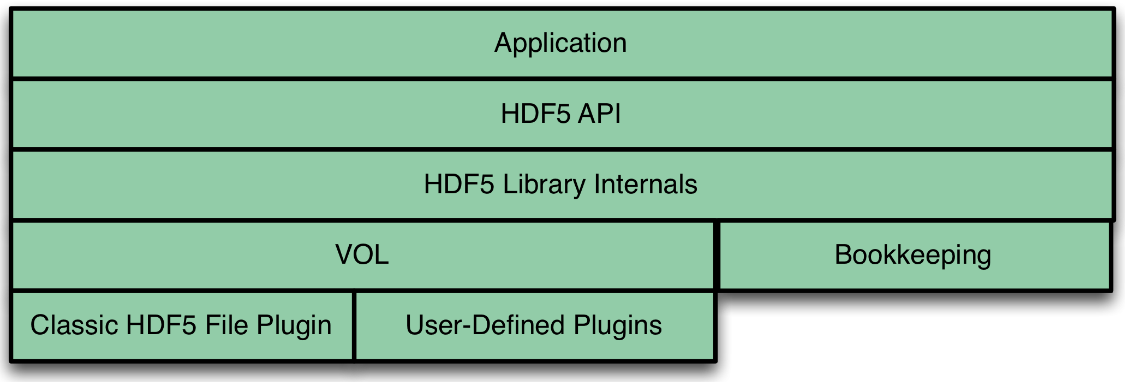
Data Layout Properties

Data layout properties, and other aspects of HDF5, IOD and DAOS software stack behavior, will be controlled by properties in HDF5 property lists (e.g. file creation, object creation, object access, etc.). New properties are set and retrieved by HDF5 API routines described below, in the API and Protocol Additions and Changes section. Existing HDF5 properties will be translated to appropriate actions on the container, e.g. the contiguous and chunked storage properties for datasets in native HDF5 containers will be used by the IOD layer to control analogous storage settings in IOD and DAOS containers. The set of behaviors controlled by properties is still under active development; more properties (and API routines to control them) will be added over the course of the project.

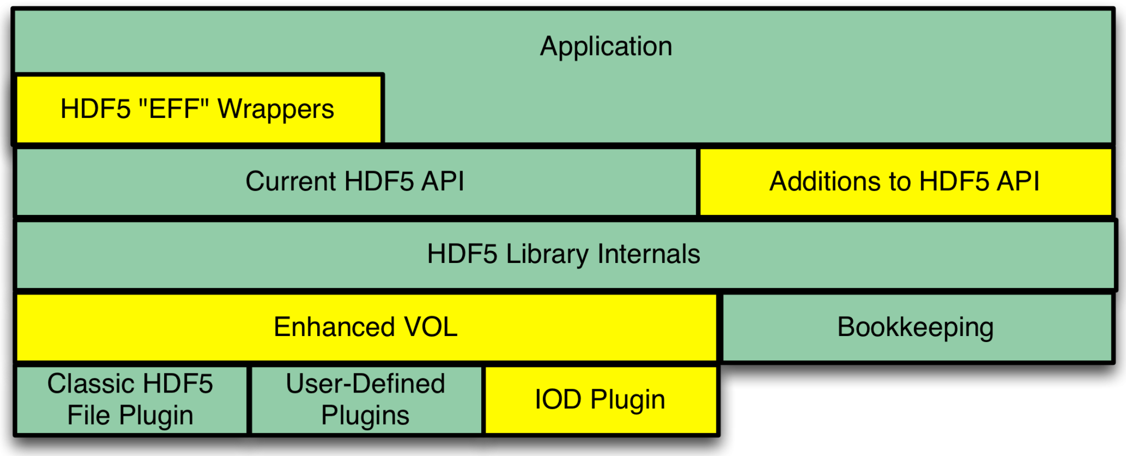
Architectural Changes to the HDF5 library

The architecture of the core HDF5 library is largely unaffected by the changes described in this document. The majority of the capabilities added to the HDF5 API are handled by a wrapper layer above the main HDF5 library, and a small number of additions to the main API routines (details of these API changes are described below in the API and Protocols Changes section). Adding transactions requires extending the VOL interface to incorporate some additional callbacks and/or parameters as well. Fortunately, the VOL is already designed to support asynchronous operations (although it is currently not used by any existing plugins), so few changes are required to support that capability.

The following diagram shows an overview of the HDF5 library architecture before the EFF capabilities are added:



The following diagram shows an overview of the HDF5 library architecture after the EFF capabilities are added, with the new or enhanced portions highlighted:



The majority of the implementation work is localized to the EFF wrapper routines and the IOD VOL plugin. In particular, the end-to-end integrity checksums are created and validated in the IOD plugin, and data layout information is translated from HDF5 properties to IOD hints there as well. Transactions and asynchronous operation information is encapsulated in HDF5 properties by the EFF wrapper routines and retrieved, interpreted and returned by the IOD plugin in the same way. Details of the IOD VOL plugin design are located in an accompanying document.

Storing HDF5 Objects in IOD Containers

Objects in the HDF5 data model and operations on them are mapped to IOD objects and operations, as they are handled by the IOD VOL plugin. See section 4.3.2 in the IOD design document for a description of the mapping from HDF5 objects to IOD objects and the accompanying IOD VOL plugin design document for a description of how those mappings are carried out.

API and Protocol Additions and Changes

There are two kinds of changes to the HDF5 library API: generic changes to existing API routines that accommodate new capabilities, such as asynchronous I/O and transactions, and additions to the HDF5 API which add new features. Both of these types of changes to the HDF5 API are described below.

Generic changes to HDF5 API routines

Many HDF5 API routines operate on HDF5 file objects and need to be extended in the same way. Rather than describing each of the modified HDF5 API routines, a generic modification is described below, along with a list of HDF5 API routines that are affected.

Existing HDF5 routines that operate on HDF5 file objects are extended by adding two new parameters: a transaction number and a pointer to an asynchronous operation request object. Additionally, HDF5 API routines that are extended in this manner have a suffix appended to the routine name, to distinguish these routines from existing routines. The following pseudo-function prototypes describe the method for these changes to HDF5 API routines:

Current routine:

```
<return type> H5Xexisting_routine(<current parameters>);
```

Extended routine:

```
<return type> H5Xexisting_routine_ff(<current parameters>,

    uint64_t transaction_number, H5_request_t *request_ptr);
```

In other words, each extended HDF5 API routine has a suffix of "_ff"[1] added to the API's routine name and two new parameters added to its parameter list: a transaction number, which indicates the transaction this operation is part of, and a pointer to a request object, for testing/waiting on the asynchronous completion of the operation. Passing a NULL pointer for the request object pointer value indicates that an operation should complete synchronously.

[1] "ff" is short for "FastForward"

As a concrete example, the following prototypes show the change to the group creation API routine for HDF5, H5Gcreate[2]:

Current routine:

```
hid_t H5Gcreate(hid_t loc_id, const char *name, hid_t lcpl_id,

    hid_t gcpl_id, hid_t gapl_id);
```

Extended routine:

```
hid_t H5Gcreate_ff(hid_t loc_id, const char *name, hid_t lcpl_id,

    hid_t gcpl_id, hid_t gapl_id, uint64_t transaction_number,

    H5_request_t *request_ptr);
```

Note that the error value returned when a routine is invoked asynchronously only indicates the status of the routine up to the point when it is scheduled for later completion. The asynchronous test and wait routines (below) return the error status for the "second half" of the routine's execution.

We anticipate that if the features from the FastForward project are productized in a future public release of HDF5, the "_ff" suffix will be removed and affected API routines will be versioned according to the standard convention for modifying HDF5 API routines[3].

A note on the design of the API changes: We considered alternate forms of passing the transaction and request information into and out of the HDF5 API routines, such as using HDF5 properties in one of the property lists passed in to API routines to convey the information. Using HDF5 properties had a number of drawbacks however: (1) several of the API routines did not have property list parameters and so would have to be extended with more parameters anyway, (2) setting the additional information in properties can sometimes obscure the fact that an operation's behavior has been changed, and (3) it is particularly tedious for application developers to retrieve the asynchronous request object from a property list after each API call.

The following is a list of all HDF5 API routines extended in the manner described above:

HDF5 Attribute Routines:

- H5Acreate
- H5Acreate_by_name
- H5Adelete
- H5Adelete_by_name
- H5Adelete_by_idx
- H5Aexists
- H5Aexists_by_name
- H5Aget_info_by_idx
- H5Aget_info_by_name
- H5Aget_name_by_idx
- H5Aiterate
- H5Aiterate_by_name
- H5Aopen
- H5Aopen_by_idx
- H5Aopen_by_name
- H5Aread
- H5Arename
- H5Arename_by_name
- H5Awrite

[2] http://www.hdfgroup.org/HDF5/doc/RM/RM_H5G.html#Group-Create2

[3] HDF5's API versioning conventions are described here: http://www.hdfgroup.org/HDF5/doc/RM/APICompatMacros.html

HDF5 Dataset Routines:

- H5Dcreate
- H5Dcreate_anon
- H5Dopen
- H5Dread
- H5Dset_extent
- H5Dwrite

HDF5 Group Routines:

- H5Gcreate
- H5Gcreate_anon
- H5Gget_info_by_idx
- H5Gget_info_by_name
- H5Gopen

HDF5 Link Routines:

- H5Lcopy
- H5Lcreate_external
- H5Lcreate_hard
- H5Lcreate_soft
- H5Lcreate_ud
- H5Ldelete
- H5Ldelete_by_idx
- H5Lexists
- H5Lget_info
- H5Lget_info_by_idx
- H5Lget_name_by_idx
- H5Lget_val
- H5Lget_val_by_idx
- H5Literate
- H5Literate_by_name
- H5Lmove
- H5Lvisit
- H5Lvisit_by_name

HDF5 Object Routines:

- H5Ocopy
- H5Odecr_refcount
- H5Oexists_by_name
- H5Oget_comment
- H5Oget_comment_by_name
- H5Oget_info
- H5Oget_info_by_idx
- H5Oget_info_by_name
- H5Oincr_refcount
- H5Olink
- H5Oopen
- H5Oopen_by_idx
- H5Ovisit
- H5Ovisit_by_name

HDF5 Datatype Routines:

- H5Tcommit
- H5Tcommit_anon
- H5Topen

Additions to the HDF5 API

The following routines will be added to the HDF5 API to support the new capabilities in the library.

Asynchronous Operations:

Note: The test and wait operations below can be expanded with MPI-like testall/waitall and/or testany/waitany variants as needed.

H5AOtest() – Test if an asynchronous operation has completed:

```
herr_t H5AOtest(H5_request_t *request_ptr, H5_status_t *status_ptr);
```

Calling H5AOtest will determine if an asynchronous operation has completed, and return the operation's status to the application. Possible values returned for the operation's status are:

- H5AO_PENDING – The operation has not yet completed

- H5AO_SUCCEEDED – The operation completed successfully
- H5AO_FAILED – The operation has completed, but failed

Once an asynchronous operation has completed (successfully or not), the request object becomes is invalid for future test/wall calls.

The return value from H5AOtest is negative on failure and non-negative on success.

H5AOwait() – Wait for an asynchronous operation to complete:

```
herr_t H5AOwait(H5_request_t *request_ptr, H5_status_t *status_ptr);
```

Calling H5AOwait waits for an asynchronous operation to complete, returning the operation's status to the application. Possible values returned for the operation's status are:

- H5AO_SUCCEEDED – The operation completed successfully
- H5AO_FAILED – The operation has completed, but failed

Once an asynchronous operation has completed (successfully or not), the request object becomes is invalid for future test/wall calls.

The return value from H5AOwait is negative on failure and non-negative on success.

End-to-End Integrity:

H5Pset_data_checksum() – Set a checksum for data buffer in application memory:

```
herr_t H5Pset_data_checksum(hid_t dxpl_id, H5_checksum_t chksum);
```

H5Pset_data_checksum sets a checksum value property on a dataset access property list for the library to use in future calls to H5Dwrite. The HDF5 library will verify that the application data generates the same checksum before copying data to internal buffers. If no copy to internal buffers is necessary, the application's checksum value will be passed directly to the IOD layer.

Note: the checksum passed in must be generated with the HDF5 library's H5checksum routine (below). [Optionally, we could have the application give us function pointer to a routine that the HDF5 library can use for verifying the buffer's checksum, but this is slower when the buffer doesn't need to be copied, since the H5checksum routine must be used for passing checksums to IOD]

The return value from H5Pset_data_checksum is negative on failure and non-negative on success.

H5checksum() – Perform a checksum on a buffer:

```
H5_checksum_t H5checksum(const void *buffer, size_t size, H5_checksum_t
input_checksum);
```

Perform a checksum on a buffer, returning the value generated. Passing in a non-zero input checksum will use that value as the "seed" for the checksum's initial value, allowing a single checksum value to be generated for multiple buffers (possibly from a set of data elements scattered in memory). The checksum algorithm will produce the same result on a single contiguous buffer as on multiple separated buffers containing the same data values.

H5Pset_edc_check() – *Existing routine* – Enables/disables checksum verification on data element reads.

Transactions:

Note: The transaction routines below are incomplete and are included for a general sense of the routines being planned.

H5TRcreate() – Begin new transaction

H5TRabort() – Abort a transaction, abandoning all changes to the container made within the transaction

H5TRcomplete() – Complete a transaction, committing all changes made within the transaction to the container

H5TRpersist() – Persist a transaction's data from IOD to DAOS storage

H5TRremove() – Remove a transaction's data from the IOD storage

H5TRprefetch() – Prefetch a transaction's data from DAOS to IOD storage

H5TRcreate_snapshot() – Create a container snapshot. (Opening a container snapshot is specified as a property on the file open operation, described below)

H5TRremove_snapshot() – Remove a snapshot from a container

Data Layout Properties:

H5Pset_layout() – *Existing routine* – Choose chunked or contiguous layout for dataset storage. This property will be translated to an IOD hint when the dataset is created in the IOD/DAOS container.

Library Instructure:

EFF_init() – Initialize the Exascale FastForward storage stack:

```
int EFF_init(MPI_Comm comm, MPI_Info info, const char *fs_driver, const
char *fs_info);
```

Must be called by an application before any HDF5/IOD/DAOS API calls are made. The MPI communicator and info objects are used to set aside the IONs from the CNs and set up communication channels between each CN and an ION. The fs_driver and fs_info parameters choose the network driver to use for function shipper communications and pass configuration information to that driver, respectively.

The return value from EFF_init is negative on failure and non-negative on success.

File Objects/Properties:

H5Pset_fapl_vol_iod() – Use the IOD VOL plugin for container operations:

```
herr_t H5Pset_fapl_vol_iod(hid_t fapl_id, MPI_Comm comm, MPI_Info info);
```

Calling H5Pset_fapl_vol_iod will cause the HDF5 library to use the IOD VOL plugin for accessing the HDF5 container object (as opposed to the native HDF5 file format, or another storage/access mechanism). The communicator and info parameters are used to set up communication channels for collective operations on the HDF5 container.

Calling this routine is *mandatory* to use the HDF5 API capabilities described in this document.

The return value from H5Pset_fapl_vol_iod is negative on failure and non-negative on success.

H5Pset_eff_snapshot() – Set snapshot to use when opening a container:

```
herr_t H5Pset_eff_snapshot(hid_t fapl_id, uint64_t snapshot_value);
```

Calling H5Pset_eff_snapshot will set a container snapshot value in the file access property list, to use when opening the HDF5 container, instead of the default action of accessing the latest consistent version of the container.

The return value from H5Pset_eff_snapshot is negative on failure and non-negative on success.

Dataset Objects:

None yet

Group Objects:

None yet

Named Datatype Objects:

None yet

Attribute Objects:

None yet

Link Objects:

None yet

Open Issues

Some of the existing HDF5 routines that are extended above don't need a transaction ID (e.g. routines which only read information from the container, like H5Lexists) and so might need to be modified differently (they would be "generically" modified to take an asynchronous request object, but not a transaction number parameter).

During our internal design discussions, we have considered having a mechanism for tagging objects in some way so that they are prefetched/persisted/removed together. It also seems more likely that an application would want to prefetch/persist/remove objects at the IOD layer instead of transactions. We are considering use cases for these behaviors and may include them in the full design for transactions, next quarter.

Risks & Unknowns

As the changes to the HDF5 library are dependent on capabilities added to multiple lower layers of the software stack (the function shipper, IOD and DAOS layers), it is likely that changes at those layers will ripple up through the HDF5 API and cause additional work at this layer. On the other hand, we can always mitigate the effect of changes at lower levels by abstracting those capabilities and implementing support within the HDF5 library for features missing or different below it.

Conversely, the demands of the applications that use the HDF5 API may pull the features and interface in unexpected directions as well, in order to provide the necessary capabilities for the application to efficiently and effectively store its data. These two forces must be balanced over the course of the project, hopefully producing a high quality storage stack that is useful to applications at the exascale.