Summary of HACC with HDF5 and the Fast Forward Storage Stack

M. Scot Breitenfeld[1] and Quincey Koziol[2]

[1] brtnfld@hdfgroup.org, [2]koziol@hdfgroup.org

Nomenclature

BB	Burst Buffer
CN	Compute Node
DAOS	Distributed Application Object Storage
FF2	Exascale FastForward
IOD	I/O Dispatcher
ION	I/O Node
VDS	Virtual DataSet
AIO	Asynchronous I/O

I. Introduction to HACC I/O

HACC (Hardware/Hybrid Accelerated Cosmology Code) is a N-body cosmology code framework where a typical simulation of the universe demands extreme scale simulation capabilities. However, a full simulation of HACC requires terabytes of storage and hundreds of thousands of processors, far exceeding the computational resources available in the current FF2 project. Consequently, a smaller benchmark I/O code (*GenericIO* by Hal Finkel) was created which mirrors the I/O calls in HACC without the need to run an entire simulation (http://trac.alcf.anl.gov/projects/genericio). In the benchmark, all the "heavyweight" data is handled using POSIX I/O, and the "lightweight" data is handled using collective MPI-IO.

Critical features for HACC's I/O include:

- Resiliency for data verification,
  - Checksumming from the application's memory to the file and vice-versa,
  - Mechanism for retrying I/O operations.
- Sub-filing,
  - Should avoid penalties in the file system associated with locking and contention.
- Self-describing file.

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

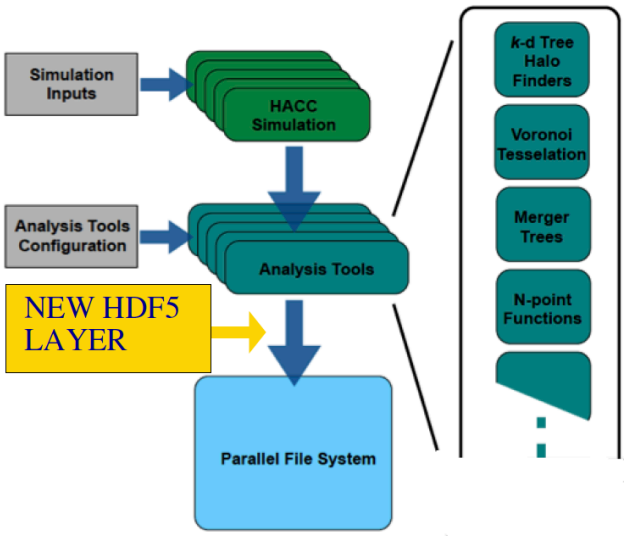

Figure 1 — HACC I/O scheme with HDF5 (hacc_pflops.pdf, 2013).

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

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

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

Table 1: GenericIO Performance (production runs on IBM Blue Gene/Q)

No. Particles	No. Processes	File size (GiB)	Write time (s)	Write Bandwidth (GiB/s)
1024^3	512	43.8	22.0	1.90
3200^3	16384	1332.4	99.0	12.88
10240^3	262144	43821.6	380.5	109.9

Table 1 — HACC/GenericIO performance on Mira [1].

II. Implementation of HACC with HDF5 on the FF2 stack

As mentioned in Section I, the current I/O implementation in HACC uses either POSIX or MPI-IO. Therefore, the first step in getting HACC onto the FF2 stack is to add HDF5 to HACC (see Section II.A). A github repository, https://github.com/brtnfld/genericio, was setup to maintain the HDF5 and FF2 additions to the original code mentioned in Section I.

II.A. Introduction of HDF5 into HACC's I/O Driver

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

The use of offsets as pointers to variables (note that these offsets are stored within the HACC file) is eliminated in the HDF5 implementation by instead using datasets to store variables. Currently there are nine particle variables used in GenericIO: *pid, x, y, z, vx, vy, vz, phi* and *mask*. The HDF5 implementation stores the variables as datasets in the file's root group (/), Fig. 2.

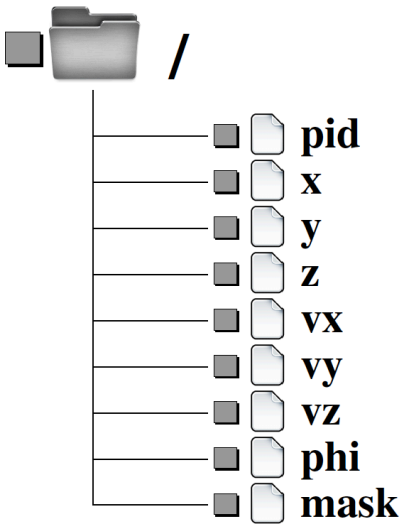

Figure 2 — Proposed HACC/HDF5 file structure.

Although data is checksummed automatically for FF2, that is not the case for HDF5 files by default. However, checksums can be easily implemented in HDF5 by simply computing a checksum for the array (assuming no partial writes are taking place) and writing the checksum as an attribute of the dataset. The reading program can then read the attribute and do a checksum for the array being read. Since HACC typically accesses the entire array when performing I/O, this is acceptable our purposes.

Additionally, simply using the HDF5 object interface's iteration and inquiry functions, such as H5Ovisit, can eliminate additional file metadata containing the number of variables and variable names in the current GenericIO file format.

II.B. HACC's HDF5 I/O Driver on the FF2 stack

A transaction in the FastForward storage stack consists of a set of updates to a container. Updates in the form of additions, deletions and modifications are added to a transaction and not made directly to a container. Once a transaction is committed, the updates in the transaction are applied atomically to the container.

The basic sequence of transaction operations an application typically performs on a container that is open for writing is:

1) *start* transaction N
2) add *updates* for container to transaction N
3) *finish* transaction N

One or more processes in the application can participate in a transaction, and there may be multiple transactions in progress on a container at any given time. Transactions are numbered, and the application is responsible for assigning transaction numbers while a container is open for write. Transactions can be finished in any order, but they are committed in strict numerical sequence. The application controls when a transaction is committed through its assignment of transaction numbers in "create transaction / start transaction" calls and the order in which transactions are finished, aborted, or explicitly skipped.

The ***version*** of the container after transaction N has been committed is N. An application reading this version of the container will see the results from all committed transactions up through and including N.

The application can ***persist*** a container version, N, causing the data (and metadata) for the container contents that are in the BB to be copied to DAOS and atomically committed to persistent storage.

The application can request a ***snapshot*** of a container version that has been persisted to DAOS. This makes a permanent entry in the namespace (using a name supplied by the application) that can be used to access that version of the container. The snapshot is independent of further changes to the original container and behaves like any other container from this point forward. It can be opened for write and updated via the transaction mechanism (without affecting the contents of the original container), it can be read, and it can be deleted.

Using transactions allows HACC to push large amounts of finished *in situ* analyzed data to disk while an application reader is concurrently accessing the data in the HDF5 files.

Furthermore, HACC does not currently handle Asynchronous I/O (AIO) operations, but there is interest in using AIO in FF2 for HACC. AIO can improve performance in applications by

overlapping I/O requests with computational calculations or post-processing previously completed I/O. With AIO, a task can initiate a number of I/O operations without having to block or wait for them to complete. The typical flow of non-blocking AIO is depicted in Figure 3. The application initiates a read request and returns immediately. The application then continues to perform computations or additional read requests while the past read requests are completed in the background. Once the read response is received, the application can then complete the I/O transaction.

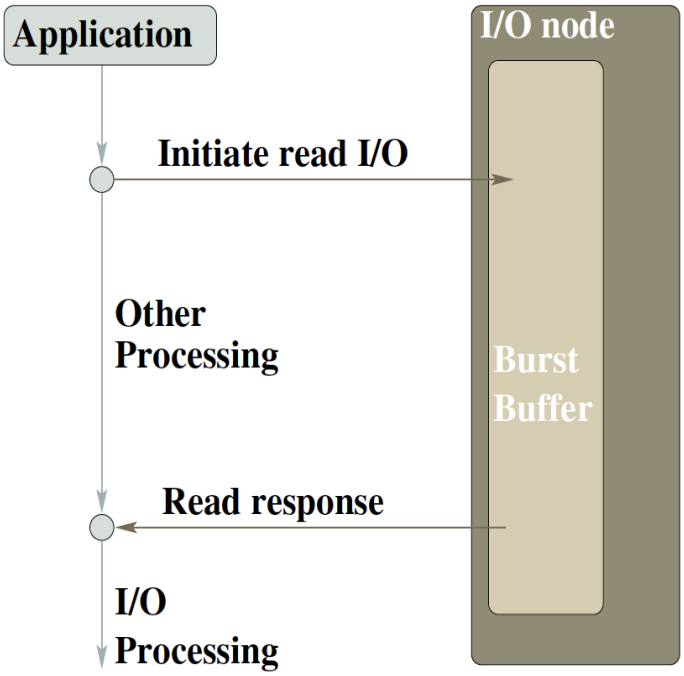

Figure 3 — Flow of non-blocking asynchronous I/O (AIO).

Support for AIO in HDF5 is implemented by:

1) Building a description of the asynchronous operation,
2) Shipping that description from a Compute node to an I/O node for execution,
3) Generating a request object and inserting it into an Event Stack object that the application provides, while the operation completes on the I/O node.

An Event Stack provides an organizing structure for managing and monitoring the status of operations that have been invoked asynchronously. Various Event Stack object APIs allow for the management of the Event Stack, completion status, and error information for the asynchronous I/O. Note that it is mandatory that the application must not deallocate or modify data element buffers used in asynchronous operations until the asynchronous operation has completed. Therefore, the application code must finalize the data by staging the I/O tasks appropriately before initiating AIO [3]. Since HACC does a large amount of *in situ* data analysis

before writing to disk, it would be useful for HACC to overlap the data analysis with writing the finished analysis quantities to disk.

Bibliography

[1] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, Venkatram Vishwanath, Zarija Lukic, Saba Sehrish, Wei-keng Liao. "HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures" New Astronomy, Volume 42, January 2016, Pages 49–65.

[2] "Design and Implementation of the FastForward Features in HDF5: FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O". Excerpts from The HDF Group.