Date:
21 December 2015

Summary of HACC with HDF5 and the Fast Forward Storage Stack

Extreme Scale Storage and I/O RND

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

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

NOTICES

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

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.

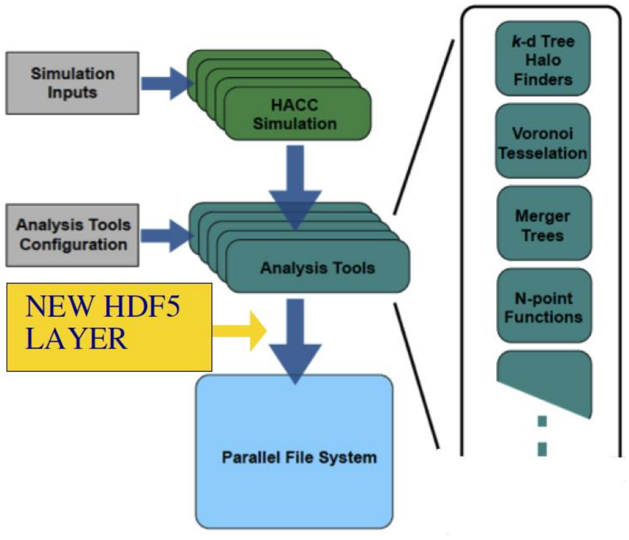


Figure 1 — HACC I/O scheme with HDF5 (hacc_pflops.pdf, 2013).

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

The default I/O strategy in HACC is to have each process write data into a distinct region within in a single file using 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 cyclic redundancy code (CRC) 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 FF 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 FF stack is to add HDF5 to HACC's GenericIO framework (see Section II.A). A github repository, https://github.com/brtnfld/genericio, was setup to maintain the HDF5 and FF additions to the original code mentioned in Section I. The HDF5 implementation (without the FF additions) is located in the "*hdf*" branch and the HDF5 + FF implementation is in the "*EFF*" branch. The original master code provided by the HACC team is located in the "master" branch. The code consists of two programs, a program for writing (*GenericIOBenchmarkWrite.cxx*) and a program for reading (*GenericIOBenchmarkRead.cxx*). The program *GenericIO.cxx* contains generic functions to handle either reading or writing the data in POSIX, MPI-IO, or HDF5.

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 automatically be 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 programs: *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.

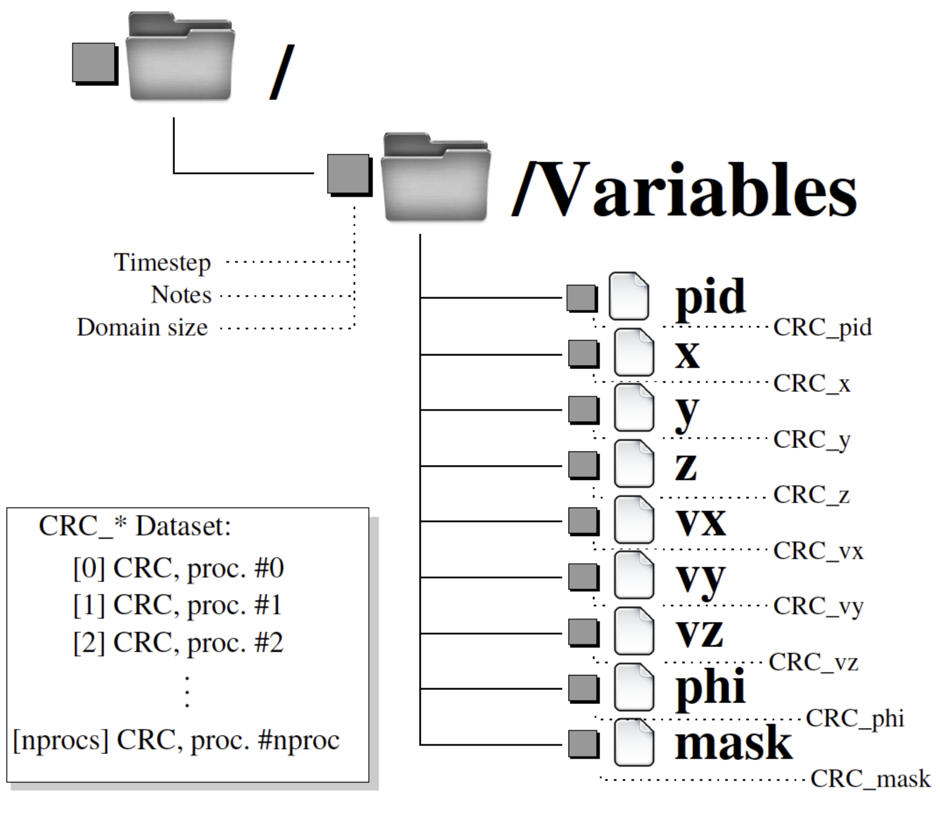


Figure 2 — Proposed HACC/HDF5 file structure.

Attributes at the root level store time step information, such as the time stamp and notes about the simulation parameters. Additional attributes can easily be added if needed. All the datasets are stored under the '*variable'* group (nine total for this demonstration).

Associated with each variable's dataset is the cyclic-redundancy check (CRC). The CRC uses the High-Performance CRC64 Library from Argonne National Laboratory. A CRC is computed for each variable, and each processor computes the CRC for the portion of the array residing on that process. Although the FF stack automatically performs a checksum from the ION to the disk, this is not the case for HDF5 files by default. However, as mentioned earlier, the CRC can easily be implemented within the HDF5 file by simply computing a CRC for the array (assuming no partial writes are taking place) and writing the CRC as a dataset. The reading program can then read the dataset, compute the CRC for the read data and perform a comparison to the values stored in the CRC dataset. The implied restriction is that the layout of the array among the processors is the same for both the writing and the reading of the arrays. Insuring a matching CRC for data written and data being read is important when creating raw binary files because the file can be transferred to a machine with a different endianness. Therefore, checks have to be made to ensure the endianness conversions were implemented correctly. In HDF5 however, the library will convert and verify the byte-order automatically, so the use of the CRC may no longer be necessary. Additionally, using HDF5 object interface's iteration and inquiry functions, such as H5Ovisit, can eliminate additional file metadata currently stored in the raw GenericIO file format. For the demonstration, the code had a set number of variables, so these inquiry functions were not used, but can be added in the future.

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.

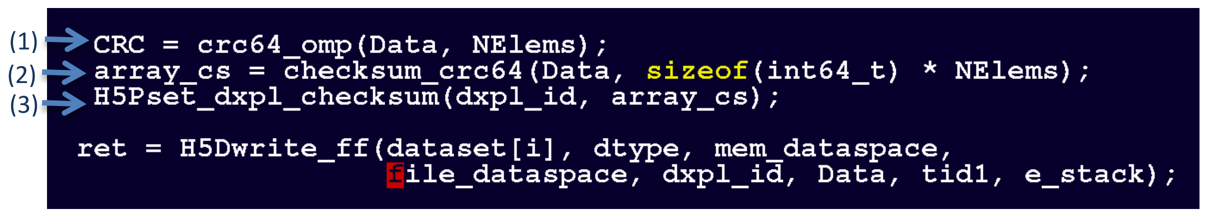
The ***version*** of the container after transaction N has been committed is N. An application reading this version of the container will see the results from all committed transactions up through and including N.

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.

Summary of the Demonstration on December 17th, 2015

The key objectives of the demonstration were to show (1) HACC running on the FF stack with HDF5, and (2) complete checking of the data integrity, including writing from the ION layer, all the way to reading in the array with a separate program. The CRC check was implemented according to the convention summarized in “Section II.A. Introduction of HDF5 into HACC’s I/O Driver”, line (1) in the figure below. The checksum, line (2), from the ION to the server was handled by the data transfer property list, *dxpl*, set by *H5Pset_dxpl_checksum*. The *dxpl* was then used by H5Dwrite_ff to validate the writes from the ION to the disk.



```
CRC = crc64_omp(Data, NElems);
array_cs = checksum_crc64(Data, sizeof(int64_t) * NElems);
H5Pset_dxpl_checksum(dxpl_id, array_cs);

ret = H5Dwrite_ff(dataset[i], dtype, mem_dataspace,
                  file_dataspace, dxpl_id, Data, tid1, e_stack);
```

The first demonstration ran on two processors and was for a problem size of 128 elements. The write program, *GenericIOBenchmarkWrite.cxx,* was run first and verified the checksum of the stack during the writing process, it then wrote the CRC values to datasets. The second program, *GenericIOBenchmarkRead.cxx*, read the HDF5 file and verified the correctness of the CRC. Both tests passed.

The second demonstration again ran on two processors and was for a problem size of 128 elements. However, HDF5 disabled data integrity checks stored at the IOD for the write in order to test if the transfer checksum would capture the corruption. The corruption was injected for the "*x*" variable dataset, and it was shown that the server could detect the corruption.

A retry after a data corruption was detected was not demonstrated simply because the retry implementation will happen at the new DAOS layer. The HDF5 layer can simulate a “pseudo” error and retry at the DAOS layer, but this would just use an HDF5 property to simulate a retry at the IOD+DAOS layer and would not be a true demonstration of the error handling at the DAOS layer. The retry requirement should be demonstrated in the new DAOS layer. Furthermore, HACC/HDF5 should not have to be concerned with error correction and retry as those functions should be transparent to the application.

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.