Parallel IO Libraries for Managing HEP Experimental Data

Amit Bashyal[1,*], *Christopher* Jones[2], *Kyle* Knoepfel[2], *Peter* Van Gemmeren[1], *Saba* Sehrish[2], and *Suren* Byna[3,4]

[1]Argonne National Laboratory
[2]Fermi National Accelerator Laboratory
[3]Lawrence Berkeley National Laboratory
[4]The Ohio State University

Abstract. The computing and storage requirements of the energy and intensity frontiers will grow significantly during the Run 4 & 5 and the HL-LHC era. Similarly, in the intensity frontier, with larger trigger readouts during supernovae explosions, the Deep Underground Neutrino Experiment (DUNE) will have unique computing challenges that could be addressed by the use of parallel and accelerated data-processing capabilities. Most of the requirements of the energy and intensity frontier experiments rely on increasing the role of high performance computing (HPC) in the HEP community. In this presentation, we will describe our ongoing efforts that are focused on using HPC resources for the next generation HEP experiments. The HEP-CCE (High Energy Physics-Center for Computational Excellence) IOS (Input/Output and Storage) group has been developing approaches to map HEP data to the HDF5 , an IO library optimized for the HPC platforms to store the intermediate HEP data. The complex HEP data products are serialized using ROOT to allow for experiment independent general mapping approaches of the HEP data to the HDF5 format. The mapping approaches can be optimized for high performance parallel IO. Similarly, simpler data can be directly mapped into the HDF5, which can also be suitable for offloading into the GPUs directly. We will present our works on both complex and simple data model models.

1 Introduction

`ROOT` has been the main storage format for the HEP experiments with major HEP experiments using it to store their data [1]. The `ROOT` framework not only provides computational tools for HEP data analysis, but also provides effective IO operations in the High Throughput Computing (HTC) environment. However, projections made by ATLAS [2] and CMS [3] shows that the current computing infrastructure alone cannot fulfill the requirements of these experiments. Similarly, with a large far detector (four 10 kT liquid argon modules), DUNE will have data rates, although fewer than that of LHC experiments, in the order of few GBs (beam induced) to 100s of terabytes (supernova readouts).

*e-mail: abashyal@anl.gov

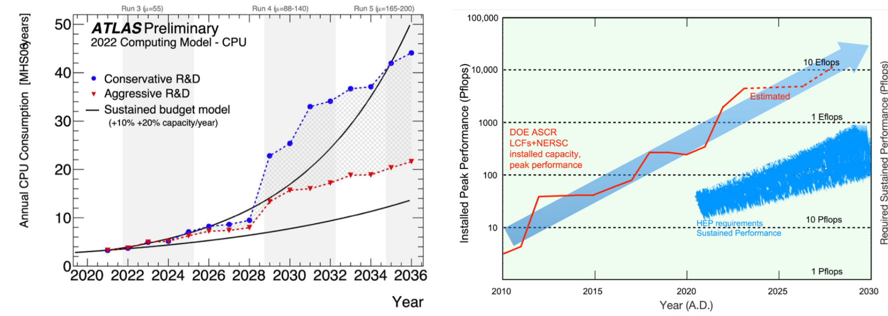

Figure 1. Left: The Projected CPU consumption rate for the ATLAS experiment from 2023 to 2036. The area between the conservative(blue) and aggressive (red) dot lines shows the range of CPU consumption based on conservating and aggressive R&D in the computing infrastructure of the ATLAS experiment. The top (bottom) black line shows the impact of 10% (20%) increase in R&D budget annual on the CPU consumption. Figure is taken from [2]. Right:Projection of peak performance of various DOE supercomputers (red line) is much higher than the HEP CPU resource requirements (blue shade). Figure is taken from [4].

As shown in figure 1, the High Performance Computers (HPC) could help to fulfill some of the computing needs of the HEP experiments. However, HPC systems use parallel file system to enable efficient data storage and access for faster IO. High-level libraries like HDF5 [5, 6], provide interface to use parallel file systems through their parallel IO libraries which are optimized for the HPCs. To utilize the HPC resources, HEP experiments need to utilize these high level libraries. Since `ROOT` provides all major tools for computing and analysis of HEP data, experiments will depend upon the `ROOT` framework for all their major computing needs. Our goal for this work is to study HDF5 in ways that can complement the existing use of `ROOT`.

The HEP-CCE IOS group studied the scaling and performance of the HDF5 libraries as an intermediate storage for HEP applications that have a need of parallel random access to the data in a single large file. Many HEP application frameworks are either multi-threaded (art, CMSSW)[7] or use multiprocessing (Athena, ATLAS)[8]. While HDF5's support for multi-threaded application is primitive, their support for multiprocessing parallel IO is well studied and optimized, especially for HPC.

In this report, we will describe our use of HDF5 that can be used for any of the experiments' frameworks mentioned above. The HEP-CCE IOS group developed an IO test framework to study the scaling of the IO performance with HDF5, `ROOT`, and a few other custom data formats.

Parameters like file size, memory usage, compression, computing time etc were taken into account to evaluate the scaling performance. Performance for both the serial and parallel IO operations were evaluated in the case of the HDF5.

1.1 Description of the Test Framework

The test framework [9] is hosted by the HEP-CCE `github` repository. The framework is experiment agnostic but supports reading data of major experiments like DUNE, CMS and ATLAS for testing. Multi-threading operations are supported by `tbb` libraries. The main testing program is called `threaded_io_test` and the instructions to build and run are given in the `README` file. The framework can either generate fake

data and write into the user specified format or take data from one of the supported source data and write into the user specified output storage system. If the source data is experiment specific, then the user needs to provide the appropriate `ROOT` object serialization dictionaries to read and write the data. Parallel IO is only supported for the HDF5 format with the HDF5 native collective IO and MPI calls to send and receive data and message during the parallel operations.
Although the framework can support the IO testing with data written in `ROOT`, HDF5 and a CCE specific testing format called Packed Data Stream (`PDS`), this report will primarily focus on the tests done with data saved in HDF5 format.
In the HEP framework paradigm, data associated with the same particle collision/interaction within the detector (or from simulation) is grouped in an `Event`. Data within an `Event` is further grouped into `Data Products`. These `Data Products` can include particles created from the interaction, energy deposition by the particles, location of interactions etc. These attributes help to reconstruct the physical quantities like particle identification, momentum, energy etc. Since most of the HEP experiments design their data to utilize resources provided by the `ROOT` framework, their data have similar structure such that different `Data Products` and metadata of each events are stored individually (usually as `ROOT::TBranch`). A `Data Product` could be stored as fundamental types (`int, chars, double, float, arrays etc`) or complicated C++ objects.

HEP frameworks are designed to schedule algorithms where each algorithm processes data from an `Event`. The output of one algorithm may be used as the input to another algorithm. Multi-threaded HEP frameworks typically allow concurrent processing of algorithms where the algorithms are processing data from different Events. The frameworks may also allow concurrent processing of different algorithms within a single `Event` (still enforcing dependency requirements between algorithms). One threading performance bottleneck in HEP frameworks is in the calls to the `ROOT` IO APIs which must be done serially.

The CCE test framework mimics the multi-threaded behavior of the CMS software framework. The test framework processes multiple `Events` concurrently (where the number of concurrently processed events is set at configuration time). In addition, all `Data Products` within the `Events` can be concurrently processed. Therefore it is possible to have all `Data Products` for all available `Events` having C++ object serialization being run concurrently. The framework also allows concurrent writing to external storage (if a format is used that supports it) else it has an efficient non-blocking queuing system to serialize parts of an algorithm which can not be run concurrently. The framework supports generating `Events` and writing them in multiple formats and allows the extension of additional formats using C++ polymorphism. A new input or output algorithm can be added to the framework by using C++ inheritance. The framework currently supports reading and writing `Events` in these algorithms include ones which can read and write `ROOT` and file formats with various multi-threading optimizations. For parallel HDF5 IO, the test framework supports multi-processing using MPI.

2 HDF5 Design approaches

Typically HEP data is organized in several levels of aggregations; runs, subruns (also called a luminosity block), `Events`, and `Data Products`. An `Event` usually consists of collections of `Data Products` that describes the various attributes of the data. To keep our design approach independent of experiment specific `Data Products` and

assumptions, we use a simplified organization of Data Products. The Data Products are serialized into a byte stream using ROOT before we map them to the HDF5 data model. The framework supports the concurrent serialization of the objects until writing them into HDF5 file which is only supported sequentially.

The HDF5 concepts that we used are HDF5::DataSets, HDF5::Groups, and HDF5::Attributes. The data from a subrun is stored in one HDF5::Group. Information about run, subrun, compression, etc are written as HDF5::Attributes. Currently the framework supports two different ways of writing HEP data into HDF5 format.

2.1 Data Products based approach

In the Data Product based approach, each kind of Data Product in an Event is represented by two HDF5::DataSets. One HDF5::DataSet is used to store the serialized byte stream of Data Product values and the other one is used to store the offset describing the event boundaries for a specific data product. Since most of the HEP data is stored in ROOT with various data products of an event in different ROOT::TBranches, HDF5::DataSet based approach is a one to one mapping of the data stored in ROOT and HDF5 where each ROOT::TBrach object is mapped to corresponding HDF5::DataSet. An additional HDF5::DataSet for each Data Product is created to store the size of serialized Data Product values. There is one additional HDF5::DataSet to store the Event identifiers(IDs). The size of the Event IDs HDF5::DataSetis equal to the number of Events. Hence, in this approach, n Data Products are mapped to $2n + 1$ HDF5::DataSets. In HEP experimental data, usually an Event may have thousands of data products. This approach requires creating thousands of HDF5::DataSet's, and large metadata to map the HEP data. Tests done with this approach did not show good performance due to large number of write operations per HDF5::DataSets across all HDF5::DataSets. However, the data stored with this method provides flexibility and allows users to explore the data products individually without having to read the entire Event. All the Events in a subrun are stored in a single HDF5::Group. Information like run number, subrun number, compression levels, *etc* are stored as HDF5::Attributes.

2.2 Event based approach

The second approach is optimized for the cases when an entire Event is accessed for processing. As described in section 2.1, each Event is a collection of data products of different types. Instead of creating one HDF5::DataSet per data product type, only one HDF5::DataSet is used to store all the HDF5::DataSets of the Events. We used HDF5::DataSet with type char to store the Data Products that have already been serialized using ROOT. There is additional information needed to describe begin and end of each Event, which is stored in another HDF5::DataSet with type uint32_t. The final HDF5::DataSet is used to store event identifier. As a result, this approach consists of three HDF5::DataSets; one to store events IDs, one to store Events and one to store the location of each Event in the HDF5::DataSet that stores Events. The information regarding different classes that represent different data products is stored as HDF5::Attribute. The run number, subrun number, compression algorithm and other information are all stored as attributes of the group that has all the HDF5::DataSets stored. Additionally, the class name of each Data Products are saved as HDF5::Attribute as well. Using one HDF5::DataSet for all the events

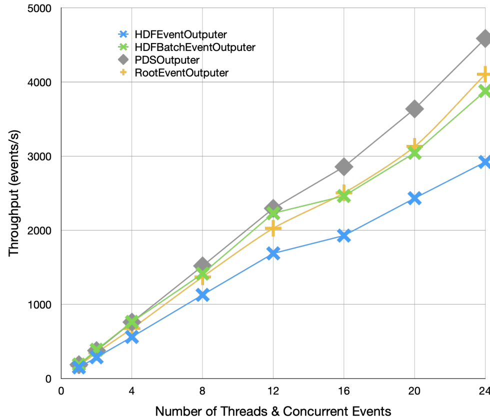

Figure 2. IO performance of the framework in various output modes. Figure shows the total throughput when writing events into various formats. `HDFEventOutputer` and `HDFBatchEventOutputer` are explained in section 2.2. `PDSOutputer` writes the data in packed data stream format and `ROOTEventOutputer` in the `ROOT` format respectively.

significantly reduces the cost to create thousands of `HDF5::DataSets` and metadata for those `HDF5::DataSets`, but adds thousands of `HDF5::Attributes`. We aggregate `Events` into batches before writing to the file. The number of `Events` in a batch is determined by how many `Events` can be processed and written with in memory limit on a given node. `Event` batching reduces the number of times a `HDF5::DataSet` needs to be extended and the number of write calls to the file system as compared with calling these functions per `Event`-wise. Once a batch of `Events` is ready to be written to an HDF5 file, we use the same sequence of HDF5 functions to do actual writes as described in previous subsection. The writing process is simplified with `Event` based and batching saves the IO cost, however, retrieving data products after they are read as an `Event` blob is non-trivial.

Besides HDF5, the framework also supports output modules in various formats. IO performance of these modules are shown in Figure 2. Figure 2 shows comparable throughput obtained by various formats using the same thread count for processing and same amount of data for writing sequentially.

3 Parallel HDF5 IO Design

In addition to using serial HDF5 and understanding its usability and performance in the HEP application space, we also worked on designing and implementing output modules with parallel HDF5. Parallel HDF5 allows multiple processes to write to one file. It eliminates the need of concatenating thousands of files that are generated at the end of processing via multiple independent grid nodes.

Parallel HDF5 uses MPI (Message Parsing Interface) -IO, a high performance parallel IO library. MPI-IO supports independent IO and collective IO. Independent IO means that each process does IO independently, while collective IO requires all processes to participate when doing IO operations. The advantage of collective IO is that it allows MPI-IO to do optimization to improve IO performance. We used collective IO approach because different processes that need to process a set of `Events`

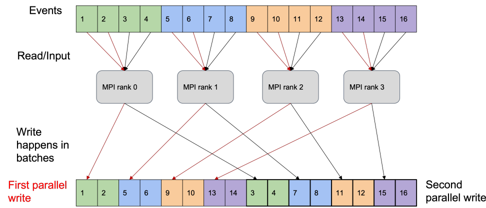

Figure 3. Schematic diagram of the Event Batch based IO design to write data into the output HDF5 File. Here 4 MPI ranks (parallel processes) are writing 16 events into the HDF5 file. Each MPI rank writes a total of 4 events with 2 events in each iteration. Hence in 2 iterations, 4 MPI ranks write 16 events into the HDF5 file. Note that the events are serialized before writing into the HDF5 file.

to be written to a file needs to synchronize with all the other processes to be able to know where to write in a particular `HDF5::DataSet`. The parallel IO design uses `Event` based approach with batching as described in 2.2.

In the parallel version, each MPI rank processes batches of events and collectively writes one batch per iteration to a single output file. The `Event` distribution algorithm assumes that the number of events to process, the number of MPI ranks, and the batch size per rank are known. To simplify the distribution problem, the first step is to round the number of events to the nearest number divisible by the number of MPI ranks to process. Then divide the number of events per MPI rank into batches. This approach guarantees that when using collective operations like `MPI_Scan`, all MPI ranks are participating in each iteration of write. In each iteration of write, all of the participating MPI ranks communicate the offsets and size of data to be written with each other. We used `MPI_Scan` to calculate partial sum of the data size that each rank has to write. Figure 3 shows an example; 4 MPI ranks are processing 16 events in total, and the batch size is 2. Since the number of events is divisible by the number of MPI ranks, no adjustment is needed. Then each rank determines independently that it will be writing 2 batches of events. Each rank uses `MPI_Scan` to determine the individual offsets to perform write operation. An `MPI_AllReduce` call followed by the scan operation determines the maximum value, which is 6. The `HDF5::DataSet` is extended by the maximum value plus the size of data to be written, which is 2 in this example.

One of the main assumptions of this design is that it relies on the number of events to be written is known in advance. We have also implemented an algorithm without using number of events as a known factor. In this approach, writing happens in multiple iterations, and is stopped when there are no events to process. In each iteration, each MPI rank reads the same number of events. Each MPI rank then takes a non-overlapping batch of events and processes it and writes it to the file. The processing stops when there are no events to read and process. Our initial test results show that this approach does not perform as well as our initial approach of known events.

Our goal was to understand how well does the parallel HDF5 implementation scales. We used Cori at NERSC to run most of the tests presented in this study; Cori was a Cray XC40 with a peak performance of about 30 petaflops that retired on May 31, 2023.

We have already demonstrated that on a single node, we see comparable performance among different output approaches including `ROOT` and HDF5. For the parallel HDF5 tests, we use the configuration that gave the best performance with serial HDF5 writing. The number of threads per node was kept constant at 64 regardless of the number of MPI ranks on a given node. The compression level was also set to 6 to match what was used in the serial performance testing. The HDF5 chunk size for writing was set to 10MB. The number of events that were aggregated in a batch in each iteration before writing was set to 100 events. Our studies showed that using burst buffers on Cori always performed better than using the parallel file system, hence we have only included the tests with burst buffers in this report. The burst buffer was based on Cray DataWrap and used flash or SSD technology to improve the performance.

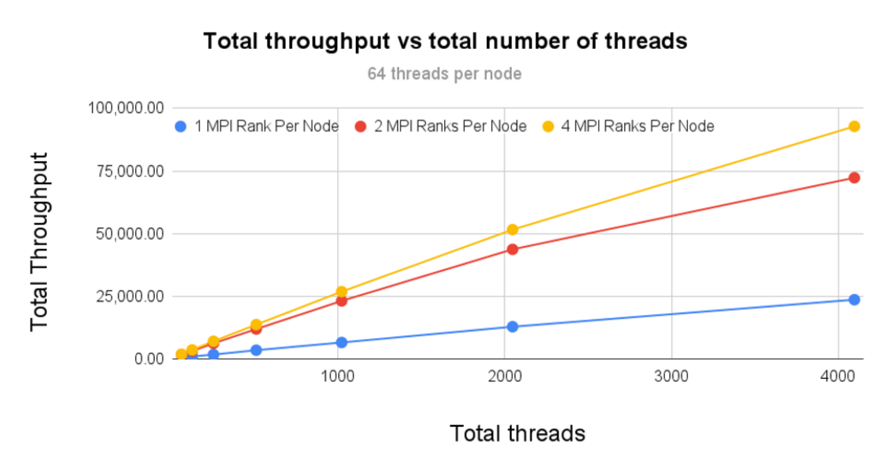

Figure 4. Scaling of Parallel HDF5 writing with multiple nodes on Cori (at NERSC). Threads per node is kept at 64.

Table 1 shows the average breakdown of the IO operations while writing events using the parallel HDF5 output module. According to the table, 54% of the IO calls of the framework falls in the `Others` category. It includes operations like metadata related operations, creating, opening and closing of required `H5Objects` like files, attributes, datasets etc. Some of the operations in this categories like metadata related operations could be optimized to furhter improve the performance.

Code stage	Percentage of total run time
MPI_Scan	9
MPI_AllReduce	5
HDF5::Write	32
Others	54

Table 1. Break down of the IO operations to execute the Event Batch based data model in the CORI machine. The "Others" category includes processes like creating input file, write headers, extension of data size etc.

4 Conclusion

We have developed a test framework that enables us not only to explore HDF5 IO designs in a generalized HEP computing framework but also allows to improve IO design for the HEP experiments in the HPC environment. The framework provides both serial and parallel IO testing of HEP `Events` in various input and output formats. Our studies shows that HEP experiments can leverage the IO libraries of high level libraries like HDF5 for parallel IO in the HPC environment. We have developed and tested multiple ways of mapping the HEP data to HDF5, access methods of the data and tuning of the HDF5 parameters for better parallel IO access.

5 Future Plans

In the first iteration of HEP-CCE, we have shown that using ROOT as a serialization tool, already serialized HEP data can be stored in HPC friendly format using the parallel libraries. However, the serialized data cannot be offloaded into current generation of HPCs which utilize GPGPUs to accelerate their IO. Data models need different organization to make them GPU friendly. Hence in the second iteration of the CCE, we will look into the study and design of the data models that can not only be stored in the HPC friendly format, but also can be offloaded into both host and device directly. This study will also complement another proposed study which aims to develop data model that will be friendly to `ROOT::RNTuple`, which is the latest IO subsystem provided for HEP experiments.

Acknowledgment

This work was supported by the U.S. Department of Energy, Office of Science, Office of High Energy Physics, High Energy Physics Center for Computational Excellence (HEP-CCE).

References

[1] A. Bashyal, P. Van Gemmeren, S. Sehrish, K. Knoepfel, S. Byna, Q. Kang (HEP-CCE IOS Group), *Data Storage for HEP Experiments in the Era of High-Performance Computing*, in *2022 Snowmass Summer Study* (2022), `2203.07885`

[2] Z. Marshall, J. Catmore, A. Calafiura, Paolo Di Girolamo, A. Collaboration, Tech. rep. (2022), `https://cds.cern.ch/record/2800627`

[3] O. Peckham, *CERN is betting big on Exa-Scale* (2022), `"https://www.hpcwire.com/2021/04/01/cern-is-betting-big-on-exascale/"`

[4] P. Calafuira, S. Habib, *HEP computational requirements on behalf of HEP-CCE* (2022)

[5] The HDF Group, *Hierarchical data format version 5* (2000-2010), `"http://www.hdfgroup.org/HDF5"`

[6] S. Byna, M. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal et al., JCST (2020)

[7] P. Elmer, B. Hegner, L. Sexton-Kennedy, *Experience with the CMS event data model*, in *Journal of Physics: Conference Series* (IOP Publishing, 2010), Vol. 219, p. 032022

[8] A. Collaboration, *Athena* (2021), `https://doi.org/10.5281/zenodo.4772550`

[9] C. Jones, S. Sehrish, *Root-serialization*, `https://github.com/hep-cce2/root_serialization/tree/master` (2021)