HDF5 as intermediate storage for HPC processing

Saba Sehrish for the CCE IOS team

HEP CCE All-hands Meeting Meeting April 2022

Goal

Evaluate if moving intermediate data (i.e. data between different processing steps) of an HEP workflow to a parallel file format (such as HDF5) could be beneficial for HPC processing?

Activities

- Development of a simplified HEP multi-threaded framework
- Design and implementation of output and input modules
  - ROOT-based, PDS, HDF5
- Evaluation and performance studies on Cori
  - Alternate layout design, other optimizations (e.g. compression)
- Introduction to parallel HDF5 design
  - Implementation and verification in progress
- Future plans and directions

Testing Framework Goal

- Test storage formats for use on HPC
- Able to compare
  - Read & write performance scaling with number of available cores
  - Memory usage scaling
  - File sizes
  - Large scale usage impact on HPC sites

Testing Framework

A simplified HEP multi-threaded framework based on CMS's framework is developed. The framework supports

- High levels of concurrency
  - Concurrent read/process/write of events as a whole
  - Concurrent read/write of individual data products within an event
  - Serial processes are scheduled by framework and never block a thread
- Any number of storage formats; can read from any supported format and write to any supported format
  - Just implement the necessary base class interfaces
- Experiment independent approach
  - Can read ATLAS, CMS and DUNE experiment ROOT files
  - allows testing using real world data

Different formats supported by the Framework

- Standard HEP ROOT format
  - Events stored as an element in a TTree
  - Each Event Data Product in its own TBranch of the TTree
- ROOT with one blob per Event
  - Explicitly do object serialization concurrent
  - Store all Data Products for an Event into one blob
  - Compress either external (concurrently) or internal (serially) to ROOT
- PDS Simple per Event storage format
  - 'Packed Data Stream' and originated for the CLEO experiment
  - Designed to support concurrency
  - stores events sequentially in the file
  - and supports per event compression(i.e. each Event in the file is independent of all other Events in the file)

HDF5

- HDF5 (Hierarchical Data Format) is a portable, self-describing file format designed to store large amounts of data
  - It is maintained by the HDF Group [https://www.hdfgroup.org]
  - It is widely available at HPC centers, and easily installable on laptops
  - It supports parallel IO using MPI, and has special drivers tuned for parallel file systems at HPC centers
- A few key abstractions are:
  - datasets, which are multidimensional arrays of homogeneous types,
  - groups, which are containers of datasets and other groups, and
  - attributes, which are small metadata objects to describe groups and datasets
- Allows efficient columnar data access for the “required” data products

HDF5 format for the Testing Framework

- The input data (intermediate output) is already serialized with ROOT.
  - Complex objects but presented as byte stream to HDF5
  - No “direct” use of C++ classes that represent data products
  - Not working with tabular/analysis-ready data
- An experiment-independent approach is designed, so there is no experiment specific assumptions in the implementation.
  - Will need experiment-specific set up to run it to make sure ROOT dictionaries are available

Two design options implemented

- Data product based
- Event based

Data products based HDFOutputer

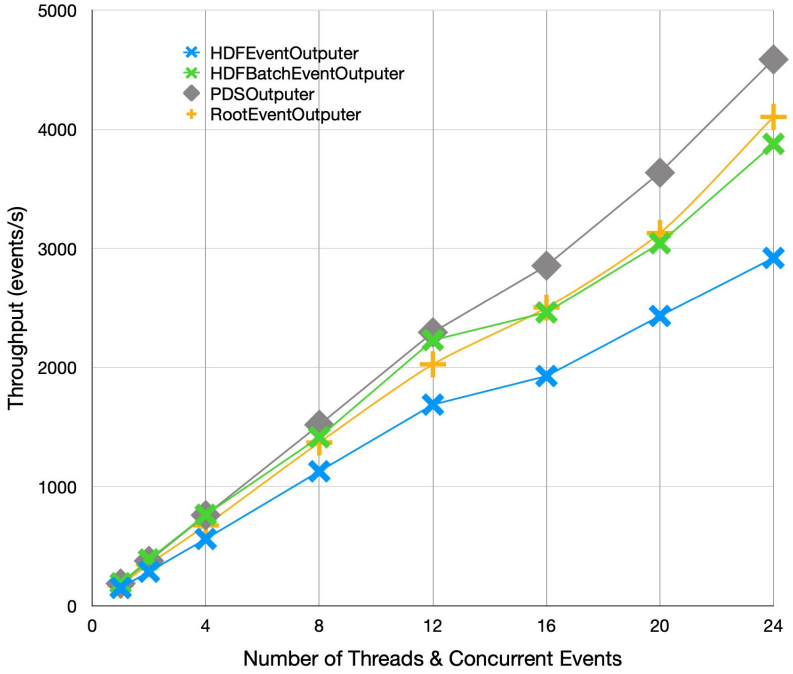
- Developed HDF5 output module that uses 1D dataset of chars for each data product, and a corresponding 1D dataset for size/offset per event.
  - Write happens in batches; a batch corresponds to number of events that are aggregated before a write happens
  - Pros: Ability to only read required dataset
  - Cons: Two data sets per data product generates a lot of metadata especially when there are thousands of data products in an event
- Improvement: Reduce the number of I/O calls on meta-data related data-sets.
  - Collect the metadata for all datasets for each event and save once.
  - Once tested more thoroughly, port to the HDFOutputer.

Event based HDFOutputer

- Store events as a blob in a single event dataset
  - Pros: Number of datasets is reduced to 3
  - Cons: Always read full event even if not needed
- Both implementations are general and rather straightforward to adopt for incorporating parallel design

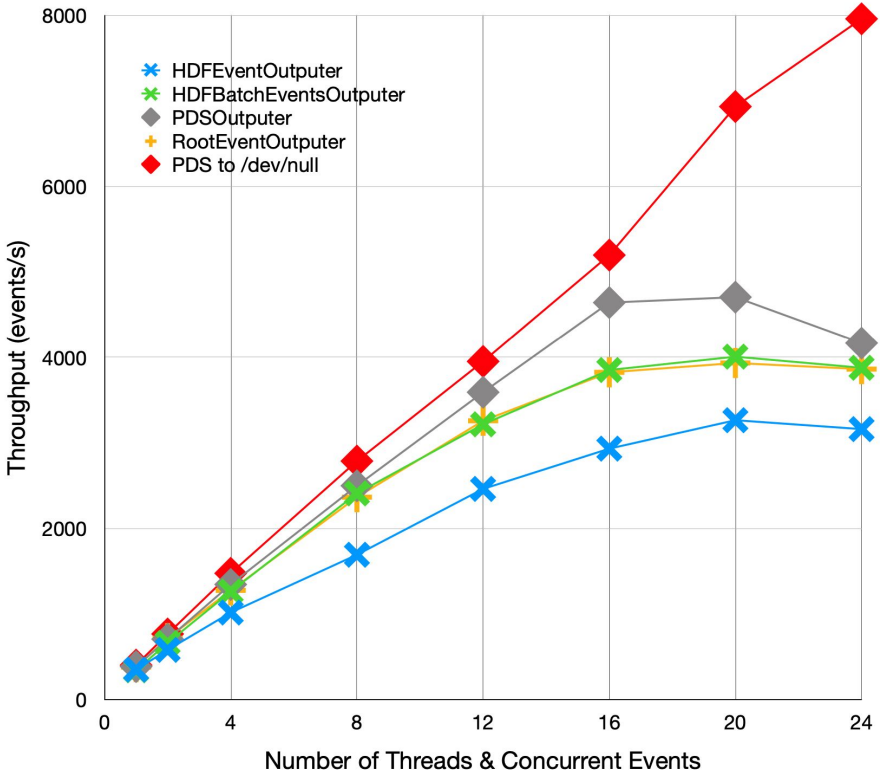
Comparing different data formats (Local VM with 24 threads)

- All are showing good scaling
  - Filling machine with jobs
- Noticed Items
  - If look at total MB/s of all concurrent jobs
    - All threads are at ~475 MB/s
  - Concurrent single threaded jobs are not running at 100% CPU utilization
- Hit disk write limit?



Comparing different data formats - Single job

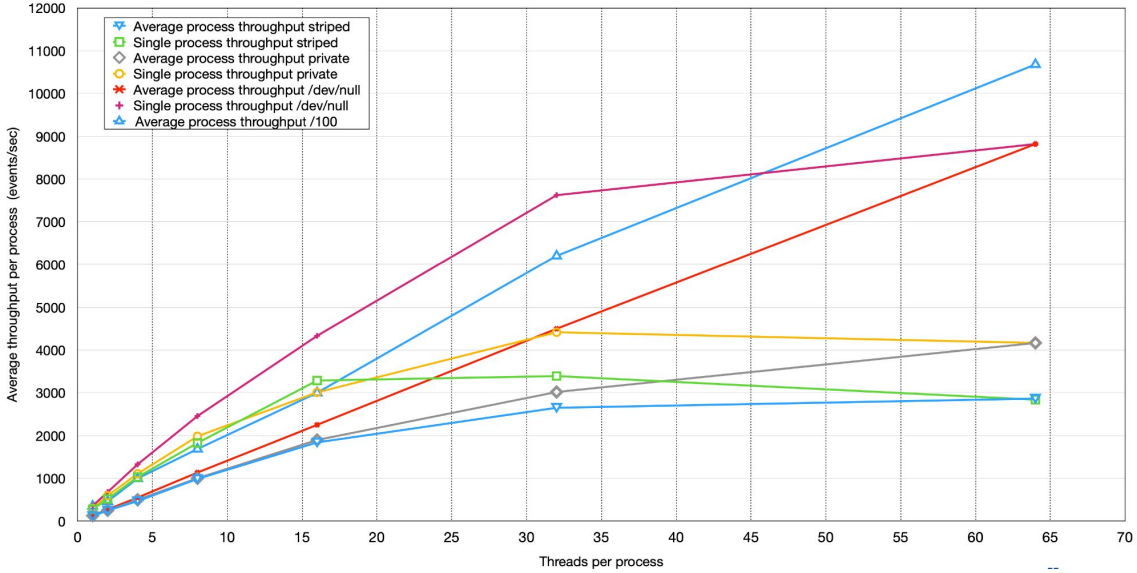
- Run only 1 job at a time
- Substantial throughput increase
- Scaling limiting components
  - Max disk write rate
  - Format specific serial time
  - Time needed for the format API to marshal through data
- Need better machine
  - More cores
  - Faster storage



Running tests on HPC

- As noticed, the throughput might be limited by the write speed of the local hard disk and the number of threads on the test VM — 24
  - Can writing to Burst Buffer on Cori nodes give faster throughput?
  - Can throughput continues scaling up to the 64 threads available on a Cori Haswell node?
- Non-trivial to set up and run any experiment code on HPC machines
  - Haven't even tried running with ATLAS or DUNE data files on Cori, but have locally used CMS, ATLAS and DUNE data during development
- We can run the test framework with CMS reco files, and are evaluating performance of currently available IO modes
  - PDS throughput is included

PDSOutputer throughput on Cori



Ran with PDSOutputer on Haswell node and burst buffer 10Gb striped

Adding MPI support to Root serialization

- Created an MPI-based version of the testing framework main application
- Goal is to be able to evaluate multi-node performance as well as pave the way for parallel IO using HDF5
- Current supported modes:
  a. N MPI ranks able to read N input files and write N output files, trivial, running multiple processes of root serialization all through MPI (any input/output mode combination is supported)
  b. N MPI ranks reading 1 input file and write N output files (any input/output mode combination should work logically)
  c. N MPI ranks reading N files and writing 1 output file (will only be supported for HDF5) → In progress

Parallel HDF5 approach

- N number of MPI ranks participate in the reading of file(s) and writing to a single HDF5 file collectively (parallel across a single dataset).
- Writing a file collectively has the advantage that final file might not need merging and less resource dedicated to merge the output files.
- Extend serial design (both data product-based and event-based) to update starting index per dataset for each rank.
- Use of MPI Functionalities to communicate with various processes to exchange relevant information such as data sizes for collective IO.
- Use of Parallel-HDF5 to do the reading and writing by various MPI nodes collectively.
  - Test improvements from serial HDF5 design to do collective I/O on writing into the HDF File.

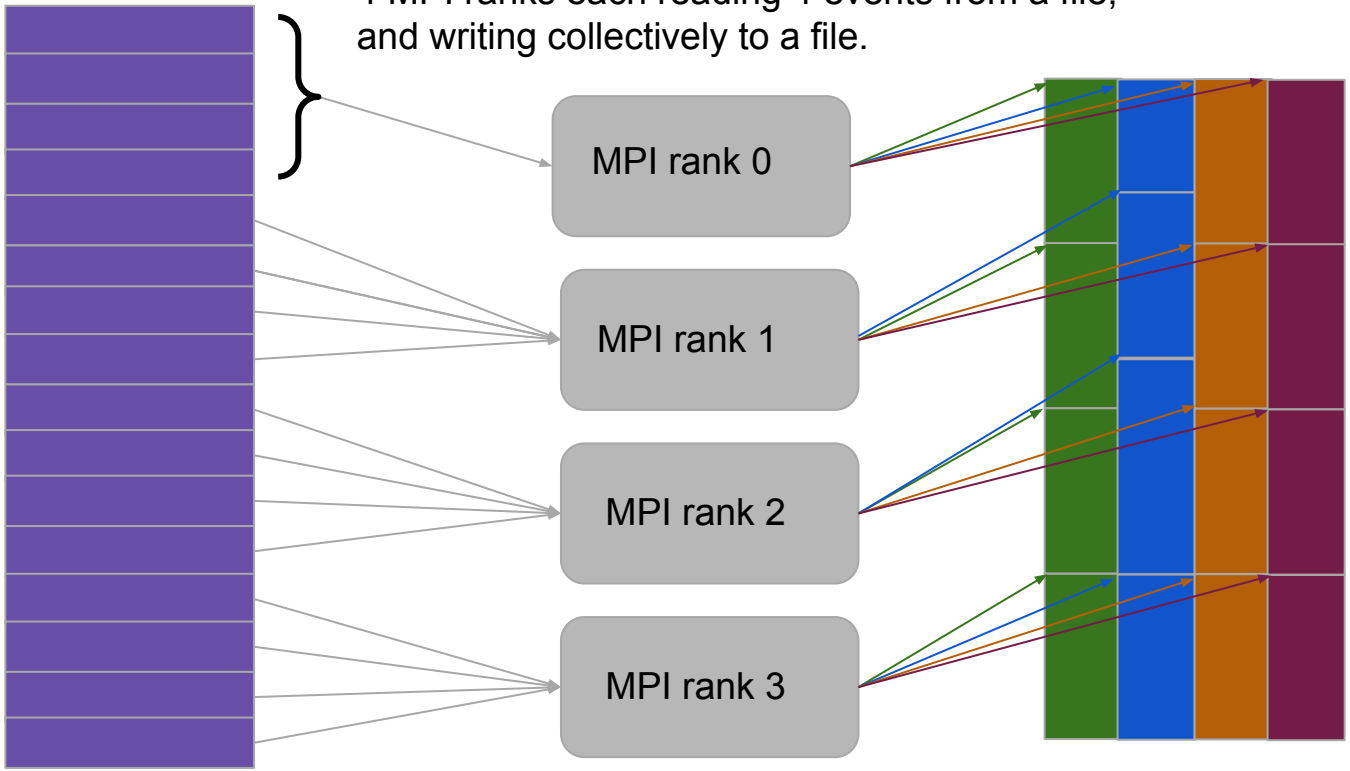
Future work

- Complete performance evaluation studies on Cori using different data formats in serial mode
  - Studies with HDF5 tuning in serial mode
    - Chunking: Looked at some prelim numbers, using 1MB chunk size
    - Asynchronous I/O: Use a background thread to perform I/O
- Complete Parallel IO (using multi-process MPI-based writes) integration to the testing framework and performance evaluation
- Compression
  - In both serial and parallel mode, and combined with chunking
- Different HDF5 layout (serial) if needed based on performance outcome
  - Compound data types
- Using node-local storage for storing intermediate HDF5 files
- Multi-threaded HDF5
  - There is a feature branch available with simple read/write patterns, which we have in our use case, that we should look into
- Explore direct storage access from GPUs

Back up slides

1 input file with 16 events

4 MPI ranks each reading 4 events from a file, and writing collectively to a file.



The colorful blocks show one HDF5 file. Each color represents a different dataset corresponding to a data product. There are four data products. Each product per event may have different sizes. What is not shown for simplicity is offsets and Event ID datasets here. All the green arrows represent parallel write to the first dataset, then all the blue arrows represent the second parallel write that happens after the first write.

With event blob approach, this design becomes much simpler.

Write Performance testing

Used CMS's smallest data format

Minimized time spent in reading

- Read first 100 events and then cache them to memory
- Replayed cached events over and over

Kept number threads == number of concurrent events

- Intra event concurrency helps when serialization happens

Kept all cores of machine busy

- # jobs = (# cores on machine) /(# threads in job)

Write Performance Testing Formats

Standard HEP ROOT format

- Events stored as an element in a TTree
- Each Event Data Product in its own TBranch of the TTree

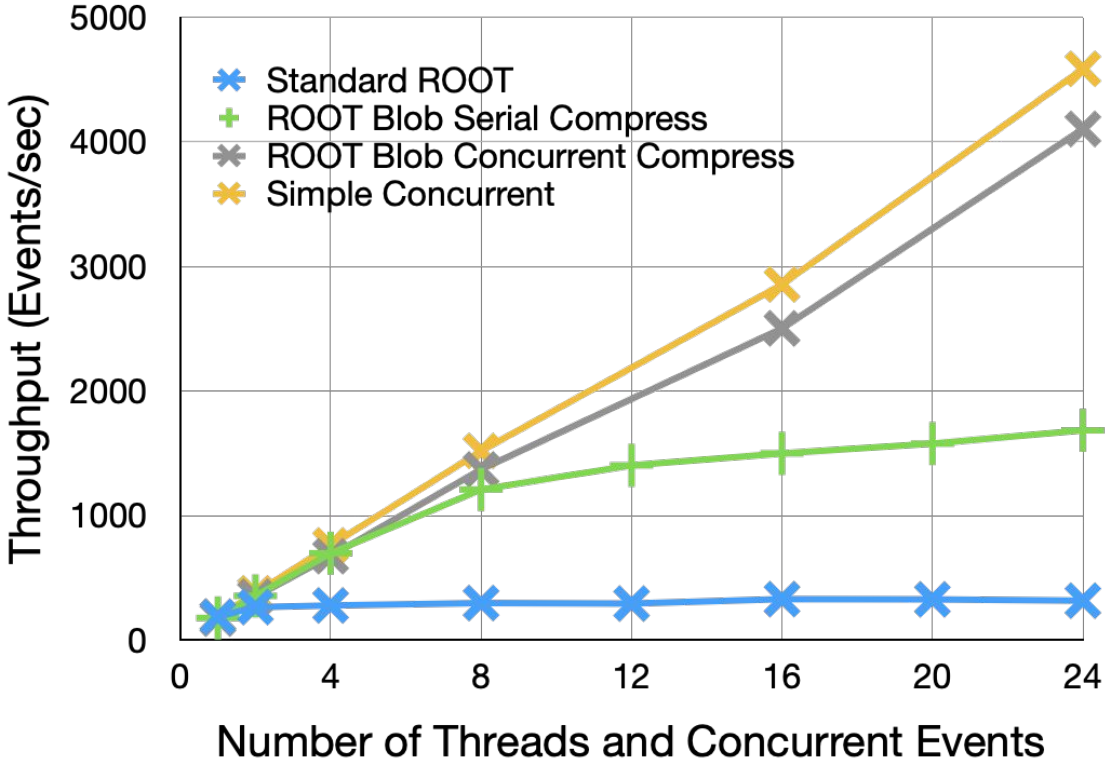
Simple per Event storage format

- Designed to support concurrency

ROOT with one blob per Event

- Explicitly do object serialization concurrent
- Store all Data Products for an Event into one blob
- Compress either external (concurrently) or internal (serially) to ROOT

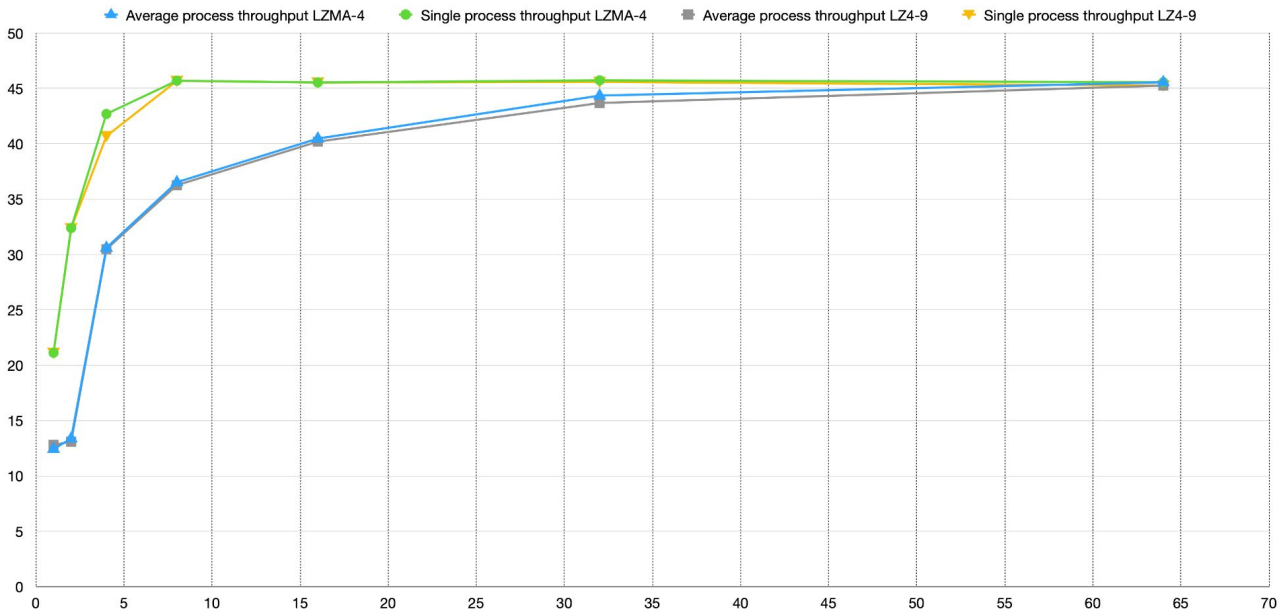
Preliminary LZ4 Compression Write Results



Throughput with Rootoutputer on Cori

- Expected to plateau as thread count increases
- Measured with IMT enabled, splitlevel=99, LZ4 compression level 9
- Measured with IMT enabled, splitlevel=99, LZMA compression level 4 which are the setting CMS uses in production.
- Used Cori haswell node, fully loaded and with single process.

Results on Cori



Design details HDFSource

- Read in the HDF5 data; only supports data product based design approach
- Read one event at a time
  - Locate start index of a data product and calculate end index using EventID dataset and offsets datasets
- Able to read events randomly since we can index into datasets as needed
- Next steps:
  - Performance evaluations
  - HDFSource for reading event blob format