HDF5 as intermediate storage for HPC processing

Saba Sehrish for the CCE IOS team

HEP CCE All-hands Meeting Meeting October 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 HEP data processing on HPC?

Current status

Major Activities	Status
Multithreaded Test framework development	Complete
Serial output and input modules development and evaluation	Complete
MPI extension and Parallel HDF5 output modules development and evaluation	In progress

In the last all hands meeting in April, we discussed multithreaded test Framework, comparisons of different output modes, and shared initial design approach towards parallel design.

Recap: Adding MPI support to the Root serialization test framework for Parallel HDF5 design

- Created an MPI-based version of the testing framework
- Goal is to be able to evaluate multi-node performance and developing parallel HDF5 output modules
- 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 (only supported for HDF5)

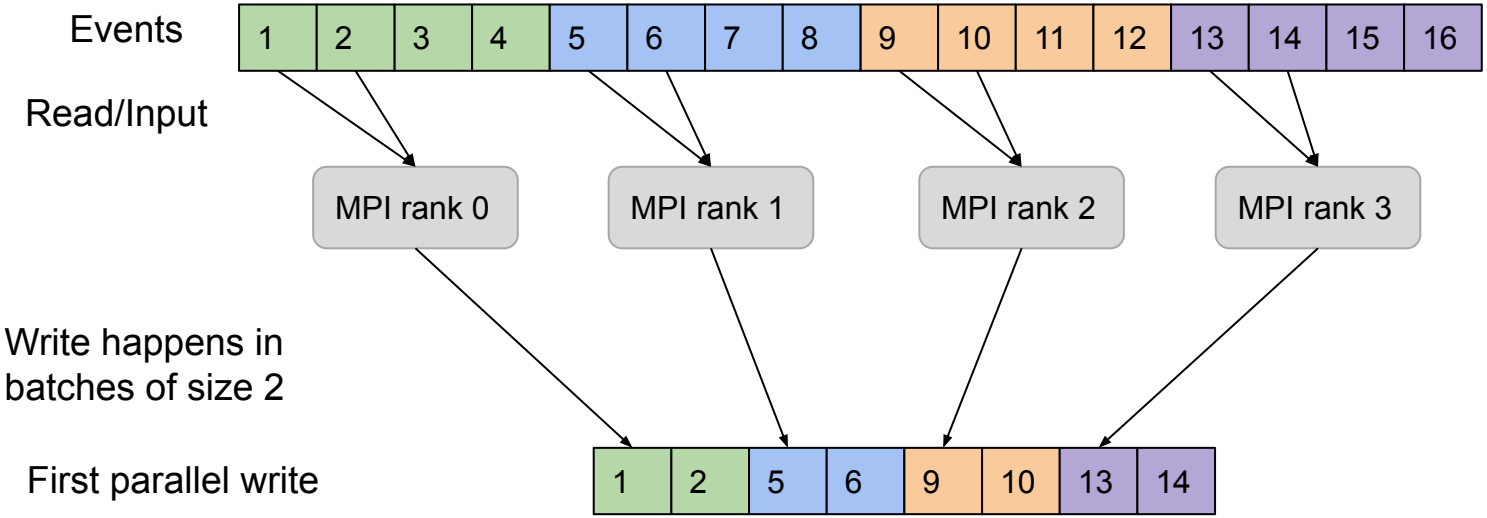
Parallel HDF5 approach

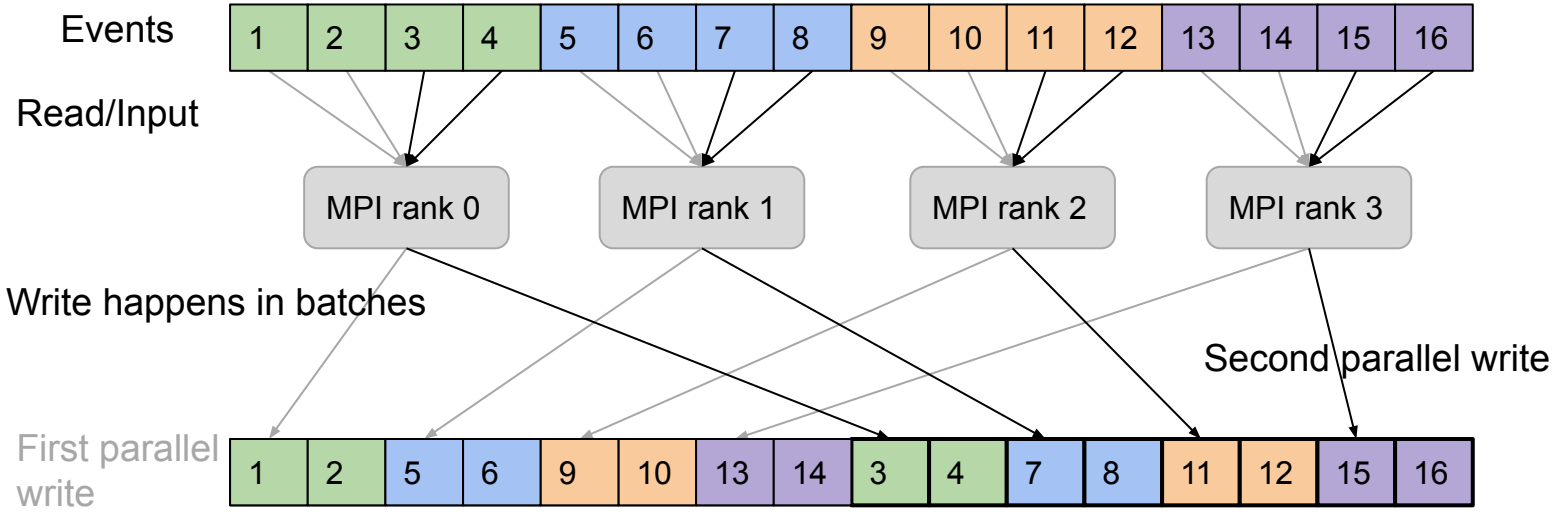
- N number of MPI ranks participate in the reading of file(s) and write to a single HDF5 file collectively (i.e. parallel write across a single dataset).
  - Writing a file collectively has the advantage that the final file might not need merging.
- The Event batch based serial HDF5 output module design seemed to be the best performing HDF5 design, hence we used that design as the basis of parallel HDF5 design.
  - Store events as blobs in a single HDF5 dataset, and aggregate events before writing
- The key feature in the parallel design is event distribution approach among MPI ranks and the use of MPI functions to exchange relevant information such as data sizes and offsets to coordinate for collective IO.

Approach 1: Number of events is known

Each MPI rank knows the total number of events to process and the batch size (i.e. how many to aggregate).

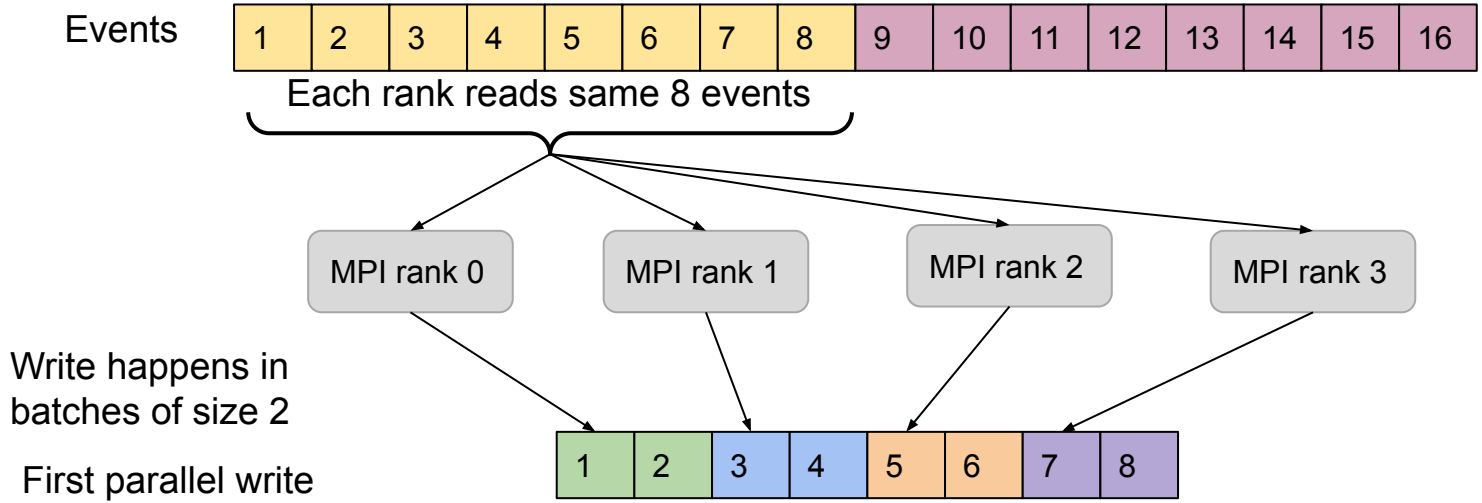
- Equally distribute events among MPI ranks
  - If not equally divisible, or every rank does not have same number of batches, adjust number of events to do so
- Each rank then calculates the starting event and number of events it will be processing
  - Each rank only reads the events it will write, except in the case where its not equally divisible, then it reads more but writes only that is requested
- Each rank reads the events and continue to accumulate until reaches batch size, then it is ready to write
- Using MPI_Scan and MPI_Reduce, exchange information with other ranks to determine the offset it will be writing to
- Each rank extends the dataset accordingly and makes the H5Dwrite call

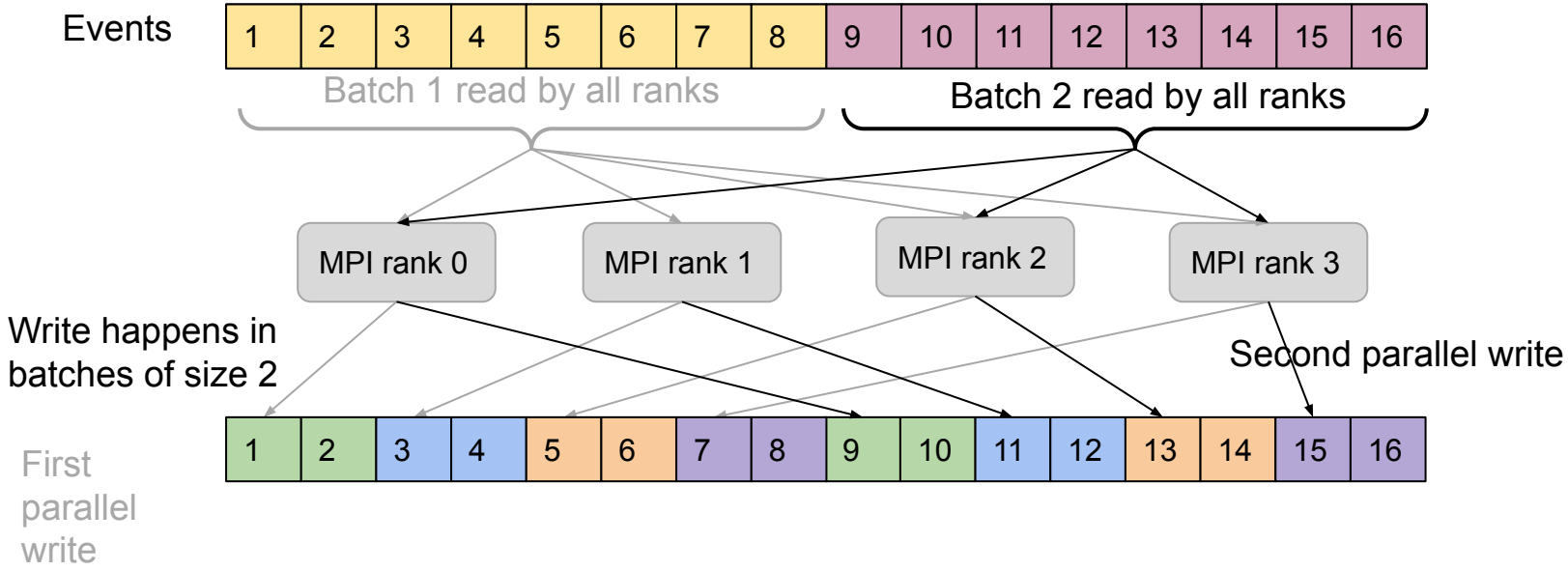

Events
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Read/Input
MPI rank 0
MPI rank 1
MPI rank 2
MPI rank 3
Write happens in batches of size 2
First parallel write
1
2
5
6
9
10
13
14


Events
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Read/Input
MPI rank 0
MPI rank 1
MPI rank 2
MPI rank 3
Write happens in batches
Second parallel write
First parallel write
1 2 5 6 9 10 13 14 3 4 7 8 11 12 15 16

Approach 2: Number of events is unknown

- A realistic use case, where at the beginning of the processing, number of events is unknown; processing continues until end of file or no more events to process
- Each rank knows the number of events it can write at a time
- For every batch write, each rank reads in the number of events that is equal to the batch size across all ranks.
- No adjustments needed in reading but we do read more data per batch write.


Events
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Each rank reads same 8 events
MPI rank 0
MPI rank 1
MPI rank 2
MPI rank 3
Write happens in batches of size 2
First parallel write
1
2
3
4
5
6
7
8


Events
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Batch 1 read by all ranks
Batch 2 read by all ranks
MPI rank 0
MPI rank 1
MPI rank 2
MPI rank 3
Write happens in batches of size 2
Second parallel write
First parallel write

Sample HDF5 output file

```
HDF5 "sample.h5" {
GROUP "/" {
  GROUP "Lumi" {
    ATTRIBUTE "BranchListIndexes" {
      DATATYPE  H5T_STRING {
        STRSIZE H5T_VARIABLE;
        STRPAD H5T_STR_NULLTERM;
        CSET H5T_CSET_UTF8;
        CTYPE H5T_C_S1;
      }
      DATASPACE  SCALAR
    }

 ......

    ATTRIBUTE "run" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SCALAR
    }
    ATTRIBUTE "uint_bunchSpacingProducer__RECO." {
      DATATYPE  H5T_STRING {
        STRSIZE H5T_VARIABLE;
        STRPAD H5T_STR_NULLTERM;
        CSET H5T_CSET_UTF8;
        CTYPE H5T_C_S1;
      }
      DATASPACE  SCALAR
    }
    DATASET "EventIDs" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 320000 ) / ( H5S_UNLIMITED ) }
    }
    DATASET "Offsets" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 35840000 ) / ( H5S_UNLIMITED ) }
    }
    DATASET "Products" {
      DATATYPE  H5T_STD_I8LE
      DATASPACE  SIMPLE { ( 23943467427 ) / ( H5S_UNLIMITED ) }
    }
  }
}
}
```

Run number, etc also stored as attributes

Data product information stored as attributes

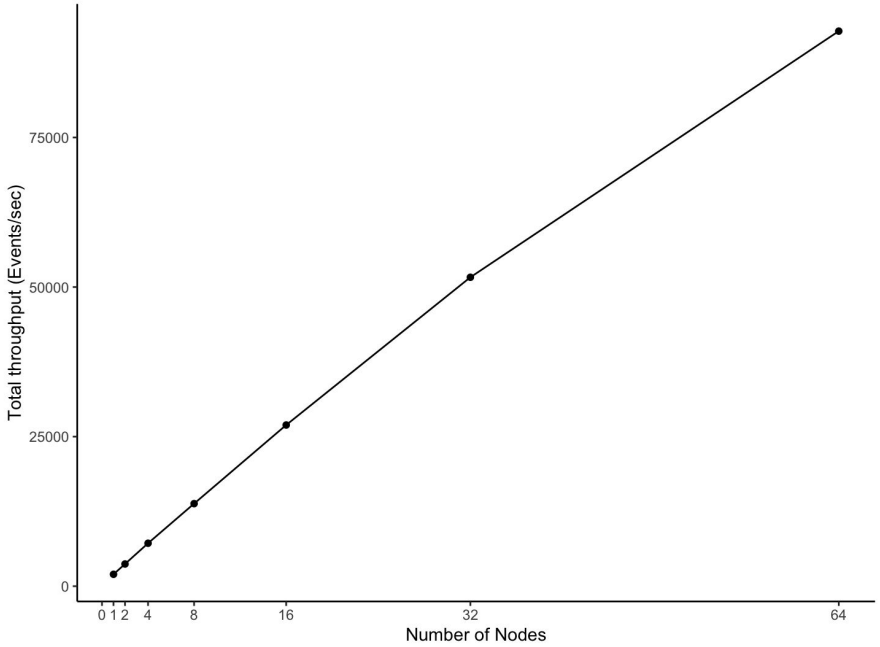
Three data sets; one for event IDs, one for storing offsets, and the last one for the products data itself

Performance studies on Cori

- Used the root serialization test framework in MPI mode; kept HDF5 chunk size, batch size constant
- Used Cori haswell nodes with burst buffers
- Scaling tests done by increasing the number of nodes
- MPI ranks varied from 1- 4 per node
- The number of threads per node kept at 64
- Increased number of events with number of nodes.
  - Number of events 320K to 20 million
- Generated output file size ranges: 25GB - 1.6 TB
- Used CMS data file
- Burst buffer stripped up to 2 TB space requested (100 burst buffer nodes)

Total throughput (Events/sec)

- Total throughput calculated using event processing time and number of events
- Performance scaling with the number of nodes
- Using 4 ranks per node (16 threads/rank) gave the best performance, ~51% improvement as compared with 1 rank per node.
- ~97K events/sec is the maximum throughput observed for 256 ranks on 64 nodes.
- This translates to 8GB/s bandwidth we were able to consume for the largest set.



IO time breakdown

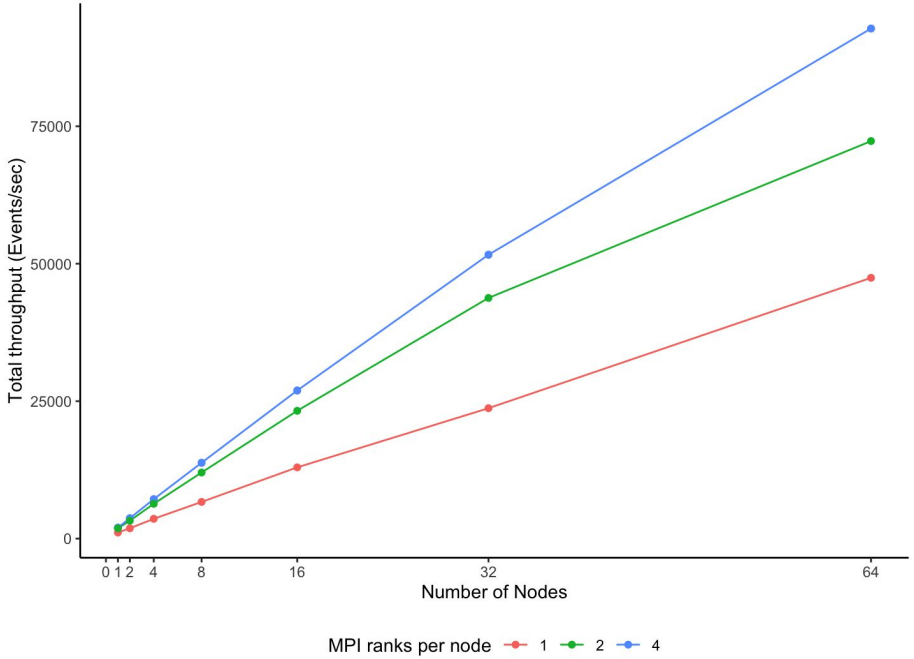
- Time spent in MPI Scan calls to coordinate offsets in the datasets is ~9% of the total IO time
- Time spent in MPI Reduce calls for the final offset is ~5% of the total IO time
- Time spent in actual HDF5 write dataset calls is ~32%
- There are other calls that we did not time, for example creating/opening HDF5 file, writing file header that includes writing hundreds of attributes (115 for the data used in tests), inquire about current data space, extend/resize datasets, etc

Next steps and plan for FY23

- Complete performance evaluation studies on Cori
  - Understand IO behavior, untimed calls that are actually constituting 50% of the IO time, use Darshan logs
- Move evaluation studies to Perlmutter soon
- HDF5 options to work with
  - Asynchronous IO
  - Multi dataset; we only have 3 datasets though
- Write a paper/report

Backup material

Throughput for all three cases



Future work

- Storing C++ objects (using ROOT's reflection) directly to HDF5 without serialization
- Multi-threaded HDF5
- Any use of Subfiling for our work?
- HDF5 streaming services
- Explore direct storage access from GPUs
- Others not directly HDF5
  - RNtuple
  - C++ object design studies